Constructor

Class functions that begin with double underscore __ are also called special functions as they have special meaning.

Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.

Self Parameter

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) - this is all the special self is about.

~~init methods~~

The e.g

```
class GFG:
        def __init__(self, name, company)
                self.name = name
                self.company = company
        def show(self):
                print("Hello my name is" + self.name
                        "work in"+ self.company +".

obj = GFG ("John", "OPPO")
obj.show()
```

The self parameter does not call it to be self, You can use any other name instead of it. Here we change the self to the word someone and the output will be the same.

__init_()method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initializi the object's state. Like methods, a constructor also contains a collection of statements (i.e. that are executed at the time of object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# Sample class with init method
class Person:
      # init method or constructor
      def __init__(self, name):
            self.name = name
      # Sample Method
      def say_hi(self):
            print('Hello, my name is', self.name)
p = Person ("Nikhil")
p.say_hi()
```

Output

```
Hello, my name is Nikhil
```

__str()metod

Python has a particular method called __str__(that is used to define how a class object should be represented as a string. When a class object is used to create a string using the built-in function print() and str(), the __str__() functi is automatically used. You can alter how object of a class are represented in strings by defining the __str__() method.

```
class GFG:
    def __init__(self, name, company)
        self.name = name
        self.company = company
    def __str__(self):
        return f"My name is {self.name} and
                    I work in {self.company}."
my_obj = GFG("John", "OPPO")
print(my_obj)
```

Output

```
My name is John and I work in OPPO.
```

Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructer or method with self whereas class variables are variables whose value is assigned in the class.

(*) Variable inside method or constructer are ~~class~~ called instance.

OOPS

Modularity

Modularity in OOP refers to grouping components with related functionality into a single unit. This helps in robustness, readability and reusability.

Encapsulation

```
class Base:
    def __init__(self):
      # Protected member
        self._a = 2
class Derived(Base):
    def __init__(self):
    # calling constructor of Base class.
        Base.__init__(self)
        print ("Calling protected member of base
                                   class:", self._a)

    # Modify the protected variable:
        self._a = 3
        print ("Calling modified protected
                       member outside class:",
                                 self._a)


obj1 = Derived()
obj2 = Base()
print ("Accessing protected member of obj1:",
                        obj1._a)

print ("Accessing protected member of obj2:",
                          obj2._a)
```

Note

In python datatype is a class. When we creating a variable this is the object of those class.

In Class we have two things

① Data or Property or Attributes.

② Functions or behaviour or method.

The name of class should be in Pascal Case.

the name of method should be in snake case

Ex Pascal Case — ThisIsPascalCase

Camel Case — thisIsCamelCase

Snake Case — this_is_snake_case.

Object= object is a instance of the class

Let's build a software of atm machine.

```
class Atm:

def __init__(self):
    print("hello")
def menu(self):
    pass

  sbi= Atm()
  ↳hello
```

Special /magic /dunder method.

↳Which keywords start with double underscore and ends with double underscore ~~and~~.

↳ Dunder method not called by object.

↳ In ~~special~~ specific case it executed.

Constructor is ~~used~~ special type of magic method. It's not directly operated by user in our application which functionality we don't want to give access to user we keep that thing under constructor.

#When we write sbi.withdraw() that means in withdraw method we pass sbi object so it #showing . 0 positional argument but 1 is given

Q Why self is required in every method.

(Ans) ↳ In OOPS.. in a class one method can't directly call another method and data the method only operated by those class object not by other class object so when write self then every cases object is called by this data class and method can communicat through the object which we written in the form of self.

(8) How can we create our own data type. we build fraction data type.

```
class Fraction:
      def __init__(self,n,d):
          self.num = n
          self. den = d
      def __str__(self):
          return "{}/{}". format(self num, self . den)

      def __add__(self, other):
          temp_num = self.num* other.den + other.num*
                                            self.den
          temp_den = self.den * other.den
          return "{}/{}". format(temp_num, temp_den)
      def __scb__ (self, other):
          temp_num = self.num * other.den - other.num*
                                            self .den
          temp_den = self.den * other.den
          return "{}/{}". ferormat (temp_num, temp_den)
      def _mul__(self,other):
          temp_num = self.num * other.num
          temp_den = self.den * ~~temp~~ other .den
          return "{}/{}". format(temp_num), temp_
                                              den
      def __truediv__(self, other):
          temp_num = self. num * other.den
          temp_den = self. den * other. num
          return "{}/{}". format(temp_num, temp_d
```

call

```
x = Fraction(3,4)
y = Fraction(4,5)
```

~~out~~

```
print(x+y)
```

Output

```
31/20
```

Encapsulation

```
class Atm:
  def __init__(self):
    self.__pin = ""
    self.__balance=0
    self.__menu()
  def get_pin(self):
     return self.__pin
  def set_pin(self, newpin):
     if type(new_pin)==str:
        self.__pin =new_pin
        print("pin Changed")
     else:
        print("Not allowed")
  def __menu(self):
      user_input = input("""Hello, how would you like
                           to proceed?
                           1. Enter 1 to create pin
                           2. Enter 2 to deposit
                           3. Enter 3 to withdraw
                           4. Enter 4 to check balance
                           5. Enter 5 to exit
                           """)
      if user_input == "1":
           self.create_pin()
      elif user_input =="2":
           self.deposit()
      elif user_input == "3":
           self.withdraw()
      elif user_input == "4":
           self.check_balance()
      else: print("bye")
```

