Chapter 7: Python Object-Oriented Programming

Contents

Red light: No AI

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself

Code Shape

Please group program code as follows.

- Get input
- Calculate
- Display

Python is Object-Oriented (OOP)

Until now, we have designed our program around functions: blocks of statements which manipulate data. This is called procedural-oriented programming.

Another way of organizing your program is to combine data and functionality and wrap it inside something called an object. This is called object-oriented programming.

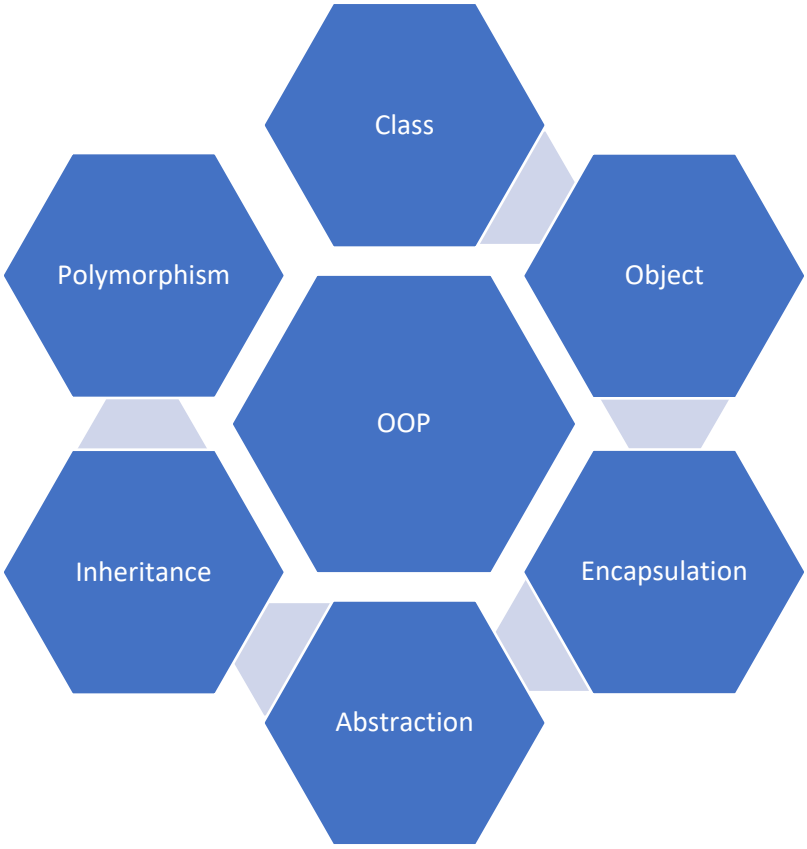
Python is an object-oriented programming language. Everything in Python is an object. We have been using many object-oriented concepts already. Object-oriented programming (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.

The key notion is that of an object. An object consists of two things: data (fields) and functions (called methods) that work with that data.

Strings in Python are objects. The data of the string object is the actual characters that make up that string. The methods are things like lower, replace, and split. To Python, everything is an object. That includes not only strings and lists, but also integers, floats, and even functions themselves.

OOP Mind Map

The following mind map gives a high level view of the different concepts that are part of Object Oriented programming.



1. **Class**: A blueprint for creating objects, defining properties and behaviors (attributes and methods).
2. **Object**: An instance of a class, representing a specific entity with defined data and behavior.
3. **Encapsulation**: Restricting direct access to an object's data, bundling data with methods for controlled interaction.
4. **Abstraction**: Hiding complex implementation details, showing only essential features to the user.

5. **Inheritance**: Enabling a class (child) to acquire properties and behaviors of another class (parent).
6. **Polymorphism**: Allowing entities to take on multiple forms, enabling methods to perform differently based on context.

A Class Definition is a Blueprint for Objects

A class definition is a blueprint (set of plans) from which many individual objects can be constructed, created, produced, instantiated, or whatever verb you prefer to use for building something from a set of plans.

An object is a programming construct that encapsulates data and the ability to manipulate that data in a single software entity. The blueprint describes the data contained within and the behavior of objects instantiated according to the class definition.

