Object-Oriented Programming (OOP)

Interview Questions

(Practice Project)

PW SKILLS

1. What are the four fundamental principles of Object-Oriented Programming (OOP)?

Answer:

The four fundamental principles of Object-Oriented Programming (OOP) are:

a) Encapsulation: This principle involves bundling data and methods that operate on that data within a single unit or object. It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of the methods and data.

b) Abstraction: Abstraction means hiding complex implementation details and showing only the necessary features of an object. It allows you to focus on what the object does instead of how it does it.

c) Inheritance: This principle allows a class to inherit properties and methods from another class. It promotes code reusability and establishes a relationship between parent and child classes.

d) Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. It provides a way to use a class exactly like its parent so that there's no confusion with mixing types, but each child class keeps its own methods as they are.

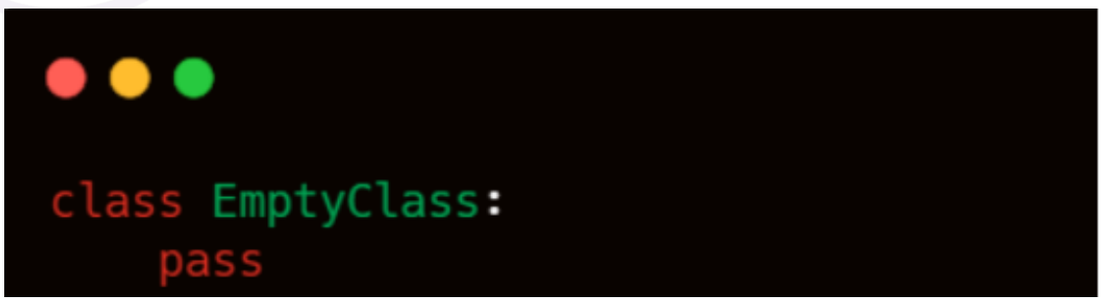
2. What is the difference between a class and an object in Python?How can we create an empty class in Python?

Answer:

A class is a blueprint or template for creating objects. It defines a set of attributes and methods that the objects of that class will have. In essence, a class is a user-defined data type.

An object, on the other hand, is an instance of a class. It's a concrete entity based on the class, and has actual values instead of variables. Each object contains data and code to manipulate the data.

To create an empty class in Python, you can use the pass statement:



3. Explain the concept of encapsulation in Python. How is it implemented?

Answer:

Encapsulation is the bundling of data and the methods that operate on that data within a single unit (class). It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of the methods and data.

In Python, encapsulation is implemented using private and protected members:

- Private members are denoted by prefixing the name with double underscores (__). They can't be accessed directly from outside the class.
- Protected members are denoted by prefixing the name with a single underscore (_). They shouldn't be accessed directly from outside the class, but can be.

```python
class EncapsulationExample:
    def __init__(self):
        self.__private_var = 10
        self._protected_var = 20

    def __private_method(self):
        print("This is a private method")

    def public_method(self):
        print("This is a public method")
        self.__private_method()

e = EncapsulationExample()
e.public_method()  # Works fine
# e.__private_method()  # This would raise an AttributeError
# print(e.__private_var)  # This would raise an AttributeError
print(e._protected_var)  # This works, but it's discouraged
```

4. What is inheritance, and why is it important in Object-Oriented Programming?

Answer:

Inheritance is a mechanism where a new class is derived from an existing class. The new class (derived or child class) inherits attributes and methods from the existing class (base or parent class).

Inheritance is important in OOP for several reasons:

- It promotes code reusability
- It allows for method overriding, enabling polymorphism
- It helps in organizing and structuring code by creating a hierarchy of classes

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

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

5. How does method overriding work in Python? Provide an example.

Answer:

Method overriding occurs when a derived class has a method with the same name as a method in its base class. The method in the derived class is said to override the method in the base class.

```python
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        print("Hello from Child")

p = Parent()
c = Child()

p.greet()  # Outputs: Hello from Parent
c.greet()  # Outputs: Hello from Child
```

6. What is the difference between public, protected, and private access modifiers in Python?

Answer:

Python doesn't have strict access modifiers like some other languages, but it does have conventions:

a) Public: By default, all members in a Python class are public. They can be accessed from outside the class.

