Encapsulation, Abstraction and methods

Building of data and methods that act on that data into a single unit(class) - Encapsulation
the interface should be independent of the implementation.It also prevents repetiton - Abstraction

Public Vs Private attributes

Public can be accessed anywhere
Private cant be accessed outside the class

protected and private are done through convention as there is so such thing in python
_<attribute> (protected)
__<attribute> (private)
This would mean that these attributes shouldnt be accessed outside the class

```
class Car:

    seats = 4

    def __init__(self, model, year, id_num, serial_num):
        self.model = model //public
        self.year = year //public
        self._id_num = id_num //protected
        self.__serial_num = serial_num //private
```

In this case model and year are public and id and serial are non public attributes

```
class Cars:

    seats = 4

    def __init__(self, model, year, id_num, serial_num):
        self.model = model
        self.year = year
        self._id_num = id_num
        self.__serial_num = serial_num

car1 = Cars("teska", 2006, 2010, "546876136476")
print(car1._id_num)
print(car1.__serial_num)
```

OUTPUT

```
2010
    print(car1.__serial_num)
AttributeError: 'Cars' object has no attribute '__serial_num'
```

No attributre is completely private in python hence we use the term non public
In this case we can access id_num eventhough we shouldnt

accessing __serial_num will give an error

```
AttributeError: 'Cars' object has no attribute '__serial_num'
```

- **To indicate to other developers that an attribute should be "protected", we add a leading underscore (_) before the name of the attribute.**
- **Adding two leading underscores (__) before the name of an attribute will trigger a process called "name mangling" and an error will be thrown if you try to access the attribute directly.**

to access it we can do

```
print(car1._Cars__serial_num)
```

hence you can access them technicallly but you shouldnt

IN PYTHON THERE ARE NO ACCESS MODIFIERS LIKE JAVA

For more info

NAME MANGLING

It helps to prevent name clashes , when we add '__' it triggers name mangling.Hence when python encounters for example __engine in the class Car it converts the attribute to _Car__engine

Name mangling is triggered in the following names

- __<attribute>
- __<arttribute>_

__<attribute>__ dosent not trigger name hangling as it is reserved for magic methods.

GETTERS AND SETTERS

- Members of a class (Methods)
- Their purpose is to get and set the value of an instance attribute

ex :

```
class Dog:

    def __init__(self, name):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, name):
        if isinstance(name, str):
            self._name = name
        else :
            print("enter a valid name")



dog1 = Dog("snoopy")
print(dog1.get_name())
dog1.set_name("bruno")
print(dog1.get_name())
dog1.set_name(1234)
```

```
dog1.set_name(1234)
```

OUTPUT

```
snoopy
bruno
enter a valid name
```

we shouldnt access them directly as they are non public attributes

a example:

```
#--- example of getter and setter---#

class Patient:

    def __init__(self, name, age, id_num, num_children=0):
        self.name = name
        self.age = age
        self._id_num = id_num #protected hence should not be acceses from outside
        self._num_children = num_children #protected hence should not be acceses from outside

    def get_id_num(self):

        return self._id_num;

    def get_num_children(self):

        return self._num_children

    def set_id_num(self, id):

        if isinstance(id,str):
            self._id_num = id

        else:
            print("not a valid id")

    def set_num_children(self, numchild):

        if isinstance(numchild, int) and (0 < numchild < 70):
            self._num_children = numchild

        else:
            print("not a valid number of children")

patient2 = Patient("ram",24,"45672",2)
print(patient2.get_id_num())
print(patient2.get_num_children())
patient2.set_id_num(456)
patient2.set_num_children(-2)
```

