Object Oriented Programming in Python

Terminology

I want to start with terminolgic words which uses in OOP in Python.

- Class
  - A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- Class Variable
  - A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- Data Member
  - A class variable or instance variable that holds data associated with a class and its objects.
- Function Overloading
  - The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- Instance Variable
  - A variable that is defined inside a method and belongs only to the current instance of a class.
- Inheritance
  - It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- Encapsualtion
  - Packing of data and functions operating on that data into a single component and restricting the access to some of the object's components. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.
- Instance
  - An individual object of a certain class. An object obj that belongs to a class Person, for example, is an instance of the class Person.
- Instantiation
  - The creation of an instance of a class.
- Method
  - A special kind of function that is defined in a class definition.
- Object
  - A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- Operator Overloading
  - The assignment of more than one function to a particular operator.
- Constructors

- Constructors are generally used for instantiating an object.The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.In Python the *init*() method is called the constructor and is always called when an object is created.

- Destructors Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.
- Context Manager
  - That allows us to allocate and release resources precisely when we want to. The most widely used example of context manager is the "with" statement.
- Generator Functions
  - Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).
- Garbage Collector
  - The garbage collector is keeping track of all objects in memory. A new object starts its life in the first generation of the garbage collector. If Python executes a garbage collection process on a generation and an object survives, it moves up into a second, older generation.

Classes

Define A Class

```
class Person():

    def __init__(self, name, age, sex):
        self.name = name
        self.age = age
        self.sex = sex


first_person = Person("Selami", 77, "Male")

print(first_person.name)
print(first_person.age)
print(first_person.sex)
```

Output

```
Selami
77
Male
```

- The first method *init*() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

Define A Class with Default Paremeters

```
class Person():

    def __init__(self, name="Regen KID", age=23, sex=None):
        self.name = name
        self.age = age
        self.sex = sex


person_one = Person()
person_two = Person(name="Debbie Harry", age=75, sex="Female")

print(f"person_one's Name: {person_one.name}, Age: {person_one.age}, Sex: {person_one.sex}")
print(f"person_two's Name: {person_two.name}, Age: {person_two.age}, Sex: {person_two.sex}")
```

Output

```
person_one's Name: Regen KID, Age: 23, Sex: None
person_two's Name: Debbie Harry, Age: 75, Sex: Female
```

- Classes can have their own functions (a.k.a 'METHOD').

Class Methods

```
import random

class Person():

    def __init__(self, name="Regen KID", age=23, sex=None):
        self.name = name
        self.age = age
        self.sex = sex
        self.location = "Turkey"

    def increase_age(self):
        self.age = self.age + 1

    def create_bank_account(self):
        self.account_id = random.randint(10000000, 99999999)

    def get_account_id(self):
        return self.account_id

    def move_to_another_country(self, new_country:str):
        self.location = new_country

person_two = Person(name="Debbie Harry", age=75)

print(f"person_two's Name: {person_two.name}, Age: {person_two.age}, Location: {person_two.location}")
person_two.create_bank_account()
print(f"{person_two.name}'s bank acoount id is: {person_two.get_account_id()}")
```

Output

```
person_two's Name: Debbie Harry, Age: 75, Location: Turkey
Debbie Harry's bank acoount id is: 37652487
```

Encapsulation

- We can reach and change the objects variables. Sometetimes We may prohibit the access from out of scope (for security etc.) to functions or variables. So with using "encapsulation", we can prevent undesired usages.
- Most programming languages has three forms of access modifiers, which are Public, Protected and Private in a class.
- In python, we define functions or variables as these three of access modifiers with underscope (_).
  - method_name or variable_name → Public
  - _method_name or _variable_name → Protected (One Underscope)
  - __method_name or __variable_name → Private (Double Underscope)

Encapsulation of Object's variables

```
class Person():

    def __init__(self, name, age=23, sex=None):
        self.__name = name
        self.age = age
        self.sex = sex

    def get_name(self):
        return self.__name


person_two = Person(name="Debbie Harry", age=75, sex="Female")
print(f"person_two's Name: {person_two.get_name()}, Age: {person_two.age}, Sex: {person_two.sex}")
print(f"person_two's Name: {person_two.name()}, Age: {person_two.age}, Sex: {person_two.sex}")
```

Output

```
person_two's Name: Debbie Harry, Age: 75, Sex: Female
Traceback (most recent call last):
  File "C:\Users\user\Projects\python_oop\encapsulation_1.py", line 14, in <module>
    print(f"person_two's Name: {person_two.name()}, Age: {person_two.age}, Sex: {person_two.sex}")
AttributeError: 'Person' object has no attribute 'name'
```

- We couldn't get name value from out of class scope because name is private value, so we wrote a getter method which is public function to reach this variable.