Example: self.public_var = 10

b) Protected: Members prefixed with a single underscore are considered protected. They shouldn't be accessed directly from outside the class, but can be. This is just a convention. Example: self._protected_var = 20

c) Private: Members prefixed with double underscores are considered private. Python mangles these names to make them harder (but not impossible) to access from outside the class. Example: self.__private_var = 30

7. What is polymorphism in Python, and why is it useful?

Answer:

Polymorphism is the ability of different classes to be treated as instances of the same class through inheritance. It allows you to use a single interface to represent different underlying forms (data types or classes).

In Python, polymorphism is implemented through method overriding and duck typing.
Polymorphism is useful because:

- It allows for more flexible and reusable code
- It simplifies code by allowing the same interface to be used for different underlying processes
- It enables you to write more generic code that can work with objects of multiple types

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

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

8. What is the purpose of the `super()` function in Python?

Answer:
The super() function in Python is used to call methods of the superclass (parent class) in the derived class. It provides a way to extend the functionality of inherited methods.

Key purposes of super():
- It allows you to call methods of the superclass in the derived class
- It's useful in method overriding when you want to extend the functionality of the parent method
- It helps in maintaining the method resolution order in cases of multiple inheritance

```python
class Parent:
    def __init__(self):
        print("Parent's __init__")

class Child(Parent):
    def __init__(self):
        super().__init__()  # Calls Parent's __init__
        print("Child's __init__")

child = Child()
# Output:
# Parent's __init__
```

9. What is the difference between composition and inheritance in object-oriented programming?

Answer:

Inheritance is a mechanism where a new class is derived from an existing class, inheriting its attributes and methods. Composition is a design principle where a class is composed of one or more objects of other classes.

Key differences:
- Inheritance represents an "is-a" relationship, while composition represents a "has-a" relationship
- Inheritance can lead to tight coupling, while composition allows for more flexibility and loose coupling
- Inheritance can result in a complex hierarchy, while composition keeps classes independent

```python
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()

car = Car()
print(car.start())  # Output: Engine started
```

10. What is a constructor in Python, and why is it important in object-oriented programming?

Answer:

A constructor in Python is a special method named __init__ that is automatically called when an object of a class is created. It's used to initialize the attributes of the class.

Importance of constructors:

- They ensure that an object is properly initialized when it's created
- They allow you to set initial values for object attributes
- They can perform any setup the object needs before it's used

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Alice", 30)
print(person.name, person.age)  # Output: Alice 30
```

11. What is method chaining in Python, and why is it useful?

Answer:
Method chaining is a programming technique where multiple methods are called in a single line by returning self from each method. This allows for a more concise and readable code.

Benefits of method chaining:

- It allows for more compact code
- It can make code more readable by clearly showing a sequence of operations
- It's particularly useful for builder patterns or fluent interfaces.

12. How can you achieve method overloading in Python? Provide an example.

Answer:

Python doesn't support method overloading in the traditional sense, but we can achieve similar functionality using:

a) Default arguments:

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

calc = Calculator()
print(calc.add(5))         # Output: 5
print(calc.add(5, 10))     # Output: 15
print(calc.add(5, 10, 15))  # Output: 30
```

b) Variable-length arguments:

```python
class Calculator:
    def add(self, *args):
        return sum(args)

calc = Calculator()
print(calc.add(5))          # Output: 5
print(calc.add(5, 10))      # Output: 15
print(calc.add(5, 10, 15))  # Output: 30
```

13. What are the advantages of using private attributes in Python classes?

Answer:

- **Encapsulation:** Private attributes help in hiding the internal details of a class.
- **Data Protection:** They prevent accidental modification of important data.
- **Naming Conflicts:** They avoid naming conflicts in inherited classes.
- **Control:** They provide control over how the data is accessed or modified.

14. Explain the concept of encapsulation in Python. How is it implemented?

Answer:

Encapsulation is the bundling of data and methods that operate on that data within a single unit (class). It's implemented in Python using:

- **Private attributes/methods:** Prefixed with double underscores (__).
- **Protected attributes/methods:** Prefixed with a single underscore (_).

