**Python OOP Assignment**

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

Ans :- Programming languages are frequently distinguished by their programming paradigm. A programming paradigm is one way of viewing and interacting with data. Object-Oriented and Functional are the two fundamental paradigms. Object-Oriented Programming is based on data structures called objects, which hold both data (properties or attributes) and code (procedures or methods). Python Object-oriented programming (OOP) is a method of organizing a program by grouping together related characteristics and behaviors into separate objects. With the help of examples, we will learn about Python Object-Oriented Programming (OOP) and its core concepts in the article below.

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

Ans :- The whole point of a namespace tool like the class statement is to support name inheritance. In Python, inheritance happens when an object is qualified, and involves searching an attribute definition tree (one or more namespaces). Every time you use an expression of the form object.

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

Ans :- Object is an instance of a class. Class is a blueprint or template from which objects are created. Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. Class is a group of similar objects.

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

Ans :- Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

Q5. What is the purpose of the init method?

Ans :- \_\_init\_\_ is one of the reserved methods in Python. In object oriented programming, it is known as a constructor. The \_\_init\_\_ method can be called when an object is created from the class, and access is required to initialize the attributes of the class.

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

Ans :- Every object has a type and the object types are created using classes. Instance is an object that belongs to a class. For instance, list is a class in Python. When we create a list, we have an instance of the list class.

Q7. What is the process for creating a class?

Ans :- A Class is like an object constructor, or a "blueprint" for creating objects.

To create a class, use the keyword class:

**Example**

Create a class named MyClass, with a property named x:

class MyClass:  
  x = 5

Q8. How would you define the superclasses of a class?

Ans :-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. Superclasses are sometimes called ancestors as well.

Q9. What is the relationship between classes and modules?

Ans :- Classes in python are templates for creating objects. They contain variables and functions which define the class objects. At the same time, 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.

Q10. How do you make instances and classes?

Ans :- To create instances of a class, you call the class using class name and pass in whatever arguments its \_\_init\_\_ method accepts.

Q11. Where and how should be class attributes created?

Ans :- Class attributes are the variables defined directly in the class that are shared by all objects of the class. Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor. Defined directly inside a class.

Q12. Where and how are instance attributes created?

Ans :- To define a class attribute, you place it **outside of the \_\_init\_\_() method**. Use class\_name. class\_attribute or object\_name. class\_attribute to access the value of the class\_attribute .

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

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

Q14. How does a Python class handle operator overloading?

Ans :- The operator overloading in Python means provide extended meaning beyond their predefined operational meaning. Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists. We can achieve this as the "+" operator is overloaded by the "int" class and "str" class.

Q15. When do you consider allowing operator overloading of your classes?

Ans :- The decision to allow operator overloading in a Python class is typically made by the developer or team responsible for designing and implementing the class. Operator overloading can be useful for making code more readable and expressive, but it can also make code more complex and harder to understand if not used carefully. Ultimately, it's up to the developer to decide whether or not to use operator overloading and how to implement it in a way that is consistent with the design and goals of the class.

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

Ans :- In Python, the most common form of operator overloading is the use of special methods to define behavior for operators. These methods have names such as **\_\_add\_\_**, **\_\_mul\_\_**, **\_\_eq\_\_**, etc. and are used to define the behavior of the corresponding operators (**+**, **\***, **==**, etc.) when they are used with instances of the class. For example, if you define a **\_\_add\_\_** method in a class, that method will be called when the **+** operator is used with instances of that class.

Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?

Ans :- 1. Classes and objects

2. Inheritance and polymorphism

Q18. Describe three applications for exception processing.

1. Ans :- Input validation: Exception processing can be used to validate user input and ensure that it meets certain conditions. For example, a program that reads a number from the user might use exception processing to catch a **ValueError** that is raised if the user enters a non-numeric value. This can help to prevent errors and crashes later on in the program.
2. File handling: Exception processing can be used to handle errors that occur when working with files. For example, a program might use exception processing to catch a **FileNotFoundError** that is raised if a specified file cannot be found, or a **PermissionError** that is raised if the program does not have permission to access the file.
3. Network communication: Exception processing can be used to handle errors that occur when communicating over a network. For example, a program that sends a request to a server might use exception processing to catch a **ConnectionError** that is raised if the server cannot be reached, or a **TimeoutError** that is raised if the request takes too long to complete.

