Chapter 15 - Classes & Objects

15 Object-Oriented Programming with Python

Object-oriented Programming (OOP) is a programming paradigm that uses the concept of "objects" that can contain attributes (known as variables) and behaviours (known as methods or functions). We have been exposed to this concept of objects when we looked at Python's standard datatypes. In this section, we will study further in-depth about how objects are constructed, destructed and along with some of the principles that define writing reusable code (in the next chapter).

15.1 Scope

Before we start on the paradigms and principles of OOP, we need to learn the concept of scope. Think of scope as a set of rules that defines the accessibility of the variables and methods within a program or a library or a package. The determination of the scope of a variable or method is determined by the placement of the variable or method within your code.

Python has 4 scope levels called **Local**, **Enclosing**, **Global** and **Built-in** scopes. Think of the scope levels like the layers of an onion, like in figure 1 below.

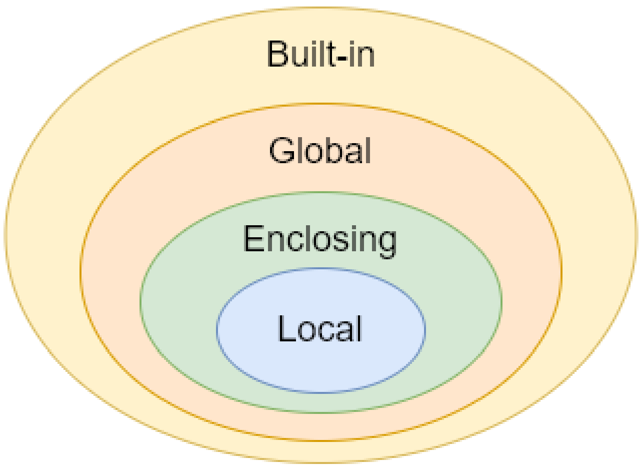

Figure 1: Python's 4 layers of scope.

In order to explain scopes, we will be looking at the inner most layer then moving outwards as we move along. Bear in mind that even as we move towards the outer scope layers, the inner scope layers still applies to the code blocks. Please refresh your knowledge on the indentation rules of Python if you are not familiar with it as it does play a part in the scope levels.

- **Local scope** can be thought of the code block or function that you defined with it's own set of variables. These variables are only visible within the code block or function. This holds true each time a function is called recursively as well.

  ```
  # function
  def update_amount(amt)
      local_to_func = amt * 5  # accessible everywhere within the function

      if <some expression>:
          local_to_if = "abc"  # not accessible outside the if
      print(local_to_if)  # this line will cause an error
  ```

In the above example, the variable `local_to_func` (in line 3) has a local scope to the function `update_amount()` and the variable `local_to_if` (in line 6) has a local scope to the `if` statement block. That means variable `local_to_if` is not accessible outside the `if` statement block but variable `local_to_func` is accessible everywhere within the function. If this function is recursive, the scope of the variables `local_to_func` and `local_to_if` will be duplicated therefore it will not result in collisions with the other variables of the duplicated functions.

- **Enclosing scope** is special in the sense that it is only applicable for nested functions. As we did not learn about nested functions, we will not be covering this scope layer.
- **Global scope** is the top most scope layer of a Python script, program or module. Any variables or methods defined in this scope layer is visible through out the program or module. To use global variables within functions, we need to use the `global` keyword to tell Python that the variable is referenced from outside the function.

```
1  global_var = 500
2  # function
3  def update_amount(amt)
4      global global_var
5      temp = amt * global_var  # accessible everywhere within the function
```

  In the above example, we have a global variable called `global_var` (line 1) and within the `update_amount()` function, we use the `global` keyword (line 4) to "tell" the function that we are going to use the `global_var` variable within it (line 5). Global variables are used sparingly as they can create confusion when we do not know the program flow because global variables can be changed by any code and during anytime of the program execution.

- **Built-in scope** is also another special Python scope that is used to define the scope of built-in modules or imported libraries of Python. The variables names and method names are accessible everywhere within your scripts once loaded. For instance, the built-in `print()` function that can be used without importing any libraries or being able to use the plotting functions of `matplotlib` in our scripts after importing the library.

Now that we have a rough understanding of scope, we can move on to Class and Objects where we will learn more about the scope layers of a class and how to create our own classes and objects.

