1. **Multiple inheritance** allows a class to inherit attributes and methods from more than one parent class. This means a single subclass can have multiple superclasses , and it can inherit the characteristics of all its parent classes.
   To implement multiple inheritance, you simply list the parent classes in the class definition, separated by commas. When an attribute or method is accessed on an instance of the subclass, Python searches for that attribute or method in the subclass first, then in the first parent class, then in the second parent class, and so on, following the order in which the parent classes are listed.

   ```
   class Animal:
      def sound(self):
         return "Generic animal sound"

   class Bird:
      def sound(self):
         return "Chirp chirp!"

   class Parrot(Animal, Bird):
      pass

   parrot = Parrot()
   print(parrot.sound())  # Output: Chirp chirp!
   ```

   In this example, we have three classes: **Animal, Bird, and Parrot**. The Parrot class inherits from both Animal and Bird, making it a subclass with multiple inheritance.
   When we create an instance of **Parrot (parrot = Parrot())**, the **sound()** method is called on the parrot object. Since **Parrot inherits from Bird,** Python finds the **sound()** method in the Bird class first, and that's why it returns **"Chirp chirp!".**
   If the **sound()** method were not found in the Bird class, Python would then search for it in the Animal class, and if found, it would have returned **"Generic animal sound"**

   It's essential to be aware of the **method resolution order (MRO**) in cases of multiple inheritance. Python uses the **C3 Linearization** algorithm to determine the MRO, which determines the order in which the parent classes are searched when resolving method calls. **The MRO is influenced by the order in which the parent classes are listed in the subclass definition.**

2. **Delegation** is a design pattern that allows one **object (the delegator)** to pass certain operations or tasks to **another object (the delegate**) without inheriting its behavior. This pattern is based on the principle of composition over inheritance, which encourages code reusability and modularity by combining multiple objects to achieve complex functionalities.

   In Python, delegation is typically implemented by creating a class that contains an instance of another class and then forwarding specific method calls or attribute accesses to the delegate object.

```
class Engine:
    def start(self):
        print("Engine started.")

    def stop(self):
        print("Engine stopped.")

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

    def start(self):
        print("Car is starting...")
        self.engine.start()

    def stop(self):
        print("Car is stopping...")
        self.engine.stop()

car = Car()
car.start()  # Output: Car is starting... Engine started.
car.stop()   # Output: Car is stopping... Engine stopped.
```

In this example, we have two classes: '**Engine' and 'Car'**. The Engine class represents an engine with '**start()'** and '**stop()'** methods. The Car class contains an instance of the Engine class, making it the delegate.
When we create a Car instance **(car = Car()),** it has an associated Engine instance as **self.engine**, achieved through composition. The Car class implements its own **start()** and **stop()** methods, but within these methods, it delegates the actual engine-related functionalities to the Engine object using '**self.engine.start()'** and **'self.engine.stop()'.**
This way, the Car class can reuse the capabilities of the Engine class without inheriting its behavior. The Engine class and its functionalities can evolve independently without affecting the Car class, promoting modularity and ease of maintenance.

3. **Composition** is a design principle that involves creating complex objects by combining multiple simpler objects, also known as **components**. Composition allows you to build complex functionalities by assembling objects in a way that promotes code reusability, flexibility, and modularity.
The key idea behind composition is to create a **"has-a" relationship between classes**. Instead of inheriting behavior from a superclass (as in inheritance), a class contains instances of other classes, known as its components. The composed class delegates certain operations to its components to achieve its functionalities.

```
class Engine:
    def start(self):
        print("Engine started.")

    def stop(self):
        print("Engine stopped.")

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

    def start(self):
        print("Car is starting...")
        self.engine.start()

    def stop(self):
        print("Car is stopping...")
        self.engine.stop()

car = Car()
car.start()  # Output: Car is starting... Engine started.
car.stop()   # Output: Car is stopping... Engine stopped.
```

In this example, we have two classes: ‘**Engine’** and ‘**Car’**. The Engine class represents an engine with ‘**start()’** and ‘**stop()’** methods. The Car class contains an instance of the Engine class, making it a composition.
When we create a Car instance ‘**(car = Car())’,** it has an associated Engine instance as ‘**self.engine’**, achieved through composition. The Car class implements its own ‘**start()’** and ‘**stop()’** methods, but within these methods, it delegates the actual engine-related functionalities to the Engine object using ‘**self.engine.start()’** and ‘**self.engine.stop()’.**
By composing the Car class with an Engine object, we create a "has-a" relationship, meaning a Car has an Engine. This is in contrast to inheritance, where a subclass "is-a" superclass.

4.

Bound method is a method that is associated with a specific instance of a class. When a class method is called on an instance, it becomes a bound method, and the instance is automatically passed as the first argument (conventionally named self). **The bound method can then access and operate on the attributes and behavior of that specific instance.**
Bound methods are essential in object-oriented programming as they allow instances to interact with their own data and behavior in a structured and object-oriented manner.

To define a method in a class, you need to include the self parameter as the first argument in the method's signature. This parameter represents the instance that the method will operate on when called on that instance.

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

  def bark(self):
    print(f"{self.name} says woof!")

  def birthday(self):
    self.age += 1
    print(f"{self.name} is now {self.age} years old.")

# Creating instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 2)

# Using bound methods on instances
dog1.bark()    # Output: Buddy says woof!
dog2.bark()    # Output: Max says woof!

dog1.birthday()  # Output: Buddy is now 4 years old.
dog2.birthday()  # Output: Max is now 3 years old.
```

In this example, we have a Dog class with two bound methods: '**bark()**' and '**birthday()'.** When these methods are called on instances of the Dog class '**(e.g., dog1.bark())',** the self parameter is automatically passed, representing the instance '**(dog1 in this case)'.** The bound method can access and modify the attributes of that specific instance, like '**self.name'** and '**self.age'**.

5.

In Python, **pseudoprivate attributes**, also known as **"name mangling,"** are attributes that have a double underscore prefix **('__')** but do not end with more than one trailing underscore. For example, attributes with names like '**__example'** or '**__data**'are pseudoprivate
In Python, there are no strict access control keywords like "private" or "protected" as seen in some other programming languages (e.g., Java). Instead, Python uses a convention to indicate that an attribute or method should be treated as "**private"** within the class. This convention involves adding a double underscore prefix **('__'**) to the attribute or method name

```
class MyClass:
  def __init__(self):
    self.__private_attribute = 42

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

  def public_method(self):
        return self._private_attribute + 10
```

In the above example, ‘**__private_attribute’** and ‘**__private_method** ‘ are pseudoprivate attributes. Their names are "mangled" by the interpreter by adding the class name as a prefix to avoid accidental name clashes. This means that if you have a subclass that defines an attribute or method with the same name, the interpreter will modify the name of the attribute or method to avoid conflicts.
For instance, if you create a subclass of ‘**MyClass**’ and try to define an attribute named ‘**__private_attribute’**, it will be name-mangled to _MyClass__private_attribute:

```
class MySubclass(MyClass):
    def __init__(self):
        self.__private_attribute = 100  # Name-mangled to _MySubclass__private_attribute

    def public_method(self):
        return self.__private_attribute + 50
```