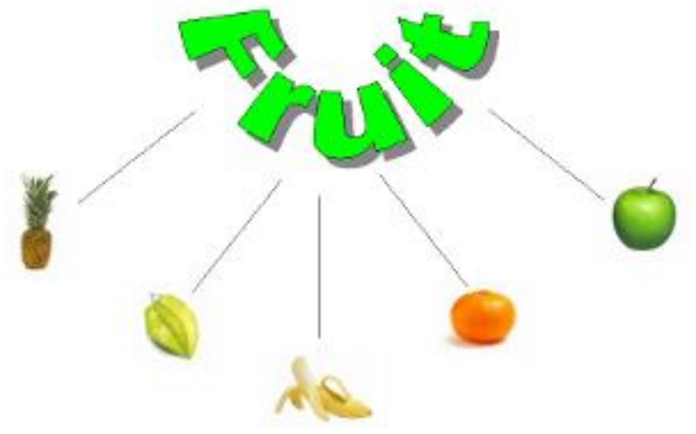
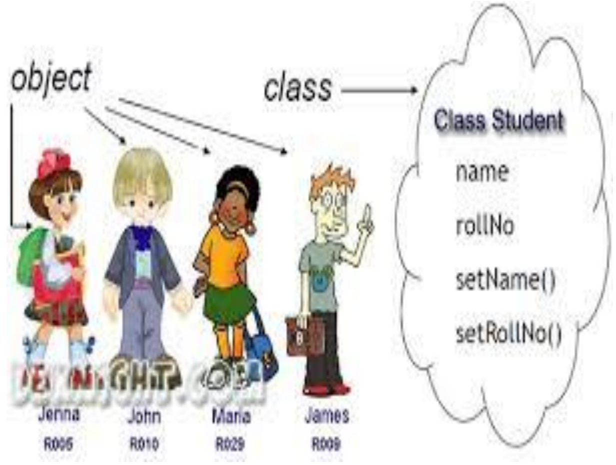
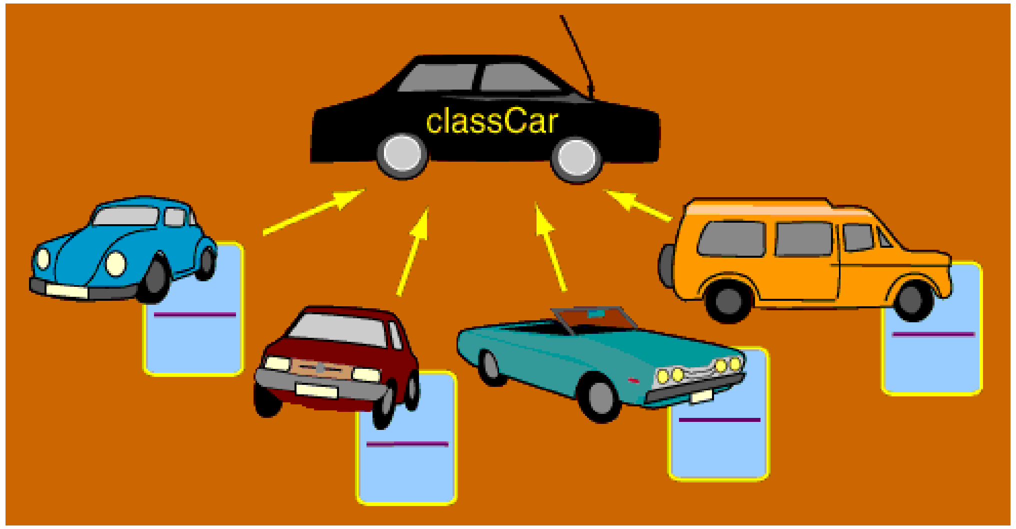
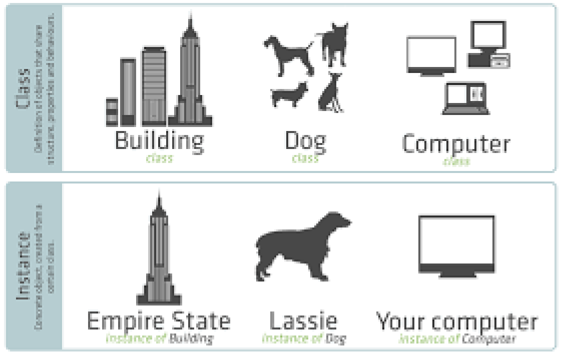
Python