Protected Methods and Variables

```
class Person():

    _name = None
    _age = None
    _location = None

    def __init__(self, name, age=27, location="Turkey"):
        """ This is constructor method. """
        self._name = name
        self._age = age
        self._location = location

    def _show_ages_and_locations(self):
        """ Protected Function. We Can access to protected data members"""

        print("Age: ", self._age)
        print("Location: ", self._location)

class Developer(Person):

    def __init__(self, name, age, location, company):
        super().__init__(name, age, location)
        self.company = company

    def show_developer_info(self):
        """ We create public method and we can access to protected data
        members and methods of super class in here. """

        print("Developer's Name: ", self._name)
        self._show_ages_and_locations()


dev_one = Developer("Dennis Ritchie", 70, "USA", "Lucent Technologies")
dev_one.show_developer_info()
```

Output

```
Developer's Name:  Dennis Ritchie
Age:  70
Location:  USA
```

- The members of a class that are declared protected are only accessible to a class derived from it

Private Method and Variables

```
class Person():

    def __init__(self, name, age=27, location="Turkey"):

        self.__name = name
        self.__age = age
        self.__location = location

    def __print_person_details(self):
        """ Private Method. We can use this method only in this class. """

        print(f"Person Details: {self.__name} {self.__age} {self.__location}")

    def access_from_outside(self):
        """
        We cannot access to public method in any shape or form. So We use the public method to
        access private/protected methods/variables.
        We can use private method here because we are in it's class scope.
        """
        self.__print_person_details()


person_one = Person("Debbie Harry", 75, "USA")
person_one.access_from_outside()
person_one.__print_person_details()
```

Output

```
Person Details: Debbie Harry 75 USA
Traceback (most recent call last):
  File "C:\Users\user\Projects\python_oop\encapsulation_3.py", line 25, in <module>
    person_one.__print_person_details()
AttributeError: 'Person' object has no attribute '__print_person_details'
```

- In Java or e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.
- Python offers a solution to this problem. The solution is called properties!
- Note: The actual money value is stored in the private _money variable. The money attribute is a property object which provides an interface to this private variable.

Properties Getter Setter

```
class Person:
    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
        # with adding underscope to salary on the left side, we can observe class behaviour (self._salary)

    @property
    def salary(self):
        print("Get Salary Method Running ... ")
        return self._salary

    @salary.setter
    def salary(self, value):
        print("Set Salary Method Running ... ")
        if value > 15000:
            print("You're RICH! You should pay taxes")
            self._salary = (value * 0.6)
        else:
            print("You're POOR! You should pay more taxes")
        self._salary = (value * 0.4)


person_one = Person("Debbie Harry", 6666)
print(f"{person_one.name}'s real salary is: {person_one.salary}")
person_one.salary = 4584000
print(f"{person_one.name}'s real salary is: {person_one.salary}")
```

Output

```
Set Salary Method Running ...
You're POOR! You should pay more taxes
Get Salary Method Running ...
Debbie Harry's real salary is: 2666.4
Set Salary Method Running ...
You're RICH! You should pay taxes
Get Salary Method Running ...
Debbie Harry's real salary is: 1833600.0
```

- We used directly variable property to modify _{variable_name} attribute. However, it is still accessible in Python. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

Inheritance

- Inheritance models what is called an is a relationship. This means that when you have a Derived class that inherits from a Base class, you created a relationship where Derived is a specialized version of Base.
  - Classes that inherit from another are called derived classes, subclasses, or subtypes.
  - Classes from which other classes are derived are called base classes or super classes.
  - A derived class is said to derive, inherit, or extend a base class.
  - Overriding: Method overriding is thus a part of the inheritance mechanism. In Python method overriding occurs by simply defining in the child class a method with the same name of a method in the parent class.
    - So We must define *init* (constructor) again for child classess.
  - Interface: For an object, is a set of methods and attributes on that object.
  - In Python, we can use an abstract base class to define and enforce an interface.
    - At a high level, an interface acts as a blueprint for designing classes. Like classes, interfaces define methods. Unlike classes, these methods are abstract. An abstract method is one that the interface simply defines. It doesn't implement the methods. This is done by classes, which then implement the interface and give concrete meaning to the interface's abstract methods.
    - Python's approach to interface design is somewhat different when compared to languages like Java, Go, and C++. These languages all have an interface keyword, while Python does not. Python further deviates from other languages in one other aspect. It doesn't require the class that's implementing the interface to define all of the interface's abstract methods.
  - This is known as the Liskov substitution principle. The principle states that "in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desired properties of the program".