An object's data is contained in variables defined within the class (often called member variables, instance variables, data members, attributes, fields, properties, etc.). The terminology used often depends on the background of the person writing the document.

An object's behavior is controlled by methods defined within the class. In Python, methods are functions that are defined within a class definition.

An object is said to have state and behavior. At any instant in time, the state of an object is determined by the values stored in its variables and its behavior is determined by its methods.

What is an Object-Oriented Program?

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an Object?

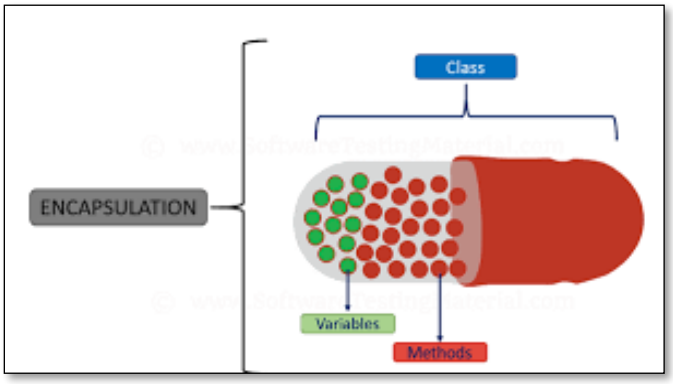
An object is a software construct that encapsulates data, along with the ability to use or modify that data.

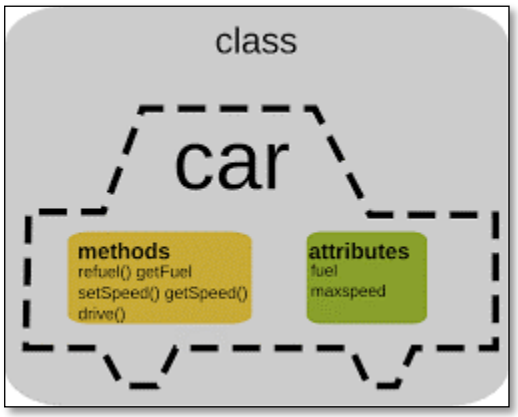
What is Encapsulation?

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

What is Data Hiding?

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.





Classes and Objects

One of the most important concepts in object-oriented programming is the distinction between classes and objects, which are defined as follows:

1. **Class** - A blueprint created by a programmer for an object. This defines a set of attributes that will characterize any object that is instantiated from this class.
2. **Object** - An instance of a class. A class is used to create an object in a program.

These are used to create patterns (in the case of classes) and then make use of the patterns (in the case of objects).

Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that **belong** to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the attributes of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called instance variables and class variables respectively.

A class is created using the **class** keyword. The fields and methods of the class are listed in an indented block.

The self Parameter

Class methods have only one specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list. This variable refers to the object itself, and by convention, it is given the name **self**.

The **self** parameter tells Python which object to operate on. **self** references the methods and class variables of that object.

In a sense, everything defined in an object belongs to it**self**.

Python Classes and Objects

At a basic level, an object is simply some code plus data structures that are smaller than a whole program. Defining a method allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain many methods as well as data that is manipulated by those methods. We call data items that are part of the object attributes.

We use the **class** keyword to define the data and code that will make up each of the objects. The **class** keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

Data Hiding: The _ (underscore) in front of the attribute name is to hide the data from anything outside the class or program. This would be considered private in other languages. In Python, this is not enforced as in other languages, but is only a convention. Only the class and its methods are supposed to access and change the data directly.

A class is like a cookie cutter. The objects created using the class are cookies. You don’t put frosting on the cookie cutter; you put frosting on the cookies. You can put different frosting on each cookie.

The following photo shows a class and two objects.

OOP Analysis

Classes = Nouns (Person, Truck, Animal, Account, Car, Client, Player, Enemy, etc.)

Attributes = Adjectives (Color, Speed, Lives, Name, Size, etc.)

Methods = Verbs (Get, Move, Display, Find, Calculate, etc.)

