

Chapter 7: Object-Oriented Programming

Contents

Chapter 7: Object-Oriented Programming	1
DRY.....	2
Code Shape.....	2
Python is Object-Oriented (OOP).....	2
A Class Definition is a Blueprint for Objects.....	3
What is an Object-Oriented Program?	4
What is an Object?	4
What is Encapsulation?	4
What is Data Hiding?.....	4
Classes and Objects.....	4
The self	5
Classes	5
How It Works	5
Methods	6
How It Works	6
Python Objects	6
Tutorial 7.1 – Sammy the Shark	9
Explanation.....	10
The __init__ method.....	10
How It Works	11
Tutorial 7.2 – Constructing Sharks	11
Explanation.....	12
Class and Object Variables.....	13
How It Works	15
Setter and Getter Methods.....	16
Tutorial 7.3 – More Sharks	16
Explanation.....	18
Movie Class	18

Classes in Separate Files	19
Tutorial 7.4 – Coin Flip.....	20
Designing an Object-Oriented Program	21
Pickle.....	22
How It Works	24
Inheritance.....	24
Tutorial 7.5 – Parent and Child Fish (and Sharks).....	25
Explanation.....	27
Tutorial 7.6 - Convert Functional to OOP	27
Glossary.....	30
Assignment Submission.....	30

Time required: 90 minutes

DRY

Don't Repeat Yourself

Code Shape

Please group program code as follows.

- Declare constants and variables
- Get input
- Calculate
- Display

Python is Object-Oriented (OOP)

In all the programs we have written until now, we have designed our program around functions: blocks of statements which manipulate data. This is called the procedure-oriented way of programming.

There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the object-oriented programming paradigm. Most of the time you can use procedural or functional

programming. When you are writing large programs or have a problem that is better suited to this method, you can use object-oriented programming techniques.

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.

As an example, 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.

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.

Unlike Java and C++, once a Python object is instantiated, it can be modified in ways that no longer follow the blueprint of the class through the addition of new data members.

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?

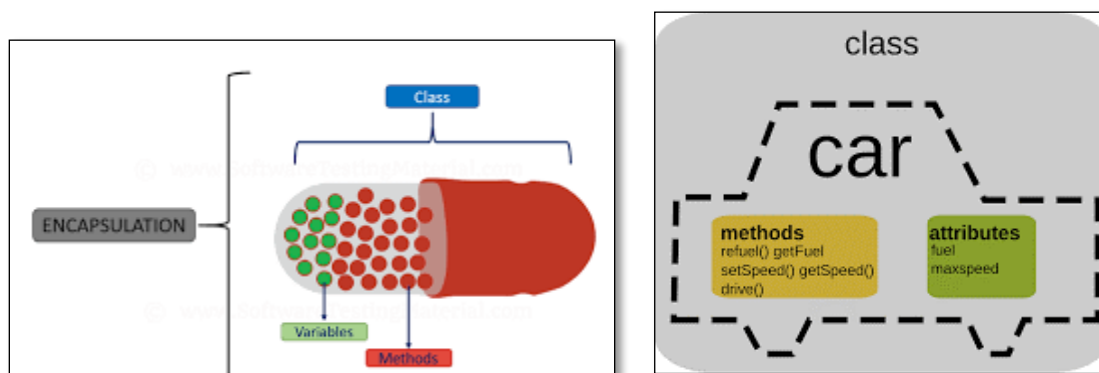
An object is a software construct that encapsulates data, along with the ability to use or modify that data.

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

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 functions and class variables of that object.

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

Classes

The simplest class possible is shown in the following example.

```
class Person:
    pass # An empty block
p = Person()
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.

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 is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra **self** variable.

We will now see an example.

```
class Person:
    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.

Python Objects

At a basic level, an object is simply some code plus data structures that are smaller than a whole program. Defining a function 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 functions (which we call methods) as well as data that is used by those functions. 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).

