Python Notes

Python Class or Static Attributes

1. What is Python class attributes?
   - All variables which are assigned a value in class declaration are class variables.
   - And variables which are assigned values inside method are instance variables.
   - For class variables, all objects a single copy maintained at class level.
   - There is no difference between python class and static variables, both are same.

2. How can we able to create and access class attributes?
   - Inside Class
     a. Inside class directly
     b. Inside constructor by using class name
     c. Inside instance method by using class name
     d. Inside class method by using cls variable or class name
     e. Inside static method by using class name
   - Outside Class
     a. From outside of class by using class name

```
class StaticDemo:
    # 1. Inside class directly
    a = 10

    # 2. Inside constructor by using class name
    def __init__(self):
        StaticDemo.b = 20

    # 3. Inside instance method by using class name
    def m1(self):
        StaticDemo.c = 30

    # 4. Inside class method by using cls variable or class name
    @classmethod
    def m2(cls):
        # 4a. using cls variable
        cls.d = 40
        # 4b. Using class name
        StaticDemo.e = 50

    # 5. Inside static method by using class name
    @staticmethod
    def m3():
        StaticDemo.f = 60


# 6. From outside of class by using class name
StaticDemo.g = 70
```

```
object_1 = StaticDemo()
object_1.m1()
# StaticDemo.m2()
object_1.m2()
# StaticDemo.m3()
object_1.m3()
print('Class Level  :', StaticDemo.__dict__)
print('*' * 50)
print('Object Level :', object_1.__dict__)
```

3. How can we able to modify class attributes?
   - Inside Class
     a. Inside Constructor using Class name
     b. Inside Class method using Class name or cls variable
     c. Inside Static method using Class name
   - Outside Class
     a. Outside of class by using class name

```
class StaticDemo:
    a = 10

    # 1. Inside Constructor using Class name
    def __init__(self):
        StaticDemo.a = 100
        # Important topic
        self.a = 101

    # 2. Inside Class method using Class name or cls variable
    @classmethod
    def m1(cls):
        # 2a. using class name
        StaticDemo.a = 200
        # 2b. using cls variable
        cls.a = 300

    # 3. Inside Static method using Class name
    @staticmethod
    def m2():
        StaticDemo.a = 400


object_1 = StaticDemo()
# StaticDemo.m1()
object_1.m1()
# StaticDemo.m2()
object_1.m2()

# 4. Outside of class by using class name
```

```
StaticDemo.a = 500
print('Class Level :', StaticDemo.__dict__)
print('Object Level:', object_1.__dict__)
```

4. How can we able to delete class attributes?
   - Inside Class
     a. Inside Constructor using Class name
     b. Inside Class method using Class name or cls variable
     c. Inside Static method using Class name
   - Outside Class
     a. Outside of class by using class name

```
class StaticDemo:
    a = 10
    b = 20
    c = 30
    d = 40
    e = 50
    f = 60

    # 1. Inside Constructor using Class name
    def __init__(self):
        del StaticDemo.a

    # 2. Inside Class method using Class name or cls variable
    @classmethod
    def m1(cls):
        # 2a. Using class name
        del StaticDemo.b
        # 2b. Using cls variable
        del cls.c

    # 3. Inside Static method using Class name
    @staticmethod
    def m2():
        del StaticDemo.d


print('Before object creation :', StaticDemo.__dict__)
object_1 = StaticDemo()
print('After object creation :', StaticDemo.__dict__)
StaticDemo.m1()
print('After calling m1() method :', StaticDemo.__dict__)
StaticDemo.m2()
print('After calling m2() method :', StaticDemo.__dict__)

# 4. Outside of class by using class name
del StaticDemo.e
print('Outside class del :', StaticDemo.__dict__)
```

Getters and Setters

1. What is Python Getters and Setters and Property?
   - Getters are used for retrieving the data. Also known as 'Accessors'.
   - Setters are used for changing the data. Also known as 'Mutators'.
   - Property is Pythonic way to implement Getters and Setters.

2. Why do we need to use Getters and Setters?
   - The main purpose of getters and setters are Data Encapsulation. To avoid direct access of variables.
   - Adding validation logic while setting and getting variables.

3. How to implement Getters and Setters?
   I. Using normal functions / methods.

```
# 1.    Using normal functions / methods
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10

    # Getter method
    def get_a(self):
        return self.__a

    # Setter method
    def set_a(self, a):
        self.__a = a


object_1 = DataEncapsulation()
print('Get value of a:', object_1.get_a())
object_1.set_a(20)
print('Get value of a:', object_1.get_a())
```

4. How to implement Property?
   I. Using python Property() function

A property has 3 methods,
getter(),
setter() and
delete().