Create Class and Object

A simple class is shown in the following example.

```
# Define a class
class Person:
    pass # An empty code block

# Create a Person() object named p
p = Person()

# Printing an object shows the memory address
print(p)
```

Example run:

```
<__main__.Person object at 0x0000028AC1B186A0>
```

How It Works

We create a new class using the **class** statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the **pass** statement. This is handy when sketching out the methods of a class.

We create an object/instance of this class using the name of the class followed by a pair of parentheses.

When you print an object directly, you get the memory address where your object is stored. The address will have a different value on your computer as Python will store the object wherever it finds space.

Methods

A method is a function that "belongs to" an object. **self** is the parameter that ties the method to an object. Method names usually include **verbs** since they represent **actions**.

```
class Person:
    # Method with the self parameter
    def say_hi(self):
        print("Hello, how are you? ")
Paul = Person()
Paul.say_hi()
```

Example run:

```
Hello, how are you?
```

How It Works

Here we see the self in action. Notice that the **say_hi()** method takes no parameters but still has the **self** in the function definition. **self** refers to the object. The object owns the method.

Cookie Class

Here is an example using methods and attributes to create different types of cookies.

```
# Filename: cookie_class.py
class Cookie:

    def apply_frosting(self, frosting):
        self.frosting = frosting

    def cut_cookie(self, shape):
        self.shape = shape

    def display_cookie(self):
        print(f"Cookie shape: {self.shape} Frosting: {self.frosting}")

# Create cookie1 object
cookie1 = Cookie()
# Call object methods
cookie1.cut_cookie("Star")
cookie1.apply_frosting("Chocolate")
cookie1.display_cookie()

# Create cookie2 object
cookie2 = Cookie()
# Call object methods
cookie2.cut_cookie("Reindeer")
cookie2.apply_frosting("Vanilla")
cookie2.display_cookie()
```

Example run:

```
Cookie shape: Star Frosting: Chocolate
Cookie shape: Reindeer Frosting: Vanilla
```

Getters and Setters

Getters and Setters allow the program to access the data attributes on the class. Getters and setters allow control over the values. You may validate the given value in the setter before setting the value.

- **getter:** returns the value of the attribute.
- **setter**: takes a parameter and assigns it to the attribute.

```
class Person:

    def get_hair_color(self):
        return self._hair_color

    def get_name(self):
        return self._name

    def set_hair_color(self, hair_color):
        self._hair_color = hair_color

    def set_name(self, name):
        self._name = name

    def say_hi(self):
        print(f"Hello, how are you? My name is {self._name}.")

Paul = Person()
Paul.set_hair_color("brown")
Paul.set_name("Paul")
Paul.say_hi()
print(f"My hair is {Paul.get_hair_color()}.")
```

Example run:

```
Hello, how are you? My name is Paul.
My hair is brown.
```

Tutorial 7.1: Sammy the Shark

The following program demonstrates classes, objects, and methods.

An object is an instance of a class. We'll make a **Shark** object called **sammy**.

Create and name the program: **shark.py**

```python
"""
    Name: shark_1.py
    Author:
    Created:
    Purpose: Demonstrate class, objects, methods,
    getters and setters
"""


class Shark:
    """Define shark methods."""

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def swim(self):
        print("The shark is swimming.")

    def be_awesome(self):
        print(f"{self._name} is being awesome.")


# Create a shark object
sammy = Shark()

# Call shark methods
sammy.set_name("Sammy")
print(f"My name is {sammy.get_name()}")
sammy.swim()
sammy.be_awesome()
```

Example run:

```
My name is Sammy
The shark is swimming.
Sammy is being awesome.
```

How It Works

The **Shark** object **sammy** is using four methods:

- **get_name()** This returns the **self._name** attribute of the object.

- **set_name()** This sets the **self._name** attribute of the object from a passed in parameter.
- **swim()** and **be_awesome()** are methods which display text.

We call these methods using the dot operator (.), which is used to reference an attribute or method of the object. In this case, the attribute is a method and it's called with parentheses.

Because the keyword **self** was a parameter of the methods as defined in the **Shark** class, the **sammy** object gets passed to the methods. The **self** parameter ensures that the methods have a way of referring to object attributes.

When we call the methods, however, nothing is passed inside the parentheses, the object **sammy** is being automatically passed with the dot operator.