The **__** (double 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 called name mangling. Only the class and its methods can access and change the data directly. This is called data hiding.

```
# Filename: party_1.py
# Define the class
class PartyAnimal:
    # Class data attribute
    party_count = 0

    # Define class party method
    def party(self):
        self.party_count = self.party_count + 1
        print(f"Party {self.party_count}")

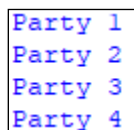
def main():

    # Create an object from the class
    Jason = PartyAnimal()
    Jose = PartyAnimal()

    # Use object methods
    Jason.party()
    Jason.party()
    Jose.party()
    Jason.party()
    Jason.party()

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

Example run:



```
Party 1
Party 2
Party 3
Party 4
```

Each method looks like a function, starting with the `def` keyword and consisting of an indented block of code. This object has one attribute **party_count** and one method **party(self)**. The method has a special first parameter **self**.

Just as the **def** keyword does not cause function code to be executed, the **class** keyword does not create an object. The **class** keyword defines a template indicating what data and code will be contained in each object of type **PartyAnimal**.

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

The following photo shows a class and two objects.



If we continue through this sample program, we see the first executable line of code:

```
party_animal = PartyAnimal()
```

This is where we instruct Python to construct (i.e., create) an object or instance of the class **PartyAnimal**. It looks like a function call to the class itself. Python constructs the object with the right data and methods and returns the object which is then assigned to the variable `an`.

When the **PartyAnimal** class is used to construct an object, the variable **party_animal** is used to point to that object. We use **party_animal** to access the code and data for that instance of the **PartyAnimal** class.

Each **PartyAnimal** object/instance contains within it a variable **party_count** and a method/function named **party**. We call the party method in this line:

```
party_animal.party()
```

When the **party** method is called, the first parameter (which we call by convention **self**) points to the instance of the **PartyAnimal** object that party is called from. Within the party method, we see the line:


```
self.party_count = self.party_count + 1
```

This syntax using the dot operator is saying 'the party_count within self.' Each time **party()** is called, the internal **party_count** value is incremented by 1 and the value is printed out.

When the program executes, it produces the following output:

```
Party 1
Party 2
Party 3
Party 4
```

The object is constructed, party method is called four times, both incrementing and printing the value for **party_count** within the **party_animal** object.

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**

```
1 '''
2     Name: shark.py
3     Author:
4     Created:
5     Purpose: Demonstrate class, objects and methods
6 '''
7
8 class Shark:
9     ''' Define shark methods '''
10    def swim(self):
11        print("The shark is swimming.")
12    def be_awesome(self):
13        print("The shark is being awesome.")
14
15 def main():
16     # Create a shark object
17     sammy = Shark()
18     # Call shark methods
19     sammy.swim()
20     sammy.be_awesome()
21
22 # Call the main function
23 if __name__ == "__main__":
24     main()
```

Explanation

The **Shark** object **sammy** is using the two methods **swim()** and **be_awesome()**. We called these 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, like how you would also call with a function.

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.

Example run:

```
The shark is swimming.  
The shark is being awesome.
```

The `__init__` method

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

This special method is run 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.

```
# Filename: person_class.py
# Demonstration class

class Person:
    # init runs automatically when the object is created
    # This init method takes a single parameter: name
    def __init__(self, name):
        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:

```
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 **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. This is the special significance of this method.

Tutorial 7.2 – Constructing Sharks

The constructor method is used to initialize data. It is run as soon as an object of a class is instantiated. Also known as the `__init__` method, it will be the first definition of a class.

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: **shark2.py**

```

1  '''
2      Name: shark2.py
3      Author:
4      Created:
5      Purpose: Demonstrate object construction
6  '''
7
8  class Shark:
9      ''' Initialize the shark object with 2 parameters'''
10     def __init__(self, name, age):
11         self.__name = name
12         self.__age = age
13
14     ''' Define shark methods '''
15     def swim(self):
16         print(f'{self.__name} is swimming.')
17     def be_awesome(self):
18         print(f'{self.__name} is being awesome.')
19     def how_old(self):
20         print(f'{self.__name} is {self.__age} years old.')
21
22 def main():
23     # Set name and age of Shark object during initialization
24     sammy = Shark("Sammy", 2)
25     sammy.how_old()
26     sammy.be_awesome()
27     # Create another shark object
28     stevie = Shark("Stevie", 1)
29     stevie.how_old()
30     stevie.swim()
31
32 # Call the main method
33 if __name__ == "__main__":
34     main()

```

Example run:

```

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

```

Explanation

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.

Class and Object Variables

We have already discussed the functionality part of classes and objects (i.e. methods), now let us learn about the data part. The data part, i.e. fields, are nothing but ordinary variables that are bound to the namespaces of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called name spaces.

There are two types of fields - class variables and object variables which are classified depending on whether the class or the object owns the variables respectively.

Class variables are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.

Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance. An example will make this easy to understand. Object variables are defined with the `__init__` constructor using **self**.