And has four arguments property(fget=None, fset=None, fdel=None, doc=None),
fget is a function for retrieving an attribute value,

fset is a function for setting an attribute value.
fdel is a function for deleting an attribute value.
doc creates a docstring for attribute.

```
# 2.    Python property() function
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10
        self.__b = 20

    # Getter method
    def get_a(self):
        print('Inside get_a()')
        return self.__a

    # Setter method
    def set_a(self, a):
        print('Inside set_a()')
        if a <= 0:
            print('Negative value is set to default value.')
            self.__a = 1
        else:
            self.__a = a

    def del_a(self):
        print('Inside del_a()')
        del self.__a

    a = property(get_a, set_a, del_a, 'Property function demo')


object_1 = DataEncapsulation()
print(object_1.__dict__)
print('Get value of a:', object_1.a)
object_1.a = -20
print('Get value of a:', object_1.a)
del object_1.a
print(object_1.__dict__)
```

II. Using @property decorators

Python @property is one of the built-in decorators. The main purpose of any decorator is to change your class methods or attributes in such a way so that the user of your class no need to make any change in their code.

```python
# 3.    Using @property decorators
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10

    # Getter method
    @property
    def a(self):
        print('Inside Getter method')
        return self.__a

    # Setter method
    @a.setter
    def a(self, a):
        print('Inside Setter method')
        if a <= 0:
            print('Negative value is set to default value.')
            self.__a = 1
        else:
            self.__a = a

    @a.deleter
    def a(self):
        print('Inside Deleter method')
        del self.__a


object_1 = DataEncapsulation()
print(object_1.__dict__)
print('Get value of a:', object_1.a)
object_1.a = -20
print('Get value of a:', object_1.a)
del object_1.a
print(object_1.__dict__)
```

Comparing Getters and Setters v/s Property() function and Property decorators

```python
# 1.    Using normal functions / methods
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10
        self.__b = 20
```

```
    # Getter method
    def get_a(self):
        return self.__a

    # Setter method
    def set_a(self, a):
        self.__a = a

    # Getter method
    def get_b(self):
        return self.__b

    # Setter method
    def set_b(self, b):
        self.__b = b


object_1 = DataEncapsulation()

# Perform addition
object_1.set_b(object_1.get_b() + object_1.get_a())
print(object_1.get_b())


# 2.    Python property() function
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10
        self.__b = 20

    # Getter method
    def get_a(self):
        print('Inside get_a()')
        return self.__a

    # Setter method
    def set_a(self, a):
        print('Inside set_a()')
        if a <= 0:
            print('Negative value is set to default value.')
            self.__a = 1
        else:
            self.__a = a

    def del_a(self):
        print('Inside del_a()')
```

```
        del self.__a

    a = property(get_a, set_a, del_a, 'Property function demo')

    # Getter method
    def get_b(self):
        print('Inside get_b()')
        return self.__b

    # Setter method
    def set_b(self, b):
        print('Inside set_b()')
        if b <= 0:
            print('Negative value is set to default value.')
            self.__b = 1
        else:
            self.__b = b

    def del_b(self):
        print('Inside del_b()')
        del self.__b

    b = property(get_b, set_b, del_b, 'Property function demo')


object_1 = DataEncapsulation()

# Perform addition
object_1.b = object_1.a + object_1.b
print(object_1.b)


# 3.    Using @property decorators
class DataEncapsulation:

    def __init__(self):
        # private variable
        self.__a = 10
        self.__b = 20

    # Getter method
    @property
    def a(self):
        print('Inside Getter method')
        return self.__a

    # Setter method
    @a.setter
```

```
    def a(self, a):
        print('Inside Setter method')
        if a <= 0:
            print('Negative value is set to default value.')
            self.__a = 1
        else:
            self.__a = a

    @a.deleter
    def a(self):
        print('Inside Deleter method')
        del self.__a

    # Getter method
    @property
    def b(self):
        print('Inside Getter method')
        return self.__b

    # Setter method
    @a.setter
    def b(self, b):
        print('Inside Setter method')
        if b <= 0:
            print('Negative value is set to default value.')
            self.__b = 1
        else:
            self.__b = b

    @a.deleter
    def b(self):
        print('Inside Deleter method')
        del self.__b


object_1 = DataEncapsulation()

# Perform addition
object_1.b = object_1.a + object_1.b
print(object_1.b)
```

The attributes of a class are made private to hide and protect them from other code.
Note: When you create private attributes the set getters and setters to public