The __init__ Method

The **__init__** method is used to define the initial state of an object.

This special method runs as soon as an object of a class is instantiated (i.e. created, constructed). This is called a constructor in other languages such as Java and C++.

This method is useful to do any initialization (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores at the beginning and at the end of the name. This indicates this is a dunder method. (double underscore)

```
# Filename: oop_init.py
class Person:
    """init runs automatically when the object is created
       This init method takes a parameter, name
    """

    def __init__(self, name):
        print("I am __init__")
        self._name = name

    # Define a method
    def say_hi(self):
        print(f"Hello, my name is {self._name}")


# Program starts here
barney = Person("Barney")
barney.say_hi()
```

Example run:

```
I am __init__
Hello, my name is Barney
```

How It Works

We define the **__init__** method as taking a parameter **name** (along with the usual self). We create a new field also called **self._name**.

When creating new instance **barney**, of the class **Person**, we do so by using the class name, followed by the arguments in the parentheses:

barney = Person("Barney")

We do not explicitly call the **__init__** method. It is called when the object is initialized.

Tutorial 7.2: Constructing Sharks

The **__init__** method is used to initialize data. It is run as soon as an object of a class is instantiated.

Classes are useful because they allow us to create many similar objects based on the same blueprint. This program creates two objects from the same class.

Create the following program and name it: **shark_2.py**

```python
"""
    Name: shark_2.py
    Author:
    Created:
    Purpose: Demonstrate object construction
"""


class Shark:
    """
        Initialize the shark object with 2 parameters
        If parameters are not passed in, the default values are used
    """

    def __init__(self, name="Name", age=0):
        self._name = name
        self._age = age

    # Define shark methods
    def swim(self):
        print(f'{self._name} is swimming.')

    def be_awesome(self):
        print(f'{self._name} is being awesome.')

    def how_old(self):
        print(f'{self._name} is {self._age} years old.')

    # Getters and setters
    def get_name(self):
        return self._name

    def get_age(self):
        return self._age

    def set_name(self, name):
        self._name = name

    def set_age(self, age):
        self._age = age


def main():
    # Set name and age of Shark object during initialization
    sammy = Shark("Sammy", 2)
    sammy.how_old()
    sammy.be_awesome()

    # Create another shark object with default values
    stevie = Shark()
    stevie.set_name("Stevie")
    stevie.set_age(1)
    stevie.how_old()
    stevie.swim()


# Call the main method
if __name__ == "__main__":
    main()
```

Example run:

```
Sammy is 2 years old.
Sammy is being awesome.
Stevie is 1 years old.
Stevie is swimming.
```

How It Works

Notice the name being passed to the object. We defined the **__init__** method with the parameter name (along with the **self** keyword) and defined a variable within the method, name.

We also defined a new class variable, **_age**. To make use of age, we created a method in the class that calls for it. Constructor methods allow us to initialize certain attributes of an object.

The _ in front of the variable name **_age** hides the variable or makes it private to the class. This is called data hiding. We only allow access to object variables through the methods.

Because the constructor method is automatically initialized, we do not need to explicitly call it, only pass the arguments in the parentheses following the class name when we create a new instance of the class.

We created a second **Shark** object called **stevie** and passed the name "Stevie" to it. Classes make it possible to create more than one object following the same pattern without creating each one from scratch.

Tutorial 7.3: Movie Class

Create a Python program named: **movie_class.py**

The movie class demonstrates more about getters and setters.

```python
"""
    Name: movie_class.py
    Author:
    Created:
    Purpose: Demonstrate class, objects and methods
"""


class Movie:
    def __init__(self, title, rating):
        self._title = title
        self._rating = rating

# ----------------- GETTERS --------------------#
    def get_title(self):
        print("Calling the title getter...")
        return self._title

    def get_rating(self):
        print("Calling the rating getter...")
        return self._rating

# ----------------- SETTERS --------------------#
    def set_title(self, title):
        print("Calling the title setter...")
        self._title = title

    def set_rating(self, rating):
        print("Calling the rating getter...")
        self._rating = rating
```

```
33  # Create Movie object with parameters
34  my_movie = Movie("The Godfather", 4.8)
35
36  # Get the title
37  print(my_movie.get_title())
38  # Get the title in a print statement
39  print(f"My favorite movie is: {my_movie.get_title()}")
40
41  my_movie.set_title(101)
42  my_movie.set_title("Princess Bride")
43  my_movie.set_rating(5.0)
44
45  print(f"My favorite movie is actually: {my_movie.get_title()}")
46  print(f"{my_movie.get_title()} is rated: {my_movie.get_rating()}")
47
```