Example:

```python
class Employee:
    def __init__(self, name, salary):
        self.__name = name    # Private attribute
        self._salary = salary  # Protected attribute

    def get_name(self):
        return self.__name

    def _get_salary(self):  # Protected method
        return self._salary

emp = Employee("Alice", 50000)
print(emp.get_name())  # Output: Alice
# print(emp.__name)  # This would raise an AttributeError
print(emp._get_salary())  # Output: 50000, but it's discouraged
```

15. Is multiple inheritance supported in Python? If yes then explain with an example.

Answer:
Python supports multiple inheritance. A class can inherit from multiple parent classes.

Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

class Flyable:
    def fly(self):
        return f"{self.name} is flying"

class Bird(Animal, Flyable):
    def chirp(self):
        return f"{self.name} is chirping"

sparrow = Bird("Sparrow")
print(sparrow.fly())    # Output: Sparrow is flying
print(sparrow.chirp())  # Output: Sparrow is chirping
```

16. How does encapsulation help in reducing complexity and increasing reusability?

Answer:

- Reduces complexity by hiding implementation details and exposing only necessary interfaces.
- Increases reusability by creating self-contained units that can be easily used in different contexts.
- Improves maintainability as changes to internal implementation don't affect external code.
- Enhances data integrity by controlling access to object's internal state.

17.What is the significance of the 'self' keyword & __init()__ methods in Python classes?

Answer:

- **'self':** Refers to the instance of the class. It's used to access variables and methods of the class within the class itself.
- **__init__():** It's the constructor method called when an object is created. It initializes the object's attributes.

Example:

```python
class Person:
    def __init__(self, name):
        self.name = name  # Using self to set an instance
variable

    def greet(self):
        print(f"Hello, my name is {self.name}")  # Using self
to access an instance variable

person = Person("Alice")
person.greet()  # Output: Hello, my name is Alice
```

18.What is the difference between __str__ and __repr__ methods?

Answer:

- **__str__:** Returns a string representation of an object meant for end-users. It's called by the str() function and used in print().
- **__repr__:** Returns a string representation of an object meant for developers. It's called by the repr() function and used in the interactive console.

Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

p = Point(3, 4)
print(str(p))   # Output: Point(3, 4)
print(repr(p))  # Output: Point(x=3, y=4)
```

19.Explain the concept of method overloading and method overriding.How are these concepts implemented in different programming languages?

Answer:

Method Overloading: Having multiple methods with the same name but different parameters in the same class.
Method Overriding: Redefining a method in a derived class that is already defined in the base class.

Implementation in different languages:

- **Python:** Doesn't support true method overloading, uses default arguments or *args. Supports method overriding.
- **Java:** Supports both method overloading and overriding.
- **C++:** Supports both method overloading and overriding.

20.Design a simple inventory management system with Python classes. Create a base class Product with attributes for product name, price, and quantity. Derive two classes: Electronics (with an additional warranty attribute) and Clothing (with an additional size attribute). Implement methods to add stock, apply a discount, and display product details.

Solution:

```python
class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def add_stock(self, amount):
        self.quantity += amount

    def apply_discount(self, percentage):
        self.price *= (1 - percentage / 100)

    def display_details(self):
        return f"Name: {self.name}, Price: ${self.price:.2f}, Quantity: {self.quantity}"

class Electronics(Product):
    def __init__(self, name, price, quantity, warranty):
        super().__init__(name, price, quantity)
        self.warranty = warranty

    def display_details(self):
        return f"{super().display_details()}, Warranty: {self.warranty} months"

class Clothing(Product):
    def __init__(self, name, price, quantity, size):
        super().__init__(name, price, quantity)
        self.size = size

    def display_details(self):
        return f"{super().display_details()}, Size: {self.size}"

# Usage
laptop = Electronics("Laptop", 1000, 5, 24)
shirt = Clothing("T-Shirt", 20, 100, "M")

laptop.add_stock(3)
laptop.apply_discount(10)
print(laptop.display_details())

shirt.add_stock(50)
shirt.apply_discount(5)
print(shirt.display_details())
```