Discussion-8 Classes

Discussion Topic:

Briefly describe the difference between a class and an instance of a class. In a Python class, how do you hide an attribute from code outside the class? Provide code examples demonstrating the usage of classes in Python. In response to your peers, provide additional code examples as well as other class differences to build upon the peer postings.

My Post:

Hello Class,

In programming, more specifically in object-oriented languages or object-oriented programming, classes can be defined as blueprints or templates for creating objects. They encapsulate data and behavior (code) into a single entity (Ramos, 2024). While objects can be defined as the instances created from those classes.
In other words, classes contain the code that defines attributes and behaviors shared by all objects of that class type, and objects are the instantiation of specific objects created from a class, possessing their own unique data, but sharing the same behavior (functionality).

In C++ and Java, hidden class attributes are called private fields; they are declared as such, meaning that these attributes cannot be accessed directly from outside a class's scope.
On the other hand, Python doesn't have true private field declarations it does not have truly private attributes. It relies on convention and name-mangling.

- By convention, a single underscore at the start of a variable name (e.g., `_internal_attr`), designates a variable attribute as an interval variable, signaling to other developers not to access it directly outside of the class scope, even though it is technically accessible (e.g., `object._internal_attr`)
- Name-mangling, double underscore at the start of a variable name (e.g., `__mangled_attr`), it triggers Python automatic name mangling (e.g., `_ClassName__mangled_attr`), it concatenates a leading underscore and the class name with the attribute.
  This makes it harder to access the attribute variable, as it hides the attribute, but it is still possible to accesses it (`e.g, object._ClassName__mangled_attr`).

In other words, C++ and Java private class attributes can not be accessed directly from outside a class's scope. On the other hand, Python hides private class attributes; these attributes can be accessed directly from outside the class's scope by using their names (If you know them).
See code below for a more in-depth illustration:

```
# Defining the class
class MyClass:
    _internal_attr = "Internal Attribute"
    __mangled_attr = "Mangled Attribute"

# Creating an instance of the class
my_class = MyClass

# Using Python built-in “as attribute” function
print(hasattr(my_class, "_internal_attr"))              # --> True
print(hasattr(my_class, "MyClass__mangled_attr"))       # --> False

# Using Python built-in “get attribute” function
print(getattr(my_class, "_internal_attr"))              # --> Internal Attribute
try:
    print(getattr(my_class, "MyClass__mangled_attr"))
except AttributeError:
    print(AttributeError)                               # --> <class 'AttributeError'>

# Using Python built-in “print” function
print(my_class._internal_attr)                         # --> Internal Attribute
print(my_class._MyClass__mangled_attr)                 # --> Mangled Attribute
```

Another important functionality of Python is the use of decorators, such as `@property`, which are descriptors under the hood. Python uses three method decorators.

- Getter --> `@property`
- Setter --> `@setter_name.setter`
- Delete --> `@deleter_name.deleter`

This decorator allows for exposing methods as if they were a simple attribute.
It is also a good way to hide class attributes in a function wrapper.

Below is an example of demonstrating the usage of classes in Python.

```
class Account:
    def __init__(self, balance: float):
        # could also be __balance for name-mangling
        self._balance = float(balance)

    # --- Getter
    @property
```

```
    def balance(self) -> float: # public property (getter)
        return self._balance

    # --- Setter
    @balance.setter
    def balance(self, value: float) -> None:
        if value < 0:
            raise ValueError("Balance cannot be negative.")
        self._balance = float(value)

# Creating an account object
acct = Account(100)
print(acct.balance)   # --> 100.0 (calls getter)
acct.balance = 250    # setter --> setter
print(acct.balance)   # getter --> 250.0
```

-Alex

References:

Ramos, L., P. (2024, February 20245). *Python classes: The power of object-oriented programming*. Real Python. https://realpython.com/python-classes/