15.2 Classes and Objects

Think of objects as different parts of a system. A Bank is an object that comprise of many different departments, such as, loans, customer service, operations, etc. Each of those departments have their own set of procedures that transforms into a product that the bank offers to its customers.

These procedures acts like a template to define how an object can be produced. Translating this into programming terms, a class can be defined as the template and the object can be thought of as a product of the class (which is also called an instance of the class). Let's take a loan product as an example. A purpose of a bank loan is to allow customers to borrow a sum of money and return it back over a period of time. This bank loan can have attributes like name of borrower, type of loan, repayment period, etc and objects the loan would be the different loans belonging to the different customers.

15.2.1 Class Members and Behaviours

So how do we create this Loan Class in Python? Lets look at the general syntax of a Class

```
class ClassName:
    "optional class docstring"
    class_suite (consisting of attributes and methods)
```

A Python Class starts with the keyword `class` to indicate that the start of a class being created followed by the name of the class (using CamelCase notation) and the colon `:`. As mentioned earlier, an object contains attributes and behaviours and in turn, a class also contains attributes and behaviours since objects are made from classes. In the case of the bank loan example above, methods that could be in a bank loan class would be `adjust_interest_rates()` or `update_balance_loan_amount()`.

Firstly, lets look at how attributes of a class is defined. There are 2 types of attributes namely class and instance attributes.

- **Instance Attributes** - are attributes that have the scope **only within the object**. Their values are **not shared** nor are they visible to any of the other objects (or instances) made from the same class.
- **Class Attributes** - are attributes that have the same scope **across all objects** made from the same class. When their values change they are visible and accessible across all objects (or instances) from the same class.

```
 1  class BankLoan:
 2      # class attributes
 3      loanCount = 0
 4
 5      # constructor
 6      def __init__(self, borrower_name, loan_type, repayment):
 7          # instance attributes
 8          self.borrower_name = borrower_name
 9          self.loan_type = loan_type
10          self.repayment_period = repayment
11          # modifying class attributes
12          BankLoan.loanCount += 1
```

How do we differentiate a class attribute from a instance attribute in code?

- Class attributes are generally located on the following line **AFTER** the class definition and before the first method (line 3).
- Class attributes also requires the class name to be used in conjunction with the variable name for accessing or manipulating of values (line 12). The general syntax `class_name.variable_name` take note of the dot `.` between them.
- Instance attributes **ALWAYS** have a `self` keyword followed by a dot `.` and the attribute name (lines 8 to 10).
- Initialization (means to be given their default values at the start) of **instance attributes** are normally done in the **constructor** (the function that looks like `__init__()`) (lines 6 to 12) and the **class attributes** are done **immediately** when they are being declared (line 3).

For methods (behaviours)

- Class methods have a declarator `@classmethod` the line before the methods and they accept a `cls` parameter which points to the class (not the object instance). Class methods also cannot modify any instance attributes.
- Instance methods are the "standard" methods with the `self` parameter as the very first parameter in the input parameter list. Instance methods are able to modify class attributes.

Both class and instance methods have a scope layer local to the class if they are declared within a class. For example, the `BankLoan` class could have an instance method called `update_loan_balance()` and a class method called `personal_loan()`.

```python
class BankLoan:
    ...
    def update_loan_balance(self, amt):
        self.loan_amt -= amt

    @classmethod
    def personal_Loan(cls):
        return cls('John Doe', 'Personal', '5 yrs')
```

The purpose of class methods are to provide "factory methods" that indirectly performs a related function but does not make changes to the instance objects. It is also used to define alternative constructors with predefined values for the class.

Take note of the indentation of the attributes and the methods within a class as a `class` is considered **one** code block. Any attributes and/or methods not belonging to the same indentation level, do not belong to that class. They are regarded as *global* and accessible to any functions or classes comprising the program.

