Module 10: Deep Dive into Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which encapsulate data and behavior. In Python, everything is an object, and understanding OOP is essential for writing clear, modular, and maintainable code. Let's delve into OOP concepts with examples:

Classes and Objects

- Classes are blueprints for creating objects. They define attributes (data) and methods (behavior).
- Objects are instances of classes.

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

 def drive(self):
 print(f"Driving {self.make} {self.model}")

my_car = Car("Toyota", "Camry")
my_car.drive()
```

Attributes and Methods

- Attributes are variables that store data within objects.
- Methods are functions that perform operations on objects.

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

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

my_circle = Circle(5)
print("Area:", my_circle.area())
```

Encapsulation

- Encapsulation restricts access to certain components of objects to prevent direct modification from outside.
- Use private attributes and methods to encapsulate implementation details.

```python
class BankAccount:
 def __init__(self, balance):
 self.__balance = balance

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

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

my_account = BankAccount(1000)
my_account.deposit(500)
my_account.withdraw(200)
```

Inheritance

- Inheritance allows a class to inherit attributes and methods from another class.
- The subclass can override methods or add new ones.

```python
class Animal:
 def speak(self):
 print("Animal speaks")

class Dog(Animal):
 def speak(self):
 print("Dog barks")

my_dog = Dog()
my_dog.speak()
```

Polymorphism

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.

- It enables flexibility and code reuse.

```
class Shape:
 def area(self):
 pass

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

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

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

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

def calculate_area(shape):
 return shape.area()

my_circle = Circle(5)
my_rectangle = Rectangle(4, 6)
print("Circle area:", calculate_area(my_circle))
print("Rectangle area:", calculate_area(my_rectangle))
```

Abstraction

- Abstraction hides complex implementation details and only exposes necessary functionalities.
- It simplifies code and improves maintainability.

```
python
Copy code
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 *
```

Composition and Aggregation

- Composition represents a "has-a" relationship, where one class contains another as a component.
- Aggregation represents a "part-of" relationship, where one class uses another class but can exist independently.

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

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

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

my_car = Car()
my_car.start()
```

Understanding these OOP concepts enables you to design robust and scalable applications in Python, promoting code reuse, modularity, and maintainability.

Conclusion

In conclusion, understanding Object-Oriented Programming (OOP) concepts in Python is essential for writing clean, modular, and maintainable code. By leveraging classes, objects, inheritance, encapsulation, polymorphism, abstraction, and composition, developers can design efficient and flexible solutions for a wide range of problems. OOP promotes code reusability, scalability, and readability, making it a fundamental paradigm for software development in Python and beyond.

Exercises and Answers for Deep Dive into Object-Oriented Programming (OOP)

Class Creation:

Create a Python class named `Rectangle` with attributes `length` and `width`, and a method `calculate_area()` that returns the area of the rectangle.

Inheritance:

Create a subclass `Square` of the `Rectangle` class. Override the `calculate_area()` method to calculate the area of a square given its side length.

Encapsulation:

Modify the `Rectangle` class to make the `length` and `width` attributes private. Provide methods `set_length()` and `set_width()` to set the length and width of the rectangle, and methods `get_length()` and `get_width()` to retrieve their values.

Polymorphism:

Create a function `print_area()` that accepts an object of either `Rectangle` or `Square` class and prints the area using the `calculate_area()` method.

Abstraction:

Create an abstract class `Shape` with an abstract method `calculate_area()`. Modify the `Rectangle` and `Square` classes to inherit from `Shape` and implement the `calculate_area()` method.

Answers:

Class Creation:

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

 def calculate_area(self):
 return self.length * self.width
```

```
rect = Rectangle(5, 4)
print("Area of rectangle:", rect.calculate_area())
```

Inheritance:

```
class Square(Rectangle):
 def __init__(self, side_length):
 super().__init__(side_length, side_length)

 def calculate_area(self):
 return self.length * self.length

square = Square(5)
print("Area of square:", square.calculate_area())
```

Encapsulation:

```
class Rectangle:
 def __init__(self, length, width):
 self.__length = length
 self.__width = width

 def set_length(self, length):
 self.__length = length

 def set_width(self, width):
 self.__width = width

 def get_length(self):
 return self.__length

 def get_width(self):
 return self.__width

rect = Rectangle(5, 4)
rect.set_length(6)
print("Length of rectangle:", rect.get_length())
```

Polymorphism:

```
def print_area(shape):
 print("Area:", shape.calculate_area())
```

```python
rect = Rectangle(5, 4)
square = Square(5)

print_area(rect)
print_area(square)
```

Abstraction:

```python
from abc import ABC, abstractmethod

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

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

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

class Square(Shape):
 def __init__(self, side_length):
 self.side_length = side_length

 def calculate_area(self):
 return self.side_length * self.side_length

rect = Rectangle(5, 4)
square = Square(5)

print("Area of rectangle:", rect.calculate_area())
print("Area of square:", square.calculate_area())
```

These exercises cover fundamental OOP concepts like class creation, inheritance, encapsulation, polymorphism, and abstraction, allowing you to practice and reinforce your understanding of OOP in Python.