OOP Viva Q/A

1. Question: What is Object-Oriented Programming (OOP) and how does it differ from procedural programming?

Answer: Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects that have data attributes and methods to manipulate the data. It differs from procedural programming in that OOP focuses on the creation of objects and their interactions, while procedural programming focuses on procedures or functions that perform specific tasks.

2. Question: Explain the concept of classes and objects in Python, and provide an example of creating a class and its instances.

Answer: In Python, a class is a blueprint for creating objects. It defines attributes and methods that are shared by all instances of the class. An object is an instance of a class. For example:

```python
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
```

3. Question: What is encapsulation in OOP, and how is it achieved in Python? Give an example of a class with private and protected attributes.

Answer: Encapsulation is the bundling of data (attributes) and methods that operate on the data within a class. It is achieved in Python by using access control modifiers like underscores (_) to make attributes private and protected. Example:

```python
class Student:
    def __init__(self, name, age):
        self._name = name  # Protected attribute
        self.__age = age   # Private attribute
```

```
    def get_age(self):
        return self.__age

student1 = Student("Alice", 20)
```

4. Question: How do you create an abstract class in Python? Explain the purpose of abstract methods and how they are implemented in subclasses.

Answer: An abstract class in Python is created by using the `ABC` (Abstract Base Class) module. It contains one or more abstract methods, which are declared but not implemented in the abstract class. Subclasses must implement these abstract methods. Example:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

5. Question: Describe the concept of inheritance in OOP. Provide an example of single inheritance and multiple inheritance in Python.

Answer: Inheritance is a feature of OOP where a class (subclass) can inherit the attributes and methods of another class (superclass). It promotes code reuse and allows creating specialized classes. Example of single inheritance:

```python
class Animal:
```

```
    def __init__(self, name):
        self.name = name
class Dog(Animal):
    def bark(self):
        return "Woof!"
d = Dog("Buddy")
print(d.name)  # Output: Buddy
print(d.bark())  # Output: Woof!
```

Example of multiple inheritance:

```python
class Bird:
    def __init__(self, species):
        self.species = species

    def fly(self):
        return "I can fly!"
class Penguin(Bird, Animal):
    def __init__(self, name, species):
        Animal.__init__(self, name)
        Bird.__init__(self, species)
```

6. Question: How can you achieve method overriding in Python? Explain the use of the `super()` function in this context.

Answer: Method overriding is the process of providing a new implementation for a method in the subclass that already exists in the superclass. It is achieved by defining a method with the same name in the subclass. The `super()` function is used to call the overridden method of the superclass. Example:

```python
class Animal:
    def speak(self):
```

```
        return "Animal sound"


class Dog(Animal):
    def speak(self):
        return "Woof! " + super().speak()
d = Dog()
print(d.speak())  # Output: Woof! Animal sound
```

7. Question: Discuss the concept of polymorphism and how it is implemented using method overloading and method overriding.

Answer: Polymorphism is the ability of a method to take on multiple forms based on the number or type of parameters passed to it. Method overloading and method overriding are two ways to achieve polymorphism in Python. Method overloading allows defining multiple methods with the same name but different parameters, while method overriding allows providing a new implementation for a method in the subclass that already exists in the superclass.

8. Question: Explain the difference between class methods and instance

methods in Python. When would you use each type of method?

Answer: Class methods are defined using the `@classmethod` decorator and take the class (cls) as the first parameter instead of the instance (self). They can be used to access or modify class-level attributes. Instance methods, on the other hand, take the instance (self) as the first parameter and can access or modify instance-specific attributes.

9. Question: What is the role of the `__init__()` method in a Python class? How does it relate to the concept of constructors?

Answer: The `__init__()` method is a special method in Python classes that is called automatically when an object is created from the class. It initializes the object and sets its attributes. It is equivalent to a constructor in other programming languages.

10. Question: Describe the purpose and use of properties in Python classes. How can you use the `@property` decorator and setters to control attribute access?

Answer: Properties in Python allow controlling the access and modification of class attributes. They are defined using the `@property` decorator to define the getter method and the `@<attribute>.setter` decorator to define the setter method. This allows defining custom behavior when getting and setting attribute values. For example:

```python
class Circle:
```

```
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius
    @radius.setter
    def radius(self, value):
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius cannot be negative.")
```

Scenario-Based Questions:

1. Question: Imagine you are developing a banking application in Python. How would you design a class hierarchy to represent different types of accounts, such as savings, checking, and fixed deposit accounts? What methods and attributes would you include in each class to handle transactions and account details?

2. You are building a game using OOP principles in Python. The game involves different types of characters, including warriors, mages, and archers. Each character has unique abilities and stats. How would you design a class hierarchy to represent these characters and their abilities? Provide an example of creating instances of different character types and using their abilities in the game.

3. Consider a scenario where you are developing a library management system. How would you design classes to represent books, library members, and borrowing records? How can you implement methods to handle book checkout and return processes while maintaining data integrity and ensuring no book is overbooked by multiple members?