Object-Oriented Programming in Python

1. Classes and Objects

- **Class**: A blueprint for creating objects. It defines attributes (data) and methods (functions) that the objects created from the class will have.
- **Object**: An instance of a class. When you create an object, you bring the blueprint (class) to life.

python

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        return f"{self.name} says woof!"

# Create an object (instance) of the Dog class
my_dog = Dog("Buddy", 5)

# Access instance attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Buddy says woof!
```

- **`__init__` Method**: The initializer (constructor) method that runs as soon as an object of a class is instantiated. It initializes the object's attributes.
- **Self**: The first parameter of any method in a class. It refers to the instance calling the method.

2. Inheritance

- **Inheritance** allows a class (derived class) to inherit attributes and methods from another class (base class). This promotes code reusability.

python

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

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return f"{self.name} says woof!"

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return f"{self.name} says meow!"

dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())  # Output: Buddy says woof!
print(cat.speak())  # Output: Whiskers says meow!
```

- **Base Class (Parent Class)**: The class being inherited from.
- **Derived Class (Child Class)**: The class that inherits from the base class.
- **Overriding Methods**: A derived class can override methods of the base class to provide specific behavior.

3. Encapsulation

- **Encapsulation** refers to the bundling of data (attributes) and methods that operate on the data into a single unit (class). It also involves restricting direct access to some of an object's components, which is done by making attributes private using underscores (`_` or `__`).

python

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance

account = Account("Alice", 1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

- **Private Attributes/Methods**: By convention, attributes or methods that start with an underscore (e.g., `_balance`) are considered private and should not be accessed directly outside the class.
- **Getter and Setter Methods**: These are used to access and modify private attributes, maintaining control over how attributes are accessed or modified.

4. Polymorphism

- **Polymorphism** allows methods in different classes to have the same name but behave differently based on the object that calls them. This is particularly useful when dealing with inheritance.

python

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
```

```
    def speak(self):
        return "Woof!"

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

# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())  # Output: Woof! Meow!
```

- **Method Overriding**: A derived class provides a specific implementation of a method that is already defined in its base class.
- **Polymorphic Functions**: Functions that can take objects of different classes and treat them uniformly.

5. Additional Concepts

Method Resolution Order (MRO): Python uses MRO to determine the order in which classes are inherited when dealing with multiple inheritance. The `super()` function can be used to call methods from a parent class in a child class.
python

```
class A:
    def method(self):
        print("A method")

class B(A):
    def method(self):
        print("B method")
        super().method()

obj = B()
obj.method()  # Output: B method
              #         A method
```

-

- **Abstraction**: This concept involves hiding the complex implementation details and showing only the essential features of an object. It's typically achieved using abstract classes and interfaces (though Python does not have interfaces, abstract classes can be created using the `abc` module).

Summary

- **Classes and Objects**: Define classes as blueprints for creating objects with attributes and methods.
- **Inheritance**: Enables classes to inherit properties and methods from other classes, promoting code reuse.
- **Encapsulation**: Restricts access to certain components, bundling data and methods inside classes to protect data integrity.
- **Polymorphism**: Allows the same method to have different behaviors based on the object that calls it, enhancing flexibility and integration.