

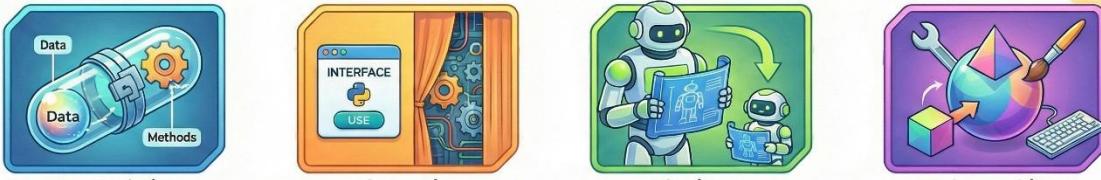
Chapter 7: Python Object-Oriented Programming

Contents

Chapter 7: Python Object-Oriented Programming	1
Videos from YouTube	2
DRY	2
Code Shape	2
Python is Object-Oriented (OOP)	3
OOP Mind Map	3
A Class Definition is a Blueprint for Objects.....	4
What is an Object-Oriented Program?	4
What is an Object?	5
What is Encapsulation?.....	5
What is Data Hiding?.....	5
Classes and Objects.....	5
The self Parameter	6
Python Classes and Objects	6
OOP Constructs	7
Defining and Using a Class	7
How It Works	8
Tutorial 7.1: Cookie Class.....	8
Getters and Setters	9
Constructor: <code>__init__</code> Method.....	10
Assignment 7.1: Pet Tracker.....	11
Tutorial 7.2: Movie Class	13
Classes in Separate Files	15
Tutorial 7.3: Coin Flip	15
Assignment 7.2: Create Your Own Class.....	18
Glossary.....	18
Assignment Submission.....	19

Python OOP: A Visual Guide to Classes & Objects

The Four Pillars of OOP



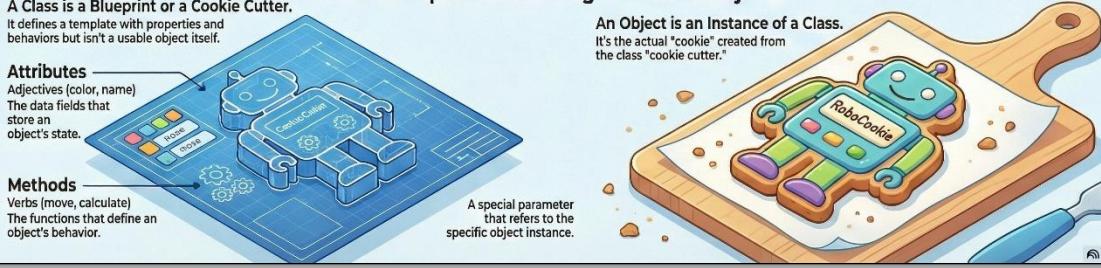
Encapsulation
Bundles data and the methods that operate on it into a single object.

Abstraction
Hides complex implementation details, showing only essential features to the user.

Inheritance
Enables a new "child" class to acquire properties from an existing "parent" class.

Polymorphism
Allows objects to take on multiple forms, enabling methods to act differently.

From Blueprint to Building: Classes & Objects



A Class is a Blueprint or a Cookie Cutter.
It defines a template with properties and behaviors but isn't a usable object itself.

Attributes
Adjectives (color, name)
The data fields that store an object's state.

Methods
Verbs (move, calculate)
The functions that define an object's behavior.

A special parameter that refers to the specific object instance.

An Object is an Instance of a Class.
It's the actual "cookie" created from the class "cookie cutter."



Red light: No AI

Time required: 120 minutes

Videos from YouTube

- [Learn Python Object Oriented Programming! \(P.O.O.P\)](#)

DRY

Don't Repeat Yourself

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.

