Learning Objectives

- **Define the term encapsulation**
- **Explain how Python implements private and public attributes**
- **Explain the limitations of Python conventions for encapsulation**
- **Differentiate between `_attribute` and `__attribute`**

What is Encapsulation?

What is Encapsulation?

Encapsulation is a concept in which related data and methods are grouped together, and in which access to data is restricted. Grouping related data and methods makes thinking about your program a bit easier. Hiding or restricting how the user interacts with the data can keep the user from making unwanted changes. Encapsulation is not unique to Python, but the ways in which each programming language implement encapsulation are a bit different.

The two main ideas of data restriction are public and private. These adjectives can refer to both attributes and methods. Public means that the attribute or method can be accessed by an instance of a class. Private means that the attribute or method can only be accessed by the class itself.

▼ **Python does not use the `public` and `private` keywords**

Some programming languages, like Java, explicitly use the keywords `public` and `private` (see the code snippet below). Python does not. Python still acknowledges `public` and `private`, but does so in a unique way.

```
public class House {
     private String address;
     private int rooms;

     public House(String address, int rooms) {
         this.address = color;
     }

     public static int getRooms() {
         return rooms;
     }

     public static void setRooms(int num) {
         rooms = num;
     }
}
```

Classes as Encapsulation

Classes in Python are a form of encapsulation; they group together related data and methods. In the image below, the attributes `num1` and `num2` are grouped together with the methods `describe` and `sum`. They are all a part of

`ExampleClass` (highlighted in red). The instance `my_example` is not a part of the class itself; it is considered to be separate.

```
class ExampleClass:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2

    def describe(self):
        return f"My numbers are: {self.num1} and {self.num2}."

    def sum(self):
        return self.num1 + self.num2

my_example = ExampleClass(5, 7)
print(my_example.describe())
print(my_example.sum())
```

Classes as Encapsulation

By default, however, classes in Python do not hide or restrict access to data. Enter the code below into the IDE and run it. There are no error messages. Therefore, all attributes and methods in a Python class are public by default.

```
class Phone:
  def __init__(self, make, storage, megapixels):
    self.make = make
    self.storage = storage
    self.megapixels = megapixels

my_phone = Phone("iPhone", 256, 12)
print(my_phone.make)
print(my_phone.storage)
print(my_phone.megapixels)
```

No Data Restrictions

A traditional class in Python does not provide any restrictions to accessing data because everything is public by default. That means any instance of the `Phone` class can change its attributes. Adjust your code to look like this:

```
my_phone = Phone("iPhone", 256, 12)
print(my_phone.storage)
my_phone.storage = 64
print(my_phone.storage)
```

This is a trivial example of downgrading the storage of the phone from 256GB to 64GB. However, unexpected changes to instance variables can cause problems in your code.

challenge

Try this variation:

- Change `my_phone.model` to `True`
- Change `my_phone.storage` to `"256GB"`
- Change `my_phone.megapixels` to `-32`

▼ **Why restrict data access?**

Having public instance variables might not seem like a bad idea. The three variations you just coded either changed the data type or changed the value to something that does not make sense given the current context. Because these instance variables now have unexpected values or data types, you may see bugs appear in your code. Public instance variables increase the possibility for errors. We will see later on how encapsulation can be used to protect your code from such errors.

Encapsulation Through Convention

Encapsulation Through Convention

We saw how attributes and methods in a class are public by default in Python. We also talked about how Python does not use the `public` and `private` keywords. Instead, the Python community relies on a convention to signify private methods and instance variables. When programmers use a single underscore (_) before an attribute or method name, that attribute or method is considered to be private.

```
class Phone:
  def __init__(self, model, storage, megapixels):
    self._model = model
    self._storage = storage
    self._megapixels = megapixels
```

The code above has a single underscore before each of the instance variables, which means programmers will treat them as private. Add the following code to the `Phone` class and click the `TRY IT` button.

```
class Phone:
  def __init__(self, model, storage, megapixels):
    self._model = model
    self._storage = storage
    self._megapixels = megapixels

my_phone = Phone("iPhone", 256, 12)
print(my_phone.__dict__)
```

The `__dict__` attribute is found in every object in Python. Its job is to stores all of the attributes in a class. You should see the following output:

```
{'_model': 'iPhone', '_storage': 256, '_megapixels': 12}
```

These are all of the attributes in the `Phone` class as well as their values.

challenge

Try this variation:

- Extend the `Phone` class by adding the private attribute `_carrier`. Instantiate the object with the string `"AT&T"`. Print the `__dict__` attribute to make sure your code worked properly.