```
'''
    Filename: robot.py
    Purpose: Demonstrate object and class variables with a robot that has a
    name
    Define Robot class
    '''
class Robot:
    # A class or static variable, counting the number of robots
    __population = 0
    # An object variable specific to each object
    __name = ' '
```

```

def __init__(self, name):
    # Initializes the attribute
    self.__name = name
    print(f'Initializing {self.__name}')

    # When this person is created, the robot
    # adds to the population
    Robot.__population += 1

def die(self):
    # I am dying
    print(f'{self.__name} is being destroyed!')
    Robot.__population -= 1
    if Robot.__population == 0:
        print(f'{self.__name} was the last one.')
    else:
        print(f'There are still {Robot.__population} robots working.')

def say_hi(self):
    # Greeting by the robot. Yeah, they can do that
    print(f'Greetings, my masters call me {self.__name}')

@classmethod
def how_many(cls):
    # A class method shared by all objects
    # Prints the current population
    print(f'We have {cls.__population} robots.')

def main():
    droid1 = Robot("R2-D2")
    droid1.say_hi()
    Robot.how_many()

    droid2 = Robot("C-3PO")
    droid2.say_hi()
    Robot.how_many()

    print("\nRobots do some work here.\n")

    print("Robots have finished their work. Let's destroy them.")
    droid1.die()

```

```
droid2.die()

Robot.how_many()

if __name__ == "__main__":
    main()
```

Example run:

```
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

How It Works

This is a long example but helps demonstrate the nature of class and object variables. Here, `population` belongs to the `Robot` class and hence is a class variable. The `name` variable belongs to the object (it is assigned using `self`) and hence is an object variable.

We refer to the `__population` class variable as `Robot.__population` and not as `self.__population`. We refer to the object variable `__name` using `self.__name` notation in the methods of that object. Remember this simple difference between class and object variables. Also note that an object variable with the same name as a class variable will hide the class variable!

`how_many` is a method that belongs to the class and not to the object. This means we can define it as either a `classmethod` or a `staticmethod` depending on whether we need to know which class we are part of. Since we refer to a class variable, let's use `classmethod`.

We have marked the `how_many` method as a class method using a decorator. Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the same as calling:

```
how_many = classmethod(how_many)
```

Observe that the `__init__` method is used to initialize the Robot instance with a name. In this method, we increase the population count by 1 since we have one more robot being added. Observe that the values of `self.__name` is specific to each object which indicates the nature of object variables.

Remember, that you must refer to the variables and methods of the same object using the `self` only. This is called an attribute reference.

All class members are public. One exception: If you use data members with names using the underscore prefix such as `_privatevar`. This is a convention that Python programmers use to indicate that this is a private data attribute.

The convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects.

Setter and Getter Methods

Class methods that set and retrieve attributes are called setters and getters. In the example below, we set the shark's name and get the shark's name.

Tutorial 7.3 – More Sharks

Create a Python program named: **shark3.py**


```

1  '''
2      Name: shark3.py
3      Author:
4      Created:
5      Purpose: Demonstrate object construction
6  '''
7
8  class Shark:
9      # Class variables
10     # Shared by all instances of the class
11     animal_type = "We are all fish!"
12     location = "ocean"
13
14     # Constuctor
15     def __init__(self, name, age):
16         self.__name = name
17         self.__age = age
18
19     ''' Define shark methods '''
20     # Method with instance variable followers
21     def set_followers(self, followers):
22         print(f'{self.__name} has {followers} followers.')
23
24     def swim(self):
25         print(f'{self.__name} is swimming.')
26     def be_awesome(self):
27         print(f'{self.__name} is being awesome.')
28     def how_old(self):
29         print(f'{self.__name} is {self.__age} years old.')
30
31 def main():
32     # Set name of Shark object during construction
33     sammy = Shark("Sammy", 2)
34     sammy.how_old()
35     sammy.be_awesome()
36     sammy.set_followers(77)
37     print(sammy.animal_type)
38     # Create another shark object
39     stevie = Shark("Stevie", 1)
40     stevie.how_old()
41     stevie.swim()
42     stevie.set_followers(45)
43     print(f'{stevie.animal_type} in the {stevie.location}')
44
45 # Call main function
46 if __name__ == "__main__":
47     main()

```

Example run:

```
Sammy is 2 years old.  
Sammy is being awesome.  
Sammy has 77 followers.  
We are all fish!  
Stevie is 1 years old.  
Stevie is swimming.  
Stevie has 45 followers.  
We are all fish! in the ocean
```

Explanation

Class variables allow us to define variables upon constructing the class. These variables and their associated values are then accessible to each instance of the class.