OUTPUT

```
45672
2
not a valid id
not a valid number of children
```

PROPERITES

In python getters and setters methods are not usually used instead we use properites ,as if we have to change a non publiv attribute to public attribute it will cause a lot of changes hence we use properties to access it without the underscore

```
variable = property(getter,setter)
```

example

```
class Patient:

    def __init__(self, name, age, id_num, num_children=0):
        self.name = name
        self.age = age
        self._id_num = id_num #protected hence should not be acceses from outside
        self._num_children = num_children #protected hence should not be acceses from outside

    def get_id_num(self):
        print("getter")
        return self._id_num;

    def set_id_num(self, id):
        print("setter")
        if isinstance(id,str):
            self._id_num = id

        else:
            print("not a valid id")


    id_num = property(get_id_num, set_id_num)

    def get_num_children(self):

        return self._num_children

    def set_num_children(self, numchild):

        if isinstance(numchild, int) and (0 < numchild < 70):
            self._num_children = numchild

        else:
            print("not a valid number of children")

    num_children = property(get_num_children,set_num_children)

patient1 = Patient("ram",24,"45672",2)
print(patient1.id_num) # we are actually calling the getter method
patient1.id_num = "8456" # we are actually calling the setter method
```

OUTPUT

```
getter
45672
```

```
setter
```

The first argument passed to property() should be the name of the getter and the second argument should be the name of the setter.

@property decorator

- more compact
- More readable
- avoid calling property directly
- no need to call get_attr and set_attr seperately

@property - will make it a getter
@<attribute>.setter - will make it setter

```
class Patient:

    def __init__(self, name, age, id_num, num_children=0):
        self.name = name
        self.age = age
        self._id_num = id_num #protected hence should not be acceses from outside
        self._num_children = num_children #protected hence should not be acceses from outside

    @property
    def id_num(self):
        print("getter")
        return self._id_num;

    @id_num.setter
    def id_num(self, id):
        print("setter")
        if isinstance(id,str):
            self._id_num = id

        else:
            print("not a valid id")

    @property
    def num_children(self):
        print("getter")
        return self._num_children

    @num_children.setter
    def num_children(self, numchild):

        if isinstance(numchild, int) and (0 < numchild < 70):
            self._num_children = numchild

        else:
            print("not a valid number of children")
```

```
patient1 = Patient("ram",24,"45672",2)
print(patient1.id_num) # we are actually calling the getter method
patient1.id_num = "8456" # we are actually calling the setter method
```

OUTPUT

```
getter
45672
setter
```

What is a decorator?

A **decorator** is a function that takes a function as argument to extend its functionality without actually modifying it.

This is the typical syntax of a decorator function:

```
def decorator_function(arg_function):
        def wrapper_function():
                # Code to extend the functionality
                arg_function()
                # Code to extend the functionality
        return wrapper_function
```

@<property>.deleter

This method is called when you want to **delete** the attribute.

```
classBus:
    def __init__(self, color):
        self._color = color
    @property
    def color(self):
        returns elf._color
    @color.setter
    def color(self, new_color):
        self._color = new_color
    @color.deleter
    def color(self):
        del self._color
```

example

```
@id_num.deleter
    def id_num(self):
        print("deletor")
        del self._id_num
```

```
del patient1.id_num
print(patient1.id_num)
```

```
deletor
AttributeError: 'Patient' object has no attribute '_id_num'
```

METHODS

- methods belong to the class
- methods have access to the data of the instance that calls them

naming style

- use lowercase with words seperated by underscores as neccessary to improve readability
- using one leading underscore for non-public methods and instance variables

Methods: PEP 8 Style Guide

- Method names should be written in lowercase separated by underscores.
  Example: display_data

- By convention, method names contain a verb.
  Example: find_area

- If the method returns a boolean value (True or False), by convention the name should describe this according to the context.
  Examples: is_red, has_children

example

```
class Calculator:

    def __init__(self, model, year, serial_num):

        self.model = model
        self.year = year
        self._serial_num = serial_num

    def add(self, a, b):
        return a*b

    def subtract(self, a, b):
        if a >= b:
            return a - b
        else:
            raise ValueError("a is smaller than b")

    def multiply(self, a, b):
        return a*b
```

```
    def divide(self, a, b):
        if(b != 0):
            return a*b
        else:
            raise ValueError("Denominator b is 0")
```

add, subtract,mulitply and divide are all methods

Calling a method

```
<obj_var>.<method>(<params>)
```

ex

```
my_calc.add(5,4)
```

in this case we dont add self while calling the object as the object automatically sends a reference to it when we do the dot notation

Alternative Syntax to Call a Method

You can also call a method using this syntax:

<ClassName>.<method_name>(<instance>, <args>)

This is the order of the elements:

- Name of the class
- Dot
- Name of the method
- Within parentheses, the instance as the first parameters and the arguments.