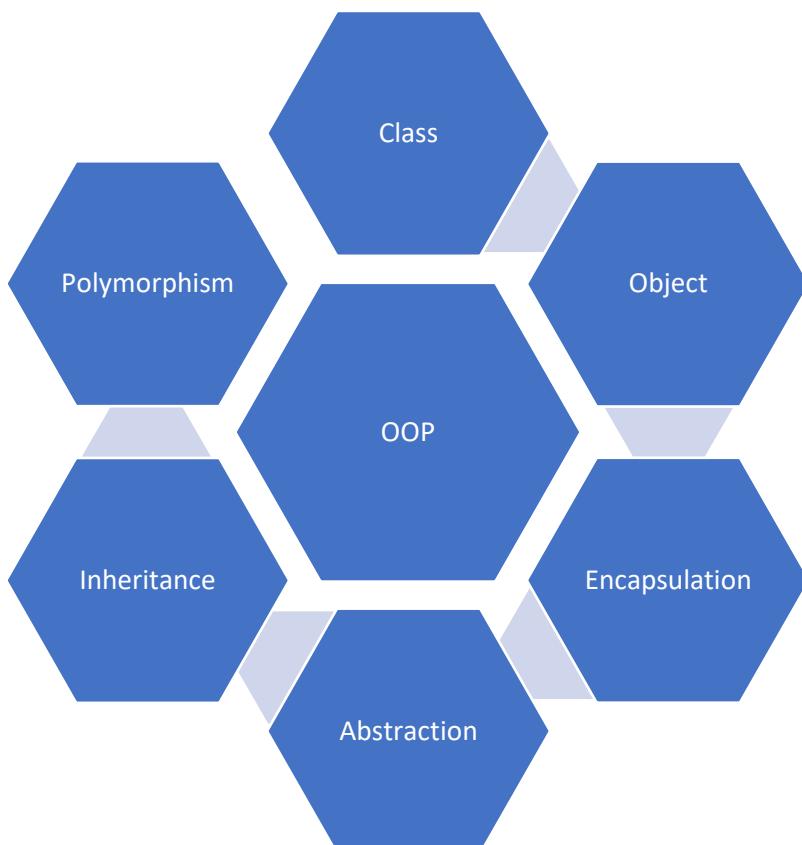
Another way of organizing your program is to combine data and functionality and wrap it inside something called an object. This is called object-oriented programming.

Python is an object-oriented programming language. Everything in Python is an object. 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.

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?

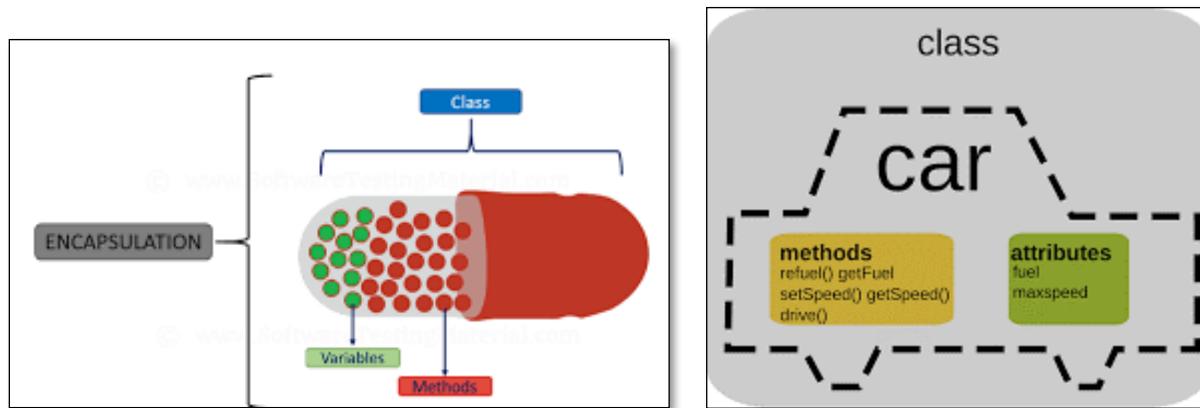
An object is a software construct that encapsulates data, along with the ability to use or modify that data.

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 **itself**.

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 Constructs

- **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.)

Defining and Using a Class

A simple class.

```
class Person:  
    def say_hi(self):  
        print("Hello!")  
  
# Create object  
p = Person()  
p.say_hi()
```

Example run:

```
Hello!
```

How It Works

- 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.
- A method is a function that “belongs to” an object. **self** is the parameter that ties the method to an object. **If** refers to the object. The object owns the method.
- We create an object-instance of this class using the name of the class followed by a pair of parentheses.
- We use the method with dot notation. **p.say_hi()**

Tutorial 7.1: Cookie Class

Here is a tutorial using methods and attributes to create different types of cookies.

```

1 # Filename: cookie_class.py
2 class Cookie:
3
4     def apply_frosting(self, frosting):
5         self.frosting = frosting
6
7     def cut_cookie(self, shape):
8         self.shape = shape
9
10    def display_cookie(self):
11        return f"Cookie shape: {self.shape} Frosting: {self.frosting}"
12
13
14 # Create cookie1 object
15 cookie1 = Cookie()
16 # Call object methods
17 cookie1.cut_cookie("Star")
18 cookie1.apply_frosting("Chocolate")
19 print(cookie1.frosting)
20 print(cookie1.display_cookie())
21
22 # Create cookie2 object
23 cookie2 = Cookie()
24 # Call object methods
25 cookie2.cut_cookie("Reindeer")
26 cookie2.apply_frosting("Vanilla")
27 print(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):
        return f"Hello, how are you? My name is {self._name}."

paul = Person()
paul.set_hair_color("brown")
paul.set_name("Paul")
print(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.
```

Constructor: `__init__` Method

- `__init__()` is a **special method** that runs when the object is created.
- Used to initialize attributes.

```

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

    def intro(self):
        return f"My name is {self.name}, and I am {self.age} years old."

# Create object
bob = Person("Bob", 30)
print(bob.intro())

```

Example run:

```
My name is Bob, and I am 30 years old.
```

Assignment 7.1: Pet Tracker

Starter Code:

```

class Pet:
    def __init__(self, name, species):
        self._name = name
        self._species = species

    def display_info(self):
        info = f"Pet Name: {self._name} /nSpecies: {self._species}"
        info += f"/n{self._age}"
        return info

pet1 = Pet("Buddy", "Dog")
print(pet1.display_info())

```

Exercise 1: Add More Attributes

Add the following to the class:

- age (integer)
- color (string)

Then update `__init__()` and `display_info()` to include them.

