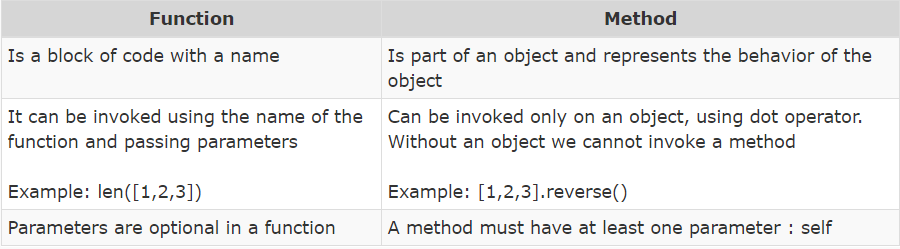
**Function vs methods**



* OOP is a style of programming which allows us to club data and behavior together
* This is more suited for coding real life scenarios
* Objects are real world entities
* Classes are logical entities and are used for classification.
* Class is a description of attributes and behavior that objects of that classification should possess
* Common attributes are created using a special method called \_\_init\_\_
* Objects can be created using ClassName() or using object literals for the built in classes
* Attributes are created using reference\_variable.attribute\_name = value syntax.
* Behavior is created by defining a function inside the class having a special parameter called self.

Encapsulation:

Binding data and code/property in single using.

Advantage:

Improves security.

Prevents accidental modification of any variable.

How to achieve encapsulation:

Declaring variable private.

Example:

Let take below code snippet.

class Customer:

    def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):

        self.cust\_id = cust\_id

        self.name = name

        self.age = age

        self.\_\_wallet\_balance = wallet\_balance

    def update\_balance(self, amount):

        if amount < 1000 and amount > 0:

            self.\_\_wallet\_balance += amount

    def show\_balance(self):

        print ("The balance is ",self.\_\_wallet\_balance)

c1=Customer(100, "Gopal", 24, 1000)

print(c1.\_\_wallet\_balance)

But with the way currently it is coded, the data can be accidentally changed by directly assigning a incorrect value to it as shown below:

c1=Customer(100, "Gopal", 24, 1000)

c1.wallet\_balance = 10000000000

c1.show\_balance()

We can put a lock on that data by adding a double underscore in front of it, as shown below:

class Customer:

    def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):

        self.cust\_id = cust\_id

        self.name = name

        self.age = age

        self.\_\_wallet\_balance = wallet\_balance #Using encapsulation

    def update\_balance(self, amount):

        if amount < 1000 and amount > 0:

            self.\_\_wallet\_balance += amount

    def show\_balance(self):

        print ("The balance is ",self.\_\_wallet\_balance)

c1=Customer(100, "Gopal", 24, 1000)

print(c1.\_\_wallet\_balance)

Now if we try to assign a value to a private variable, we end up creating a new attribute in python

Note:

In python private variable are just for name purpose, we can change the value of private variable using below convention.

instance.\_ClassName\_\_privateVariableName ----- change value of private variable

**Getter and Setter methods**

To have a error free way of accessing and updating private variables, we create specific methods for this. Those methods which are meant to set a value to a private variable are called setter/mutator methods and methods meant to access private variable values are called getter/accessor methods. The below code is an example of getter and setter methods:

class Customer:

    def \_\_init\_\_(self, id, name, age, wallet\_balance):

        self.id = id

        self.name = name

        self.age = age

        self.\_\_wallet\_balance = wallet\_balance

    def set\_wallet\_balance(self, amount):  #Setter methods

        if amount < 1000 and amount>  0:

            self.\_\_wallet\_balance = amount

    def get\_wallet\_balance(self):          #Getter method

        return self.\_\_wallet\_balance

c1=Customer(100, "Gopal", 24, 1000)

c1.set\_wallet\_balance(120)

print(c1.get\_wallet\_balance())