```
def deposit(self):
  temp = input ("Enter your pin")
  if temp == self.--pin:
      amount = int (input ("Enter the amount"))
      self.--balance = self.--balance + amount
      print ("Deposit Successful")
  else:
      print ("invalid pin")
def withdraw (self):
  temp = input ("Enter your pin")
   if temp == self.-pin:
     amount <= self.---balance
        self.--balance = self.--balance - amount
        print ("withdraw Successful")
        else:
          print ("insufficient balance")
     else:
       print ("invalid pin")
def check - balance (self):
   temp = input ("Enter your pin")
   if temp == self.--pin:
        print ( self.-- balance)
     else:
        print ("invalid pin")
```

Abstraction

Abstraction is the process of hiding a method's real implementation and only exposing its required characteristics and behavior.

For full fill the purpose of abstraction we have to add this above the 'Abstracted class'

from abc import ABC, abstractmethod.

abc → Predefined module

ABC → Class in 'abc'

abstractmethod → method.

We can not create 'object' of abstracted class or class that inherit 'Abstract Class' of predefined 'abc' module'.

Example-1) (abstract base class)

```
1) from abc import ABC, abstractmethod
   class BankApp(ABC):
        def database (self):
             ~~pass~~ print ("Connected to database")

        @ abstractmethod
        def security (self):
             pass

        @ abstractmethod
        def display(self):
           pass
```

```
1 i) class MobileApp (BankApp):
       def ~~s~~mobile_login (self):
             print ('login Into mobile')
       def security (self):
             print ('mobile security)
       def display (self):
             print ('display')
```

Create Object

```
mob = MobileApp( )
~~mob~~ mob. security ( )
```

Output

```
mobile security
```

```
obj = BankApp( )
```

→ we can't create object of abstracted class

Ex-2

```
from abc import ABC, abstractmethod
class LibraryItem (ABC):
    def __int__ (self, title, author):
        self. title = title
        self. author = author
        self. checked_out = False

    @abstractmethod
    def check_out (self):
        pass
    @abstractmethod
    def check-in(self):
        pass
class Book (LibraryItem):
    def __init__ (self, title, author, num-pages):
        super(). __init__ (title, author)
        self. num-pages = num-pages
    def check-out (self):
        if not self. checked_out:
            self. checked_out = True
            print (f"{self. title} by {self. author}
                       checked out successfully")
        else:
            print ("This book is already
                         checked out.")
    def check-in(self):
        if self. checked_out:
            self. checked-out = False
            print (f"{self. title} by {self.author}
                        checked in successfully")
        else:
            print ("This book is not checked
                                    out.")
```

```
class DVD (LibraryItem):
    def __init__(self, title, director, duration):
    super().__init__(title, director)
    self.duration = duration
    def check_out(self):
        if not self.checked_out:
            self.checked_out = True
            print(f"{self.title} by {self.author}
                        checked out successfully")
        else:
            print("This DVD is already checked
                        out.")

    def check_in(self):
        if self.checked_out:
            self.checked_out = False
            print(f"{self.title} by {self.author}
                        checked in successfully")
        else:
            print("This DVD is not checked out.")
```

Create Object

```
my_book = Book("Python","IPCS",544)
my_dvd = DVD("Inception","IPCS",148)
```

call

```
my_book.check_out()
```

O/P Python by IPCS checked out successfully.

```
my_dvd.check_out()
```

O/P Inception by IPCS checked out

```
my_book.check_in()
```

O/P Python by IPCS checked in

```
my_dvd.check_in()
```

O/P Inception by IPCS checked in successfully

Abstraction

1) Abstraction solves the problem in the design level.

2) Abshaction is used for hiding the unwanted data and giving relevant data.

3.) Abstraction lets you focus on what the object does instead of how it does it.

4) Abstraction - Outer layout, used in terms of design.

For Example -
Outer Look a Mobile Phone, like it has a display screen and keypad buttons to dial a number.

Encapsulation

1.) Encapsulation solves the problem in the implementation level.

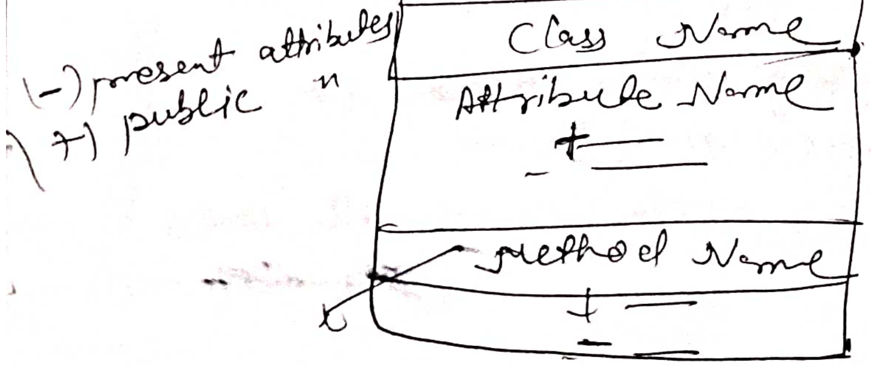
2.) Encapsulation means hiding the code and data into a single unit to protect the data from outside world.

3) Encapsulation means hiding the internal details or mechanics of how an object does something.

4) Encapsulation - Inner layout used in terms of implementation.

For Example - Inner Implementation details of a Mobile phone, how keypad button and Display screen are connected with each other using circuits.

Class Diagram


(-) present attributes
(+) public "
Class Name
Attribute Name
+
-
Method Name
+
-