▼ **Solution**

```
class Phone:
  def __init__(self, model, storage, megapixels, carrier):
    self._model = model
    self._storage = storage
    self._megapixels = megapixels
    self._carrier = carrier

my_phone = Phone("iPhone", 256, 12, "AT&T")
print(my_phone.__dict__)
```

Does the Single Underscore Really Mean Private?

No. The single underscore is a convention. That is, an informal agreement to recognize that attributes and methods with a single underscore are private. The Python interpreter does not enforce any restrictions that make these attributes private.

```
class PrivateClass:
  def __init__(self):
    self._private_attribute = "I am a private attribute"

obj = PrivateClass()
print(obj._private_attribute)
```

If `_private_attribute` really were private, Python would throw an error message. Instead, Python sees `_private_attribute` as being public and prints its value. Python does not have truly private attributes and methods, though Python can approximate this behavior.

challenge

Try this variation:

- Create the method `_private_method` that returns the string `"I am a private method"`. Call this method from outside the class.

▼ **Solution**

```
class PrivateClass:
  def __init__(self):
    self._private_attribute = "I am a private attribute"

  def _private_method(self):
    return "I am a private method"

obj = PrivateClass()
print(obj._private_method())
```

Double Underscore

Private Attributes

When you have a single underscore, the Python interpreter does not do anything. It is just a convention. Using two underscores, however, causes the Python interpreter to enforce changes. The specifics of the changes will be discussed below, but Python gives you an approximation of private attributes when using double underscores. We are going to use the same code from the previous page, and add a second underscore before the attribute `private_attribute`.

```
class PrivateClass:
  def __init__(self):
    self.__private_attribute = "I am a private attribute"

obj = PrivateClass()
print(obj.__private_attribute)
```

You should see an error message about `PrivateClass` not having the attribute `__private_attribute`. Python does not allow you to access `__private_attribute` from outside the class. Instead, let's add a helper method to the class that will return `__private_attribute` for us.

```
class PrivateClass:
  def __init__(self):
    self.__private_attribute = "I am a private attribute"

  def helper_method(self):
    return self.__private_attribute

obj = PrivateClass()
print(obj.helper_method())
```

Private Methods

You can also use the double underscores to restrict access to methods.

```
class PrivateClass:
  def __init__(self):
    self.__private_attribute = "I am a private attribute"

  def __private_method(self):
    return "I am a private method"

obj = PrivateClass()
print(obj.__private_method())
```

You should get an error message. Notice, however, that Python gives an attribute error. It says that `PrivateClass` does not have the attribute `__private_method` even though it was defined as a method.

challenge

Try this variation:

- Create the method `helper_method` that calls `__private_method` and call this method outside of the class.

▼ **Solution**

```
class PrivateClass:
  def __init__(self):
    self.__private_attribute = "I am a private attribute"

  def __private_method(self):
    return "I am a private method"

  def helper_method(self):
    return self.__private_method()

obj = PrivateClass()
print(obj.helper_method())
```

Are Double Underscores Really Private?

No. Double underscores were not added to the Python language to promote encapsulation. Rather, the double underscore is used to avoid name collisions in inheritance. Enter the following code into the IDE and click the `TRY IT` button.

```
class PrivateClass:
  def __init__(self):
    self.__private_attribute = "I am a private attribute"

obj = PrivateClass()
print(obj._PrivateClass__private_attribute)
```

When the Python interpreter encounters an attribute with a double underscore, it does not make it private. Instead, it changes the name to `_ClassName__AttributeName`. That is why Python returns and error for `print(obj.__private_attribute)`. `__private_attribute` does not exist. It has been renamed to `_PrivateClass__private_attribute`. This whole process is called name mangling, and it is designed to avoid name collisions in inheritance. Name mangling, however, gives the appearance of private attributes and methods.

Encapsulation Formative Assessment 1

Encapsulation Formative Assessment 2