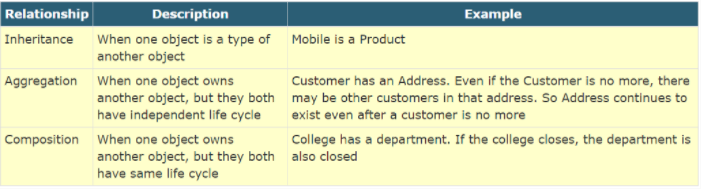
**Summary of encapsulation**

* Encapsulation is preventing access to a data outside the class
* Adding a \_\_ in front of a attribute makes it private
* Getter and Setter methods should be used to access a private attribute

**################**

**# OPPS in python #**

**################**



**Aggregation (is-A relationship)**

One object depending on another but they have different lifecycle.

If class A owns/uses class B ( instance of B ) , then class A is said to aggregate class B.

This is also commonly known as "has-A" relationship

For example, In below code , a Customer has an Address is an example of aggregation.

class Customer:

class Customer:

    def \_\_init\_\_(self, name, age, phone\_no, address):

        self.name = name

        self.age = age

        self.phone\_no = phone\_no

        self.address = address

    def view\_details(self):

        print (self.name, self.age, self.phone\_no)

        print (self.address.door\_no, self.address.street, self.address.pincode)

    def update\_details(self, add):

        self.address = add

class Address:

    def \_\_init\_\_(self, door\_no, street, pincode):

        self.door\_no = door\_no

        self.street = street

        self.pincode = pincode

    def update\_address(self):

        pass

add1=Address(123, "5th Lane", 56001)

add2=Address(567, "6th Lane", 82006)

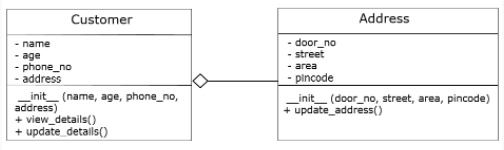
cus1=Customer("Jack", 24, 1234, add1)

cus1.view\_details()

cus1.update\_details(add2)

cus1.view\_details()

We can represent this aggregation symbolically in below way-



Example 1: Below example also show the aggregation

class Customer:

    def \_\_init\_\_(self, name, age, phone\_no):

        self.name = name

        self.age = age

        self.phone\_no = phone\_no

    def purchase(self, payment): #using instance of payment class.

        if payment.type == "card": #Accessing attribute of payment instance/class

            print ("Paying by card")

        elif payment.type == "e-wallet":

            print ("Paying by wallet")

        else:

            print ("Paying by cash")

class Payment:

    def \_\_init\_\_(self, type):

        self.type = type

payment1=Payment("card")

c=Customer("Jack",23,1234)

c.purchase(payment1)

**Composition (Has-A) Relation:**

When existence of one object is dependent on other object.

Class Car:

def \_\_init\_\_(self):

#

#Some methods and code

Class Engine:

def \_\_init\_\_(self):

#

**Inheritance**

When a class inherits from another class, then those classes are said to have an inheritance relationship. The class which is inheriting is called the child/sub/derived class and the class which is getting inherited is called the parent/super/base class.

Inheritance is also called as **"is-A" relationship.**

From a code perspective, a child class inherits:

* Constructor
* Non Private Attributes (instance, static)
* Non Private Methods (instance , static)
* Unlike other languages, private variables get inherited in Python.

**Accessing parent class private variable:**

For accessing/mutating parent class private variable, we need to define getter and setters in parent class.

class Phone:

    def \_\_init\_\_(self, price, brand, camera):

        print ("Inside phone constructor")

        self.\_\_price = price

        self.brand = brand

        self.camera = camera

    def buy(self):

        print ("Buying a phone")

    def return\_phone(self):

        print ("Returning a phone")

    def get\_price(self):

        return self.\_\_price

    def set\_price(self,price):

        self.\_\_price=price

class FeaturePhone(Phone):

    pass

class SmartPhone(Phone):

    def check(self):

        print(self.get\_price())

s=SmartPhone(20000, "Apple", 13)

s.check()

Method overriding is also called as **Polymorphism**.

**Child class can also override the parent class constructor just by creating a constructor in child class.**