Instance variables, owned by objects of the class, allow for each object or instance to have different values assigned to those variables. Unlike class variables, instance variables are defined within methods.

Movie Class

The movie class demonstrates getters and setters.

- The `__init__` method defines name and rating
- The `get_title` and `get_rating` allow us to access the attributes
- The `set_title` and `set_rating` allow us to set the attributes

```

1  """
2      Name: movie_class.py
3      Author:
4      Created:
5      Purpose: Demonstrate class, objects and methods
6  """
7
8
9  class Movie:
10
11      def __init__(self, title, rating):
12          self._title = title
13          self._rating = rating
14
15      #----- GETTERS -----#
16      def get_title(self):
17          return self._title
18
19      def get_rating(self):
20          return self._rating
21
22      #----- SETTERS -----#
23      def set_title(self, title):
24          # Test the incoming title
25          # to make sure it is a string
26          if isinstance(title, str):
27              self._title = title
28          else:
29              print("The title must be string.")
30
31      def set_rating(self, rating):
32          self._rating = rating
33
34
35      # Create Movie object
36      my_movie = Movie("The Godfather", 4.8)
37
38      # Get the title
39      print(my_movie.get_title())
40      # Get the title in a print statement
41      print(f"My favorite movie is: {my_movie.get_title()}")
42
43      my_movie.set_title(101)
44      my_movie.set_title("Princess Bride")
45      my_movie.set_rating(5.0)
46      print(f"My favorite movie is actually: {my_movie.get_title()}")
47      print(f"{my_movie.get_title()} is rated: {my_movie.get_rating()}")
48

```

Classes in Separate Files

Classes 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 but doesn't flip a coin. The **CoinFlip** program creates three coin objects and flips them.

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

```
1  '''
2      The coin class flips a coin
3  '''
4  # Import the random module
5  import random
6
7  class Coin:
8      # The __init__ method initializes the
9      # __sideup data attribute with 'Heads'
10     def __init__( self ):
11         self.__sideup = 'Heads'
12
13     # The toss method generates a random number
14     # in the range of 0 through 1. If the number
15     # is 0, then sideup is set to 'Heads'.
16     # Otherwise, sideup is set to 'Tails'.
17     def toss(self):
18         if random.randint(0, 1) == 0:
19             self.__sideup = 'Heads'
20         else:
21             self.__sideup = 'Tails'
22
23     # The get_sideup method returns the value
24     # referenced by sideup.
25     def get_sideup(self):
26         return self.__sideup
```

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

```

1  '''
2      This program imports the coin class
3      creates three instances of the Coin class
4  '''
5  # Import the coin class
6  import coin
7
8  def main():
9      # Create three objects from the Coin class
10     coin1 = coin.Coin()
11     coin2 = coin.Coin()
12     coin3 = coin.Coin()
13
14     # Display the side of each coin that is facing up
15     print('I have three coins with these sides up:')
16     print(coin1.get_sideup())
17     print(coin2.get_sideup())
18     print(coin3.get_sideup())
19     print()
20
21     # Toss the coin
22     print('I am flipping the coins...')
23     print()
24     coin1.toss()
25     coin2.toss()
26     coin3.toss()
27
28     # Display the side of each coin that is facing up
29     print('Here is how the coins landed:')
30     print(coin1.get_sideup())
31     print(coin2.get_sideup())
32     print(coin3.get_sideup())
33     print()
34
35     # Call the main function
36     if __name__ == '__main__':
37         main()

```

Designing an Object-Oriented Program

The first step is to identify the classes the program will need.

1. Identify the real-world objects, the nouns.

Customer
Address
Car
labor charges
Toyota

Some nouns contain other nouns. A Car can also be a Toyota. A customer would have an address. We need a Car and a Customer class.

2. Determine what describes or makes up the class, the data attributes.

Customer: Name, Address, Phone

3. Determine the actions, the verbs, the methods.

Move forward()

Make purchase()

create invoice()

Customer: set name, set address, get address

Keep everything that pertains to the class in the class.