Q19. What happens if you don't do something extra to treat an exception?

Ans :- If you don't do anything to handle an exception in Python, the program will terminate and an error message will be displayed. This message will include the name of the exception that was raised and a traceback, which shows the sequence of function calls that led to the exception. This can be useful for debugging, but it can also be confusing or overwhelming for the user.

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

Ans :- There are several options for recovering from an exception in a Python script:

1. **try-except** block: The most common way to handle exceptions in Python is to use a **try-except** block. The code that might raise an exception is placed in a **try** block, and the code to handle the exception is placed in an **except** block. If an exception is raised, the program will jump to the corresponding **except** block and execute the code there. This allows you to handle the exception and continue executing the program.
2. **try-except-else** block: You can also use a **try-else** block to specify code that should be executed if no exception was raised in the try block. This is useful when you want to perform some cleanup or other action in the case of successful execution, but you don't want it to be executed if an exception occurred.
3. **try-except-finally** block: You can also use a **try-finally** block to specify code that should always be executed, regardless of whether an exception occurred or not. This is useful when you want to perform some cleanup or other action after the try and except block.
4. **raise** statement : You can raise an exception again using the **raise** statement. This can be useful when you want to catch an exception, handle it, and then raise it again so that it can be handled by the next level of the call stack.
5. **assert** statement: You can use the **assert** statement to check if a certain condition is true, and if it is not true, raise an exception. This can be useful for validating input or other data, and can make it easier to understand the cause of an exception when debugging.

Q21. Describe two methods for triggering exceptions in your script.

1. Ans :- **raise** statement: The most common way to trigger an exception in Python is to use the **raise** statement. This statement allows you to raise an exception and specify the type of exception and an optional error message. For example, you can raise a **ValueError** exception with a message like "Invalid value" by using the following code: **raise ValueError("Invalid value")**.
2. Built-in functions: Python provides a number of built-in functions that raise exceptions when certain conditions are not met. For example, the **assert** statement raises an **AssertionError** if the condition it is testing is not true. The **open()** function raises a **FileNotFoundError** if the specified file cannot be found, and the **int()** function raises a **ValueError** if it is passed a non-numeric string.

Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

1. Ans :- **finally** clause : One way to specify actions that should be executed at termination time, regardless of whether or not an exception exists, is to use a **finally** clause. The code in a **finally** clause will always be executed, whether an exception is raised or not, and regardless of whether the exception is handled or not.
2. **atexit** module: Another way to specify actions that should be executed at termination time is to use the **atexit** module. This module provides a way to register functions that will be called when the program exits, regardless of whether the exit was caused by an exception or not.

Q23. What is the purpose of the try statement?

Ans :- The **try** statement in Python is used to enclose a block of code that might raise an exception. The code in the **try** block is executed, and if an exception is raised, the program jumps to the corresponding **except** block and executes the code there.

Q24. What are the two most popular try statement variations?

1. Ans :- **try-except** block: The **try-except** block is the most basic and common variation of the **try** statement. It is used to enclose a block of code that might raise an exception and provide a way to handle the exception if it is raised.
2. **try-finally** block: The **try-finally** block is used to enclose a block of code that might raise an exception and provides a way to specify code that should always be executed, regardless of whether an exception occurred or not.

Q25. What is the purpose of the raise statement?

Ans :- The **raise** statement in Python is used to raise an exception. The **raise** statement allows you to raise a specific exception and to specify an optional error message.

The purpose of the **raise** statement is to signal that something unexpected or exceptional has occurred, and that the program should take some action to handle the exception and continue executing. It is used to signal a problem or an error that the script cannot continue to execute.

Q26. What does the assert statement do, and what other statement is it like?

Ans :- The **assert** statement in Python is used to check if a certain condition is true, and if it is not true, raise an **AssertionError** exception. The **assert** statement is used to ensure that certain assumptions about the state of the program are true and to signal when they are not true.