Using Inheritence and Overriding

```
class Person():

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def get_real_age(self):
        return self.age


class Musician(Person):

    def __init__(self, name: str, age: int, genre: str, band: str, instrument=None):
        super().__init__(name, age)
        self.genre = genre
        self.band = band
        self.insturment = instrument


person_1 = Person("John Doe", 33)
print(f"person_1's age: {person_1.age} and real age: {person_1.get_real_age()}")
musician_1 = Musician("Debbie Harry", 75, "New Wave/Punk", "Blondie")
print(f"musician_1's age: {musician_1.age} and real age: {musician_1.get_real_age()}")


class Developer(Person):

    def __init__(self, name: str, age: int, company: str, location: str):
        super().__init__(name, age)
        self.company = company
        self.location = location

    def get_real_age(self):
        return self.age * 1.5


developer_1 = Developer("Dennis Ritchie", 70, "Lucent Technologies", "USA")
developer_1.get_real_age()
print(f"developer_1's age: {developer_1.age} and real age: {developer_1.get_real_age()}")
```

Output

```
person_1's age: 33 and real age: 33
musician_1's age: 75 and real age: 75
developer_1's age: 70 and real age: 105.0
```

- Abstract Base Classes
  - With Using Abstract Class;
    - Telling users of the module that objects of type Base Class can’t be created.
    - Telling other developers working on the same module that if they derive from Base Class, then they must override the abstract method.

Using Abstract Classes for Inherit A Function

```
from abc import ABC, abstractmethod


class Person(ABC):

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    @abstractmethod
    def calculate_happiness():
        pass


class Developer(Person):

    def __init__(self, name: str, age: int, company: str, location: str):
        super().__init__(name, age)
        self.company = company
        self.location = location

    def get_real_age(self):
        return self.age * 1.5

    def calculate_happiness(self):
        country_happiness = {
            "USA": 10,
            "Turkey": 9,
            "UK": 11,
            "Canada": 12,
        }
        hapiness = country_happiness[self.location] / self.age
        return hapiness

developer_1 = Developer("Dennis Ritchie", 70, "Lucent Technologies", "USA")
print(developer_1.calculate_happiness())
person_1 = Person("John Doe", 123)
person_1.calculate_happiness()
```

Output

```
0.14285714285714285
Traceback (most recent call last):
  File "C:\Users\user\Projects\advanced-python\inheritance_2.py", line 37, in <module>
    person_1 = Person("John Doe", 123)
TypeError: Can't instantiate abstract class Person with abstract method calculate_happiness
```

- We accessed to inherited class but we couldn’t accessed to abstract class and method.

Multiple Inheritence: When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

Multiple Inheritence

```
class Person():

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def increase_age(self):
        self.age = self.age + 1

    def decrease_age(self):
        self.age = self.age - 1

    def get_name(self):
        print("Getting Person Name ....")
        print(self.name)


class Developer():

    def __init__(self, name:str, nickname:"NEO", company=None, location="Turkey"):
        self.name = name
        self.nickname = nickname
        self.company = company
        self.location = location

    def move_to_another_country(self, new_location):
        self.location = new_location

    def change_nickname(self, new_nickname):
        print("You cannot change your nickname")

    def get_name(self):
        print("Developer Name")
        return self.name


class DevPerson(Person, Developer):

    def __init__(self, name, age, nickname, company, location):
        self.name = name
        self.age = age
        self.nickname = nickname
        self.company = company
        self.location = location

    def change_nickname(self, new_nickname):
        """ Overriding :D """
        self.nickname = new_nickname

person_1 = Person("Ada Lovelace", 206)
developer_1 = Developer("John Anderson", "Neo", "SomeWhereCoop", "USA")
person_1.increase_age()
person_1.decrease_age()
developer_1.change_nickname("NeoOfZion")
print(f"Person ONE: {person_1.name}, {person_1.age}")
print(f"Developer ONE: {developer_1.name}, {developer_1.nickname}, {developer_1.company}, {developer_1.location}")

hybrid_1 = DevPerson("Fiona Apple", 45, "Fiona", "SomeWhereCoop", "USA")
print(f"Hybrid ONE: {hybrid_1.name}, {hybrid_1.age}, {hybrid_1.nickname}, {hybrid_1.company}, {hybrid_1.location}")
hybrid_1.get_name()
# We get the Person Class's get_name function because while we define hybrid class, we defined Person before Developer.
hybrid_1.increase_age()
hybrid_1.change_nickname("Hot Knife")
hybrid_1.increase_age()
hybrid_1.move_to_another_country("Turkey")
hybrid_1.decrease_age()
print(f"Hybrid ONE: {hybrid_1.name}, {hybrid_1.age}, {hybrid_1.nickname}, {hybrid_1.company}, {hybrid_1.location}")
```

Output

```
You cannot change your nickname
Person ONE: Ada Lovelace, 206
Developer ONE: John Anderson, Neo, SomeWhereCoop, USA
Hybrid ONE: Fiona Apple, 45, Fiona, SomeWhereCoop, USA
Getting Person Name ....
Fiona Apple
Hybrid ONE: Fiona Apple, 46, Hot Knife, SomeWhereCoop, Turkey
```

- Diamond Problem:

- It refers to an ambiguity that arises when two classes Class2 and Class3 inherit from a superclass Class1 and class Class4 inherits from both Class2 and Class3. If there is a method “m” which is an overridden method in one of Class2 and Class3 or both then the ambiguity arises which of the method “m” Class4 should inherit.