Pickle

Python provides a standard module called **pickle** which you can use to store any plain Python object in a file and then get it back later. This is called storing the object persistently.

```
# Filename: shoplist_pickle.py

# Import the pickle module
import pickle

# The name of the file where we will store the serialized object
shoplist_file = "shoplist.pkl"

# List of things to buy
shoplist = ["apple", "mango", "carrot"]

# Write to the file
f = open(shoplist_file, "wb")

# Dump the object to a file
pickle.dump(shoplist, f)
print(f"shoplist written to {shoplist_file}")
f.close()

# Destroy the shoplist variable
del shoplist
print(f"shoplist destroyed in memory.")

# Read back from the storage
f = open(shoplist_file, "rb")
print(f"shoplist read back from {shoplist_file}")

# Load the object from the file
storedlist = pickle.load(f)
print(storedlist)

# Close the file
f.close()
```

Example run:

```
shoplist written to shoplist.data
shoplist destroyed in memory.
shoplist read back from shoplist.data
['apple', 'mango', 'carrot']
```

How It Works

To store an object in a file, **open** the file in write binary mode and then call the **dump** function of the **pickle** module. This process is called **pickling**.

We retrieve the object using the **load** function of the **pickle** module which returns the object. This process is called **unpickling**.

Inheritance

Another powerful feature of object-oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the parent class and the new class the child class.

For this example, we move our **PartyAnimal** class into its own file. Then, we can 'import' the **PartyAnimal** class in a new file and extend it, as follows:

```
# Import the PartyAnimal class from party.py
from party import PartyAnimal
# Defining and extending the
# CricketFan class based on PartyAnimal
# All PartyAnimal methods and data is available
class CricketFan(PartyAnimal):
    __points = 0
    # CricketFan method calling local and inherited party methods
    def six(self):
        self.__points = self.__points + 6
        # Inherited method from parent class
        self.party()
        print(self.__name, "points", self.__points)
sally = PartyAnimal("Sally")
sally.party()
jim = CricketFan("Jim")
jim.party()
jim.six()
# Display the attributes of the j object
# Some we defined, some are automatically defined when an object is created
print(dir(j))
```

When we define the **CricketFan** class, we indicate that we are extending the **PartyAnimal** class. This means that all the variables (**x**) and methods (**party**) from the **PartyAnimal** class are inherited by the **CricketFan** class. For example, within the **six** method in the **CricketFan** class, we call the party method from the **PartyAnimal** class.

As the program executes, we create **sally** and **jim** as independent instances of **PartyAnimal** and **CricketFan**. The **jim** object has additional capabilities beyond the **sally** object.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['_class_', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_s
ubclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas
shook__', '__weakref__', 'name', 'party', 'points', 'six', 'x']
```

In the **dir** output for the **jim** object (instance of the **CricketFan** class), we see that it has the attributes and methods of the parent class, as well as the attributes and methods that were added when the class was extended to create the **CricketFan** class.

Tutorial 7.5 – Parent and Child Fish (and Sharks)

Create the following Python program named: **fish.py**

```

1  '''
2      Name: fish.py
3      Author:
4      Created:
5      Purpose: Demonstrate Parent and Child classes
6  '''
7  class Fish:
8      ''' Parent class '''
9      # Initialize object and private class variables
10     def __init__(self, first_name, last_name="Fish",
11                 # Hard code object variables
12                 skeleton="bone", eyelids=False):
13         self.__first_name = first_name
14         self.__last_name = last_name
15         self.__skeleton = skeleton
16         self.__eyelids = eyelids
17
18     # Define class methods
19     def swim(self):
20         print(f'The fish is swimming.')
21     def swim_backwards(self):
22         print(f'The fish can swim backwards.')
23
24     # Getters and setters
25     def get_first_name(self):
26         return self.__first_name
27     def get_last_name(self):
28         return self.__last_name
29     def set_first_name(self, first_name):
30         self.__first_name = first_name
31     def set_last_name(self, last_name):
32         self.__last_name = last_name
33
34     def print_name(self):
35         print(f'{self.__first_name} {self.__last_name}')
36
37 class Clownfish(Fish):
38     ''' Child class '''
39     def live_with_anemone(self):
40         print(f'The clownfish is coexisting with sea anemone.')
41
42 class Shark(Fish):
43     ''' Override the parent class '''
44     def swim_backwards(self):
45         print(f'The shark cannot swim backwards, but can sink backwards.')
46

```