15.2.2 Class Creation

To create an instance of a class we use the general syntax

```python
var = className(arguments)
```

where `className` is the name of the class that we are making an instance of and `arguments` are the input data that is used to initialize/define the object. The number of arguments are determined by the number of input arguments defined in the constructor. For example, the `BankLoan` class has 3 arguments defined within the brackets of the constructor like so `def __init__(self, borrower_name, loan_type, repayment)` therefore when initializing an object of the type `BankLoan`, there **must** be 3 pieces of data provided for each object (namely the `borrower_name, loan_type, repayment`). The **only time** when the arguments are optional during object creation is when the arguments in the constructor has default values assigned to them.

```python
class BankLoan:
    # class attributes
    loanCount = 0

    # constructor
    def __init__(self, borrower_name, loan_type, repayment):
        # instance attributes
        self.borrower_name = borrower_name
        self.loan_type = loan_type
        self.repayment_period = repayment
        # modifying class attributes
        BankLoan.loanCount += 1
    ...

# creating objects
bl1 = BankLoan('Sam', 'housing', 25)
bl2 = BankLoan('Belle', 'study', 10)
```

To access a function from a class, we use the same dot `.` notation as mentioned above when accessing or manipulating class attributes. The syntax is similar `class_Name.function_name()`. For instance, if we would like to call the function `update_loan_amount()`, it would be called as follows

```
bl1.update_loan_amount(58)
```

15.2.3 Constructors and Destructors

With every form of creation there must be some way to destroy it or clean it up after we are done using it. With Python classes, we have something called a **destructor**. Destructors are special methods used to clean up the object after we are finished with using them. Earlier we have already seen what constructors are used for and now we will see how a destructor works. The general syntax for a destructor is as follows

```
def __del__(self):
    statement(s)
```

Destructors are called when the `del` keyword is used to delete an object or at the end when the program finishes executing. Remember that we can use the `del` keyword to remove elements from lists, tuples and dictionaries in the earlier chapters.

Example with `del` keyword

```
class Loan:
    # Initializing
    def __init__(self):
        print('Loan created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Loan deleted.')

obj = Loan()
del obj

```

the output is

```
Loan created.
Destructor called, Loan deleted.
```

Example of the destructor being called **after** the program ends

```python
class Loan:
    # Initializing
    def __init__(self):
        print('Loan created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Loan deleted.')

def Create_obj():
    print('Making Object...')
    obj = Loan()
    print('function end...')
    return obj

print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')

```

the output is

```
Calling Create_obj() function...
Making Object...
Loan created.
function end...
Program End...
Destructor called, Loan deleted.
```

Note that with Jupyter Notebook, the last statement `Destructor called, Loan deleted.` is not shown as the Python interpreter for the Notebook kernel has not exited interactive mode. To see the last statement, create a Python script with the codes and use the Command Prompt to execute this Python script.

A pitfall of having multiple classes with custom destructors is when there exist a **circular reference** between classes.

```python
class A:
    def __init__(self, bb):
        self.b = bb

class B:
    def __init__(self):
        # this line creates a instance of class A and it
        # passes a reference of class B (itself) to A
        self.a = A(self)
    def __del__(self):
        print("cleaned up")

def fun():
    b = B()

fun()
```

If codes above are executed using Command Prompt, it will still exit normally and objects `a` and `b` will still be cleaned up upon program exiting **but** if this code were to be executed in Jupyter Notebook, Python's garbage collector (that clears away all objects flagged for deletion with the `del` keyword during execution) will get confused as the custom destructor makes both objects "uncollectable" thus they will live in memory for as long as the Notebook kernel is running. The reason being that there is a circular dependency between both objects that means both objects relies on each other therefore we cannot delete one without deleting the other as well.

Note that Jupyter Notebook will **not produce** any output if you were to execute the codes above with it because of the circular dependency. You will need to Restart the kernel so that there are no more "uncollectable" objects floating in memory (also called memory leaks).

15.3 References

1. Amos, July 2020, Object-Oriented Programming (OOP) in Python 3, https://realpython.com/python3-object-oriented-programming/
2. Tagliaferri, March 2017, How To Construct Classes and Define Objects in Python 3, https://www.digitalocean.com/community/tutorials/how-to-construct-classes-and-define-objects-in-python-3
3. Sergei, Feb 2018, Circular references without memory leaks and destruction of objects in Python, https://medium.com/@chipiga86/circular-references-without-memory-leaks-and-destruction-of-objects-in-python-43da57915b8d
4. Ramos, March 2020, Python Scope & the LEGB Rule: Resolving Names in Your Code, https://realpython.com/python-scope-legb-rule/