Exercise 2: Create More Pet Objects

Create two more pets:

- A cat named "Whiskers", age 3, color "gray"
- A parrot named "Polly", age 2, color "green"

Call `display_info()` for each pet.

Exercise 3: Add a Method

Add a method named `make_sound(self)` that prints a sound based on the species:

- Dog → "Woof!"
- Cat → "Meow!"
- Parrot → "Squawk!"
- Unknown → "Unknown sound"

Use an `if/elif` structure inside the method.

Exercise 4: Add Getters and Setters

Use a leading underscore to hide the `age` field.

Create:

- `get_age(self)` – returns the pet's age
- `set_age(self, new_age)` – sets the pet's age

Update and print the age of one of your pets using these methods.

```
Pet Name: Buddy
Species: Dog
Age: 5
Color: Brown
-----
Pet Name: Whiskers
Species: Cat
Age: 3
Color: Gray
-----
Pet Name: Polly
Species: Parrot
Age: 2
Color: Green
-----
Buddy says: Woof!
Whiskers says: Meow!
Polly says: Squawk!

Buddy's updated age is: 6
Whiskers just had a birthday! New age: 4

Looping through pets:
Buddy says: Woof!
Whiskers says: Meow!
Polly says: Squawk!
```

Tutorial 7.2: Movie Class

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

The movie class demonstrates more about getters and setters.

```
1 """
2     Name: movie_class.py
3     Author:
4     Created:
5     Purpose: Demonstrate class, objects and methods
6 """
7
8
9 class Movie:
10     def __init__(self, title, rating):
11         self._title = title
12         self._rating = rating
13
14 # ----- GETTERS -----
15     def get_title(self):
16         print("Calling the title getter...")
17         return self._title
18
19     def get_rating(self):
20         print("Calling the rating getter...")
21         return self._rating
22
23 # ----- SETTERS -----
24     def set_title(self, title):
25         print("Calling the title setter...")
26         self._title = title
27
28     def set_rating(self, rating):
29         print("Calling the rating setter...")
30         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 7.3: 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**

```
1  """
2      Name: coin.py
3      Author:
4      Created:
5      Purpose: The Coin class flips a coin
6      and stores which side is up
7  """
8
9  # Import the random module
10 from random import randint
11
12
13 class Coin:
14     # The __init__ method initializes the
15     # _sideup data attribute with "Heads"
16     def __init__(self):
17         self._sideup = "Heads"
18
19     # The toss method generates a random number
20     # in the range of 0 through 1.
21     # 0, sideup is set to "Heads".
22     # Otherwise, sideup is set to "Tails".
23     def toss(self):
24         if randint(0, 1) == 0:
25             self._sideup = "Heads"
26         else:
27             self._sideup = "Tails"
28
29     def get_sideup(self):
30         return self._sideup
```

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

```
1 """
2     Name: coin_flip.py
3     Author:
4     Created:
5     Purpose: Create 3 coin objects
6 """
7
8 import coin
9
10
11 def main():
12     # Create three objects from the Coin class.
13     coin1 = coin.Coin()
14     coin2 = coin.Coin()
15     coin3 = coin.Coin()
16
17     # Display the side of each coin that is facing up.
18     print("I have three coins with these sides up:")
19     print(coin1.get_sideup())
20     print(coin2.get_sideup())
21     print(coin3.get_sideup())
22     print()
23
24     # Toss the coin.
25     print("I am tossing all three coins...")
26     print()
27     coin1.toss()
28     coin2.toss()
29     coin3.toss()
30
31     # Display the side of each coin that is facing up.
32     print("Here are the sides that are up:")
33     print(coin1.get_sideup())
34     print(coin2.get_sideup())
35     print(coin3.get_sideup())
36     print()
37
38
39 main()
```

Example run:

```
I have three coins with these sides up:  
Heads  
Heads  
Heads  
  
I am tossing all three coins...  
  
Here are the sides that are up:  
Tails  
Heads  
Heads
```

Assignment 7.2: Create Your Own Class

Create a program that uses a class in a separate file.

- 2 or more attributes
- 2 or more methods
- Getters and setters for each attribute
- `__init__` method for the 2 attributes

You can create any type of program you wish.

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.