Q1: Abstract Base Classes and Multiple Inheritance: Use an abstract base class with multiple child classes inheriting specific functionalities and overriding methods.

```
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 * self.radius

class Rectangle(Shape):     def
__init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Area of Circle: {circle.area()}") print(f"Area
of Rectangle: {rectangle.area()}")
```

Q2: Overriding Methods in Multiple Inheritance: Implement methods in child classes that inherit from multiple parent classes and explore different override scenarios.

```
class Vehicle:    def
start_engine(self):
    print("Vehicle engine started.")

class ElectricVehicle:
def start_engine(self):
    print("Electric vehicle engine started.")

class HybridCar(Vehicle, ElectricVehicle):
def start_engine(self):

    super(Vehicle, self).start_engine()
    print("Hybrid car engine started in a special way.")

hybrid_car = HybridCar()

hybrid_car.start_engine()
```

Q3: Metaclass for Class Decoration: Design a metaclass that acts as a decorator, injecting additional logic or modifying the class before its creation.

```
class AttributePrefixMeta(type):
def __new__(cls, name, bases, dct):
     prefixed_dict = {f"prefixed_{key}": value for key, value in dct.items()}


     return super().__new__(cls, name, bases, prefixed_dict)

class MyClass(metaclass=AttributePrefixMeta):
  value = 42
```

```
obj = MyClass() print(obj.prefixed_value)
# Output: 42
```