Q27. What is the purpose of the with/as argument, and what other statement is it like?

Ans :- The **with** statement in Python is used to create a context in which a specific action is performed and then to automatically clean up resources when the action is finished. The **with** statement is used in conjunction with an object that defines the **\_\_enter\_\_()** and **\_\_exit\_\_()** methods.

Q28. What are \*args, \*\*kwargs?

Ans :- In Python, **\*args** and **\*\*kwargs** are used to allow a function to accept a variable number of arguments.

**\*args** is used to pass a variable number of non-keyword arguments to a function. It is used to pass a tuple of arguments

**\*\*kwargs** is used to pass a variable number of keyword arguments to a function. It is used to pass a dictionary of arguments.

Q29. How can I pass optional or keyword parameters from one function to another?

Ans :- In Python, you can pass optional or keyword parameters from one function to another by passing them as arguments to the second function.

When passing parameters that have a default value, you can simply call the second function with the desired arguments, and the function will use the default values for any arguments that are not provided

Q30. What are Lambda Functions?

Ans :- Lambda functions are small, anonymous functions in Python that can be defined using the keyword **lambda**. They are often used as inline function definitions, and can be passed as arguments to other functions, such as the built-in **filter()** and **map()** functions. Lambda functions are defined using the following syntax: **lambda arguments: expression**. The expression is evaluated and returned when the function is called. Lambda functions are also known as anonymous or inline functions as they are not bound to a name at runtime.

Q31. Explain Inheritance in Python with an example?

Ans :- Inheritance is a mechanism in object-oriented programming where a class can inherit properties and methods from a parent class. The class that inherits the properties and methods is called the derived class or child class, and the class from which the properties and methods are inherited is called the base class or parent class.

class Vehicle:

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

def print\_details(self):

print(f'Make: {self.make}, Model: {self.model}')

class Car(Vehicle):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model)

self.year = year

def print\_details(self):

super().print\_details()

print(f'Year: {self.year}')

my\_car = Car('Toyota', 'Corolla', 2020)

my\_car.print\_details()

Q32. 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?

Ans :- In this case, if we call the **func()** method on an object of class C, Python will first check if the method is present in class A, if yes it will invoke the version of **func()** defined in class A, if not it will check in class B and invoke the version of **func()** defined in class B.

Q33. Which methods/functions do we use to determine the type of instance and inheritance?

Ans :- there are a few built-in methods and functions that can be used to determine the type of an instance and check for inheritance:

1. **type()** function: This function returns the type of the given object. For example, **type(obj)** will return the type of the object 'obj'.
2. **isinstance()** function: This function returns **True** if the given object is an instance of the specified class or a subclass of it. For example, **isinstance(obj, MyClass)** will return **True** if the object 'obj' is an instance of the class 'MyClass' or a subclass of it.
3. **issubclass()** function: This function returns **True** if the specified class is a subclass of the given class. For example, **issubclass(MyDerivedClass, MyBaseClass)** will return **True** if 'MyDerivedClass' is a subclass of 'MyBaseClass'.
4. **\_\_class\_\_** attribute: Every object in python has a **\_\_class\_\_** attribute which returns the class of that object.
5. **\_\_bases\_\_** attribute: Every class has a **\_\_bases\_\_** attribute that returns a tuple of base classes of that class.
6. **\_\_subclasses\_\_()** method: This method returns a list of the immediate subclasses of a class. For example, **MyClass.\_\_subclasses\_\_()** will return a list of the immediate subclasses of the class 'MyClass'.

Q34.Explain the use of the 'nonlocal' keyword in Python.

Ans :- The **nonlocal** keyword in Python is used to indicate that a variable is not a local variable, but is instead a non-local variable that is defined in an outer scope.

In Python, when a function is defined within another function, the inner function can access the variables defined in the outer function, but if we want to modify the value of a variable defined in the outer function from the inner function, we need to use the **nonlocal** keyword.

Q35. What is the global keyword?

Ans :- the **global** keyword is used to indicate that a variable is a global variable and not a local variable. A global variable is a variable that is defined outside of any function or class and can be accessed from anywhere in the program.