Chapter 4: Object-Oriented Programming

Introduction to object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects". Objects are data structures that contain both data and functions, and they are used to represent real-world entities or concepts in a program.

OOP is designed to help developers write reusable, modular, and maintainable code. It allows developers to organize their code into classes and objects, which makes it easier to understand and maintain.

Classes and objects

In object-oriented programming (OOP), a class is a template that defines the data and functions that an object will contain. Objects are then created from these classes, and are called instances of the class.

Here is an example of a simple class:

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

    def greet(self):
        print(f'Hello, my name is {self.name} and I am {self.age} years old.')
```

This class defines a **Person** object with a **name** and **age** attribute, and a **greet** function that prints a greeting message.

To create an instance of this class, we can use the **__init__** method, which is a special method that is called when an object is created:

```
person = Person('John', 30)
person.greet()
# Output: Hello, my name is John and I am 30 years old.
```

In this example, **person** is an instance of the **Person** class. It contains the data and functions defined in the class, and can be modified at runtime.

Objects can interact with each other through their functions. For example, we can create another object and have it call the **greet** function of the **person** object:

```
class Dog:
    def bark(self, person):
        person.greet()
        print('Woof woof!')
```

```
dog = Dog()
dog.bark(person)
# Output: Hello, my name is John and I am 30 years old.
#         Woof woof!
```

Inheritance and polymorphism

Inheritance

Inheritance is the ability of a class to inherit the attributes and methods of another class. This allows developers to create a new class that is a modified version of an existing class, without having to rewrite all of the code.

For example, consider the following classes:

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

    def make_sound(self):
        print('Some generic animal sound')

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name, species='Dog')

    def make_sound(self):
        print('Woof woof!')
```

In this example, the **Dog** class is a subclass of the **Animal** class. It inherits the **name** and **species** attributes and the **make_sound** method from the **Animal** class, and defines its own version of the **make_sound** method.

To create an instance of the **Dog** class, we can use the **__init__** method as follows:

```
dog = Dog('Fido')
dog.make_sound()
# Output: Woof woof!
```

Polymorphism

Polymorphism is the ability of a class to take on multiple forms. This can be achieved through inheritance, where a subclass can override or extend the methods of its superclass.

For example, consider the following code:

```
class Animal:
    def __init__(self, name, species):
```

```
        self.name = name
        self.species = species

    def make_sound(self):
        print('Some generic animal sound')

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species='Dog')
        self.breed = breed

    def make_sound(self):
        print('Woof woof!')

class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species='Cat')
        self.breed = breed

    def make_sound(self):
        print('Meow meow!')

def make_sounds(animals):
    for animal in animals:
        animal.make_sound()

dog = Dog('Fido', 'Labrador')
cat = Cat('Fluffy', 'Siamese')

make_sounds([dog, cat])
# Output: Woof woof!
#         Meow meow!
```

In this example, we have defined an **Animal** class with a **name** and **species** attribute, and a **make_sound** function that prints a generic animal sound. We have then defined a **Dog** class that inherits from the **Animal** class, and has its own **__init__** method and **make_sound** function. The **Dog** class has a **breed** attribute and overrides the **make_sound** function to print a specific dog sound.

The **dog** object is an instance of the **Dog** class, which is a subclass of the **Animal** class. The **dog** object has the ability to take on multiple forms by inheriting the data and functions of the **Animal** class and by overriding the **make_sound** function with its own implementation.

Similarly, when the **cat** object is an instance of the **Cat** class, which is a subclass of the**Animal** class. The **cat** object has the ability to take on multiple forms by inheriting the data and functions of the **Animal** class and by overriding the

make_sound function with its own implementation.

When the **make_sound** function is called on the **cat** object, it calls the **make_sound** function of the **Cat** class, which prints the specific cat sound "Meow meow!".

Encapsulation and data hiding

Encapsulation is a key concept in object-oriented programming (OOP) that refers to the idea of bundling data and methods that operate on that data within a single unit, or object. Encapsulation helps to protect the data from outside access and modification, and it can also be used to implement data hiding.

You can use class definitions to implement encapsulation. For example:

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

    def get_balance(self):
        return self.__balance

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

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print('Insufficient funds')
```

Data hiding is a technique in object-oriented programming (OOP) that refers to the idea of keeping certain data and implementation details private within a class, so that they cannot be accessed or modified from outside the class. Data hiding helps to protect the integrity of the data and to prevent unintended modifications, and it is often used in conjunction with encapsulation to protect data and implementation details within an object.

In this example, we have defined a **BankAccount** class that has a **name** and **balance** attribute, as well as methods for depositing, withdrawing, and checking the balance. We have prefixed the **name** and **balance** attributes with **__**, which is a convention to indicate that these attributes should not be accessed or modified directly from outside the class.

To access or modify the **name** and **balance** attributes, we have defined methods such as **get_balance** and **deposit** that allow us to safely manipulate the data. This is an example of encapsulation and data hiding in action.