Object-oriented Programming In Python

Michael H. Goldwasser ; David Letscher

Python from Scratch - Lesson 4: Object Oriented Programming. In the previous chapters, we intentionally avoided object oriented programming (OOP). We think it's easier and more fun to start learning Python without having to think about OOP. However, once you have mastered the basics of Python, it's time to move on to more advanced topics. In this chapter, we'll introduce you to object-oriented programming (OOP) in Python.

OOP is a programming paradigm that allows you to structure your code in a more organized way. It's based on the idea of classes and objects, which are used to create reusable code.

In this chapter, we'll cover:

- The basics of classes and objects in Python
- Inheritance and polymorphism
- Encapsulation and abstraction
- Exception handling

By the end of this chapter, you'll have a solid understanding of OOP in Python, and you'll be able to write more efficient and maintainable code.

Let's get started!

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that allows you to structure your code in a more organized way. It's based on the idea of classes and objects, which are used to create reusable code.

A class is a blueprint for creating objects. It defines the properties and methods that an object of that class will have.

An object is an instance of a class. It contains the values of the properties defined in the class.

In Python, you can define a class using the class statement.

```
class ClassName:
  pass
```

For example, let's define a class called `Car` that represents a car.

```
class Car:
  pass
```

In this class, we can define properties and methods.

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

  def start(self):
    print(f'Starting {self.make} {self.model}')
```

We can then create an object of the `Car` class using the `new` operator.

```
car = Car('Toyota', 'Camry')
car.start()
```

This will output:

```
Starting Toyota Camry
```

In this example, we've defined a `__init__` method, which is called when we create a new object. We've also defined a `start` method, which prints a message.

Inheritance

Inheritance is a way to define a class that is based on another class. This is done using the `extends` keyword.

```
class Car:
  pass

class Sedan(Car):
  pass
```

In this example, we've defined a new class called `Sedan` that extends the `Car` class.

Polymorphism

Polymorphism is the ability of an object to take on many forms. This is done using the `super` method.

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

  def start(self):
    print(f'Starting {self.make} {self.model}')

class Sedan(Car):
  def start(self):
    print(f'Starting {self.make} {self.model} (Sedan)')
```

In this example, we've defined a new method called `start` in the `Sedan` class.

```
car = Car('Toyota', 'Camry')
car.start()  # Outputs: Starting Toyota Camry
sedan = Sedan('Toyota', 'Camry')
sedan.start()  # Outputs: Starting Toyota Camry (Sedan)
```

Encapsulation

Encapsulation is the practice of hiding the internal details of a class from the outside world. This is done using the `@property` decorator.

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

  @property
  def make(self):
    return self._make

  @make.setter
  def make(self, make):
    self._make = make

  def start(self):
    print(f'Starting {self.make} {self.model}')
```

In this example, we've defined a private property called `_make`.

```
car = Car('Toyota', 'Camry')
car.make = 'Honda'
car.start()  # Outputs: Starting Honda Camry
```

Exception Handling

Exception handling is a way to handle errors that occur during the execution of a program. This is done using the `try`/`except` statement.

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

  @property
  def make(self):
    return self._make

  @make.setter
  def make(self, make):
    self._make = make

  def start(self):
    try:
      print(f'Starting {self.make} {self.model}')
    except Exception as e:
      print(f'Error: {e}')
```

In this example, we've defined a `start` method that raises a `ValueError` if the make is not a string.

```
car = Car('Toyota', 'Camry')
car.start()  # Outputs: Starting Toyota Camry
car = Car('Toyota', '123')
car.start()  # Outputs: Error: '123' is not a valid make
```

In this chapter, we've covered the basics of object-oriented programming in Python. We've looked at classes and objects, inheritance, polymorphism, encapsulation, and exception handling.

We hope that you found this chapter helpful. If you have any questions or comments, please don't hesitate to reach out to us.

Thank you for learning Python with us!