```

47 def main():
48     # Create object with one argument
49     casey = Clownfish("Casey")
50     print(f'Create a clownfish named ')
51     print(f'{casey.get_first_name()} {casey.get_last_name()}')
52     casey.swim()
53     casey.live_with_anemone()
54     casey.swim_backwards()
55     # Create object with two arguments
56     shari = Shark("Shari", "the Shark")
57     print(f'Create a shark named Shari the Shark using two arguments')
58     shari.swim_backwards()
59     shari.set_last_name("the Salmon")
60     print(f'Change Shari\'s last name with a set method.')
61     shari.print_name()
62
63 # Call the main function
64 if __name__ == "__main__":
65     main()

```

Example run:

```

Create a clownfish named
Casey Fish
The fish is swimming.
The clownfish is coexisting with sea anemone.
The fish can swim backwards.
Create a shark named Shari the Shark using two arguments
The shark cannot swim backwards, but can sink backwards.
Change Shari's last name with a set method.
Shari the Salmon

```

Explanation

With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the pass keyword.

The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

The Clownfish class inherits all the attributes and methods from its parent, Fish. It has a special method will permit it to live with sea anemone.

The Shark method **swim_backwards()** prints a different string than the one in the Fish parent class because sharks are not able to swim backwards in the way that bony fish can.

Tutorial 7.6 - Convert Functional to OOP

In Chapter 5 we did a tutorial called **purchase_price_with_functions.py** This tutorial shows how to turn this into an OOP. Both programs give the same results. The OOP is much more expandable and simpler to use.

This is the original tutorial.

```
1 '''
2     Name: purchase_price_with_functions.py
3     Author:
4     Created:
5     Purpose: Calculate total sale with functions
6 '''
7
8 def main():
9     '''    Return functions without arguments    '''
10    purchase_price, quantity = get_input()
11
12    # Return function with arguments
13    total_sale = calculate_total_sale( purchase_price, quantity )
14
15    # Void function
16    display_sale( purchase_price, quantity, total_sale )
17
18 def get_input():
19     '''    Get purchase price and quantity from the user    '''
20    purchase_price = float(input('Enter the purchase price: '))
21    quantity = int(input('Enter the quantity: '))
22    # Return two values
23    return purchase_price, quantity
24
25 def calculate_total_sale( price, quantity ):
26     '''    Calculate the total sale    '''
27    total_sale = price * quantity
28    return total_sale
29
30 def display_sale( purchase_price, quantity, total_sale):
31     '''
32         Display information about the sale
33         {"Purchase Price":>15 formats string literal
34         > align right 15 field width
35     '''
36    print (f'\n{"Purchase Price":>15} ${purchase_price:,.2f}')
37    print (f'{"Quantity":>15} {quantity}')
38    print (25 * '-')
39    print (f'{"Total Sale":>15} ${total_sale:,.2f}')
40
41 # If a standalone program, call the main function
42 # Else, use as a module
43 if __name__ == '__main__':
44     main()
```

This is the same program written in OOP.

```
1 '''
2     Name: purchase_price_oop.py
3     Author:
4     Created:
5     Purpose: Calculate total sale with OOP
6 '''
7
8 class PurchasePrice():
9     # Initializes object
10    def __init__( self ):
11        # Private class variables
12        self.__purchase_price = 0
13        self.__quantity = 0
14        self.__total_sale = 0
15
16    def get_input( self ):
17        '''    Get purchase price and quantity from the user    '''
18        self.__purchase_price = float(input('Enter the purchase price: '))
19        self.__quantity = int(input('Enter the quantity: '))
20
21    def calculate_total_sale( self ):
22        '''    Calculate the total sale    '''
23        self.__total_sale = self.__purchase_price * self.__quantity
24
25    def display_sale( self ):
26        '''
27            Display information about the sale
28            {"Purchase Price":>15 formats string literal
29             > align right 15 field width
30        '''
31        print (f'\n{"Purchase Price":>15} ${self.__purchase_price:,.2f}')
32        print (f'{"Quantity":>15} {self.__quantity}')
33        print (25 * '-')
34        print (f'{"Total Sale":>15} ${self.__total_sale:,.2f}')
35
36    def main():
37        '''    Create a PurchasePrice() object    '''
38        purchase_price = PurchasePrice()
39
40        # Call object methods
41        purchase_price.get_input()
42        purchase_price.calculate_total_sale()
43        purchase_price.display_sale()
44
45    # If a standalone program, call the main function
46    # Else, use as a module
47    if __name__ == '__main__':
48        main()
```

Example run:

```
Enter the purchase price: 20
Enter the quantity: 2

Purchase Price: $20.00
Quantity: 2
-----
Total Sale: $40.00
```

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

Attach all program files to the assignment in BlackBoard.