Example run:

```
Calling the title getter...
The Godfather
Calling the title getter...
My favorite movie is: The Godfather
Calling the title setter...
Calling the title setter...
Calling the rating getter...
Calling the title getter...
My favorite movie is actually: Princess Bride
Calling the title getter...
Calling the rating getter...
Princess Bride is rated: 5.0
```

Classes in Separate Files

Classes and programs can be in separate files. A program can have several class files. This is very common in larger programs.

Tutorial 7.4: Coin Flip

Coin Flip has two classes. The **Coin** class holds all the attributes and methods to flip a coin. That is all it does, flip a coin. Nothing else.

The **coin_flip** program creates three coin objects and flips them.

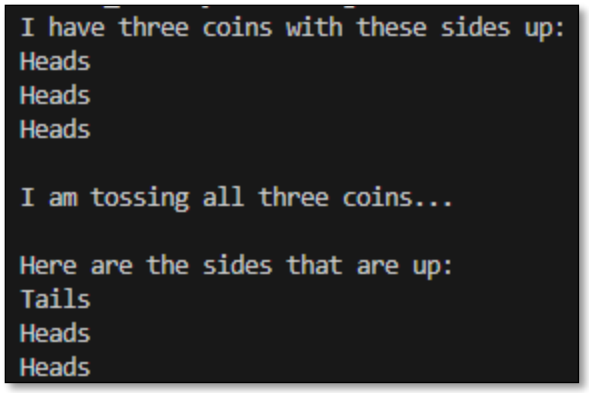
Create a Python program named **coin.py**

```python
"""
    Name: coin.py
    Author:
    Created:
    Purpose: The Coin class flips a coin
    and stores which side is up
"""

# Import the random module
import random


class Coin:
    # The __init__ method initializes the
    # _sideup data attribute with "Heads"
    def __init__(self):
        self._sideup = "Heads"

    # The toss method generates a random number
    # in the range of 0 through 1.
    # 0, sideup is set to "Heads".
    # Otherwise, sideup is set to "Tails".
    def toss(self):
        if random.randint(0, 1) == 0:
            self._sideup = "Heads"
        else:
            self._sideup = "Tails"

    def get_sideup(self):
        return self._sideup
```

Create a Python program named **coin_flip.py** that uses the Coin class.

```python
"""
    Name: coin_flip.py
    Author:
    Created:
    Purpose: Create 3 coin objects
"""

import coin


def main():
    # Create three objects from the Coin class.
    coin1 = coin.Coin()
    coin2 = coin.Coin()
    coin3 = coin.Coin()

    # Display the side of each coin that is facing up.
    print("I have three coins with these sides up:")
    print(coin1.get_sideup())
    print(coin2.get_sideup())
    print(coin3.get_sideup())
    print()

    # Toss the coin.
    print("I am tossing all three coins...")
    print()
    coin1.toss()
    coin2.toss()
    coin3.toss()

    # Display the side of each coin that is facing up.
    print("Here are the sides that are up:")
    print(coin1.get_sideup())
    print(coin2.get_sideup())
    print(coin3.get_sideup())
    print()


main()
```

Example run:



Glossary

attribute A variable that is part of a class.

class A template that can be used to construct an object. Defines the attributes and methods that will make up the object.

child class A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor An optional specially named method (__init__) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

destructor An optional specially named method (__del__) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional attributes and methods defined by the child class.

method A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object A constructed instance of a class. An object contains all the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangeably with 'object'.

parent class The class which is being extended to create a new child class. The parent class contributes all its methods and attributes to the new child class.

Assignment Submission

1. Attach all tutorials.
2. Attach screenshots showing the successful operation of each tutorial program.
3. Submit in Blackboard.