For example:

```
class Bus:

    def __init__(self, color):
        self._color = color

    def welcome_student(self, student_name):
        print(f"Hello {student_name}, how are you today? ,the bus has a great {self._color} color
")

bus = Bus("blue")
Bus.welcome_student(bus, "Johnathan")
bus.welcome_student("Johnathan")
```

You would create an instance and call the method:

To get this output:

```
Hello Johnathan, how are you today? ,the bus has a great blue color
Hello Johnathan, how are you today? ,the bus has a great blue color
```

both achieve the same output as we pass the instance in both cases

Non-Public Methods and Name Mangling

- To make a method "non-public", you should add a leading underscore.

Example: def _display_data:

- If you add two underscores, the process of name mangling will occur, just like with non-public attributes.

Example: def __display_data:

DEFAULT ARGUMENTS

ex :

```
def show_data(self,show_owner = True):
    ......

while calling it you can either do
show_data() /the default value will be be passes
or pass a value
show_data(False)
```

CALLING METHODS FROM ANOTHER METHOD

inside the class

```
self.method_name(<args>)
```

ex:

```
class CashRegister:

    tax = 0.05

    def __init__(self, cashier, serial):
        self.cashier = cashier
        self._serial = serial

    @property
    def serial(self):
        return self._serial

    def display_total(self, subtotal):
        print("=== Welcome to our store ===")
        print("The subtotal is:", subtotal)
```

```
        print("The tax is:", self._calculate_tax(subtotal))
        print("----------------------")
        print("The total is:", self._calculate_total(subtotal))

    def _calculate_total(self, subtotal):
        return subtotal + self._calculate_tax(subtotal)

    def _calculate_tax(self, amount):
        return amount * CashRegister.tax

register = CashRegister("Melanie", "3453513")
register.display_total(5022.5) # we cant call _calculate_total and _calculate_tax as they are non
public methods
```

```
class CashRegister:

    tax = 0.05
    def __init__(self, cashier, serial):
        self.cashier = cashier
        self._serial = serial
    @property
    def serial(self):
        return self._serial
    def display_total(self, subtotal):
        print("=== Welcome to our store ===")
        print("The subtotal is:", subtotal)
        print("The tax is:", self._calculate_tax(subtotal))
        print("----------------------")
        print("The total is:", self._calculate_total(subtotal))
    def _calculate_total(self, subtotal):
        return subtotal + self._calculate_tax(subtotal)
    def _calculate_tax(self, amount):
        return amount * CashRegister.tax

register = CashRegister("Melanie", "3453513")
register.display_total(5022.5)
```

we cant call _calculate_total and _calculate_tax as they are non public methods

OUTPUT

```
The subtotal is: 5022.5
The tax is: 251.125
----------------------
The total is: 5273.625
```

Method Chaining

In Object-Oriented Programming, **Method Chaining** is a very common syntax that lets us call several methods one after the other on the same instance.

ex:

- We can do this by returning self from a method.
- To make the code more readable, you can write the method calls in several lines using \ to indicate

- To make the code more readable, you can write the method calls in several lines using `\` to indicate that the next line is a continuation of the current line.

```
class Pizza:

    def __init__(self):
        self.toppings = []

    def add_topping(self, topppping):
        self.toppings.append(topppping)
        return self #returns the object instance

    def display_topping(self):
        print("the toppings this pizza has are")
        for topping in self.toppings:
            print(topping)

pizza = Pizza()
pizza.add_topping("olives").add_topping("cheese").add_topping("tomato").display_topping()

print("\n using \\ \n ")
#or another way
pizza.add_topping("chicken")\
    .add_topping("rabbit")\
    .add_topping("capsicum") \
    .display_topping()
```

OUTPUT

```
the toppings this pizza has are
olives
cheese
tomato

 using \

the toppings this pizza has are
olives
cheese
tomato
chicken
rabbit
capsicum
```

Method chaining can be used to improve the readability of your code when it makes the code more concise and when it avoids creating variables to store intermediate results.

In our previous example, without method chaining, we would have had to write the name of the instance for each function call:

```
pizza.add_topping("mushrooms")
pizza.add_topping("olives")
pizza.add_topping("chicken")
pizza.display_toppings()
```