OOP Principles

This reading introduces you to the OOP principles in more detail using some examples.

The object-oriented paradigm was introduced in the 1960s by Alan Kay. At the time, the paradigm was not the best computing solution given the small scalability of software developed then. As the complexity of software and real-life applications improved, object-oriented principles became a better solution.

You previously encountered the four main pillars of object-oriented programming. These are: encapsulation, polymorphism, inheritance and abstraction. Let's look at a few examples that demonstrate how these principles translate when using Python.

Encapsulation

The idea of encapsulation is to have methods and variables within the bounds of a given unit. In the case of Python, this unit is called a class. And the members of a class become locally bound to that class. These concepts are better understood with scope, such as global scope (which in simple terms is the files I am working with), and local scope (which refers to the method and variables that are 'local' to a class). Encapsulation thus helps in establishing these scopes to some extent.

For example, the Little Lemon company may have different departments such as inventory, marketing and accounts. And you may be required to deal with the data and operations for each of them separately. Classes and objects help in encapsulating and in turn restrict the different functionalities.

Encapsulation is also used for hiding data and its internal representation. The term for this is information hiding. Python has a way to deal with it, but it is better implemented in other programming languages such as Java and C++. Access modifiers represented by keywords such as public, private and protected are used for information hiding. The use of single and double underscores for this purpose in Python is a substitute for this practice. For example, let's examine an example of protected members in Python.

```
class Alpha:

def __init__(self):
      self._a = 2.  # Protected member ‘a’
      self.__b = 2.  # Private member ‘b’
```

self._a is a protected member and can be accessed by the class and its subclasses.

Private members in Python are conventionally used with preceding double underscores: __. self.__b is a private member of the class Alpha and can only be accessed from within the class Alpha.

It should be noted that these private and protected members can still be accessed from outside of the class by using public methods to access them or by a practice known as name mangling. Name mangling is the use of two leading underscores and one trailing underscore, for example:

`_class__identifier`

Class is the name of the class and identifier is the data member that I want to access.

Polymorphism

Polymorphism refers to something that can have many forms. In this case, a given object. Remember that everything in Python is inherently an object, so when I talk about polymorphism, it can be an operator, method or any object of some class. I can illustrate the case for polymorphism using built-in functions and operations, for example:

```
string = "poly"
num = 7
sequence = [1,2,3]
new_str = string * 3
new_num = 7 * 3
new_sequence = sequence * 3

print(new_str, new_num, new_sequence)
```

The output is:

```
polypolypoly 21 [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In the example, I have used the same operator () to perform on a string, integer and a list. You can see the () operator behaves differently in all three cases.

Let's examine one more example.

```
string = "poly"
sequence = [1,2,3]
print(len(string))
print(len(sequence))
```

The output is:

```
4
3
```

The len() function is able to take variable inputs. In the example above it is a string and a list that provides the output in integer format.

Inheritance

Inheritance in Python will be covered later in the course, but the basic template for it is as follows:

```
class Parent:
      Members of the parent class

class Child(Parent):
      Inherited members from parent class
      Additional members of the child class
```

As the structure of inheritance gets more complicated, Python adheres to something called the Method Resolution Order (MRO) that determines the flow of execution. MRO is a set of rules, or an algorithm, that Python uses to implement monotonicity, which refers to the order or sequence in which the interpreter will look for the variables and functions to implement. This also helps in determining the scope of the different members of the given class.

Abstraction

Abstraction can be seen both as a means for hiding important information as well as unnecessary information in a block of code. The core of abstraction in Python is the implementation of something called abstract classes and methods, which can be implemented by inheriting from something called the abc module. "abc" here stands for abstract base class. It is first imported and then used as a parent class for some class that becomes an abstract class. Its simplest implementation can be done as below.

```
from abc import ABC,
class ClassName(ABC):
      pass
```

You will cover abstraction in a little more detail later in the course.