Classes & Objects



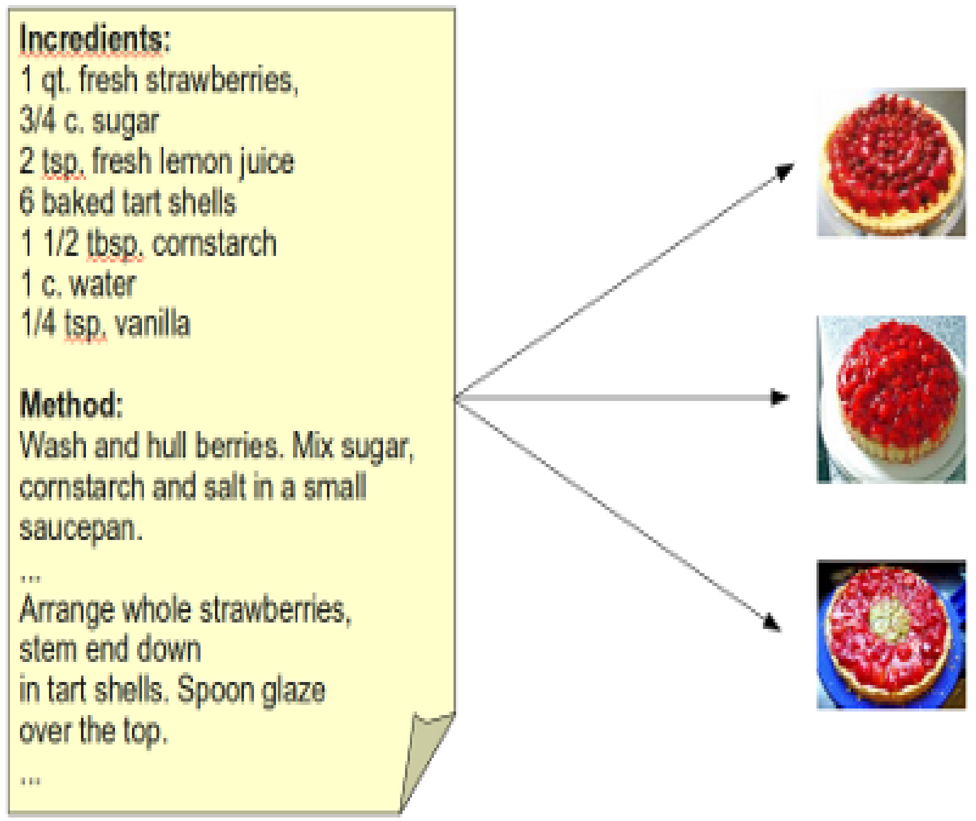






What are classes ?

- Classes are like a blueprint or a prototype that you can define to use to create objects.
- A class brings to life, the abstract charecteristics of a real life ***thing***.
- A class definition can be compared to the recipe to bake a cake. A recipe is needed to bake a cake. The main difference between a recipe (class) and a cake (an instance or an object of this class) is obvious.
- A cake can be eaten when it is baked, but you can't eat a recipe, unless you like the taste of printed paper.
- Like baking a cake, an OOP program constructs objects according to the class definitions of the program.
- A class contains variables and methods. If you bake a cake you need ingredients and instructions to bake the cake.
- Accordingly a class needs variables and methods. There are class variables, which have the same value in all methods and there are instance variables, which have normally different values for different objects.
- A class also has to define all the necessary methods, which are needed to access the data.



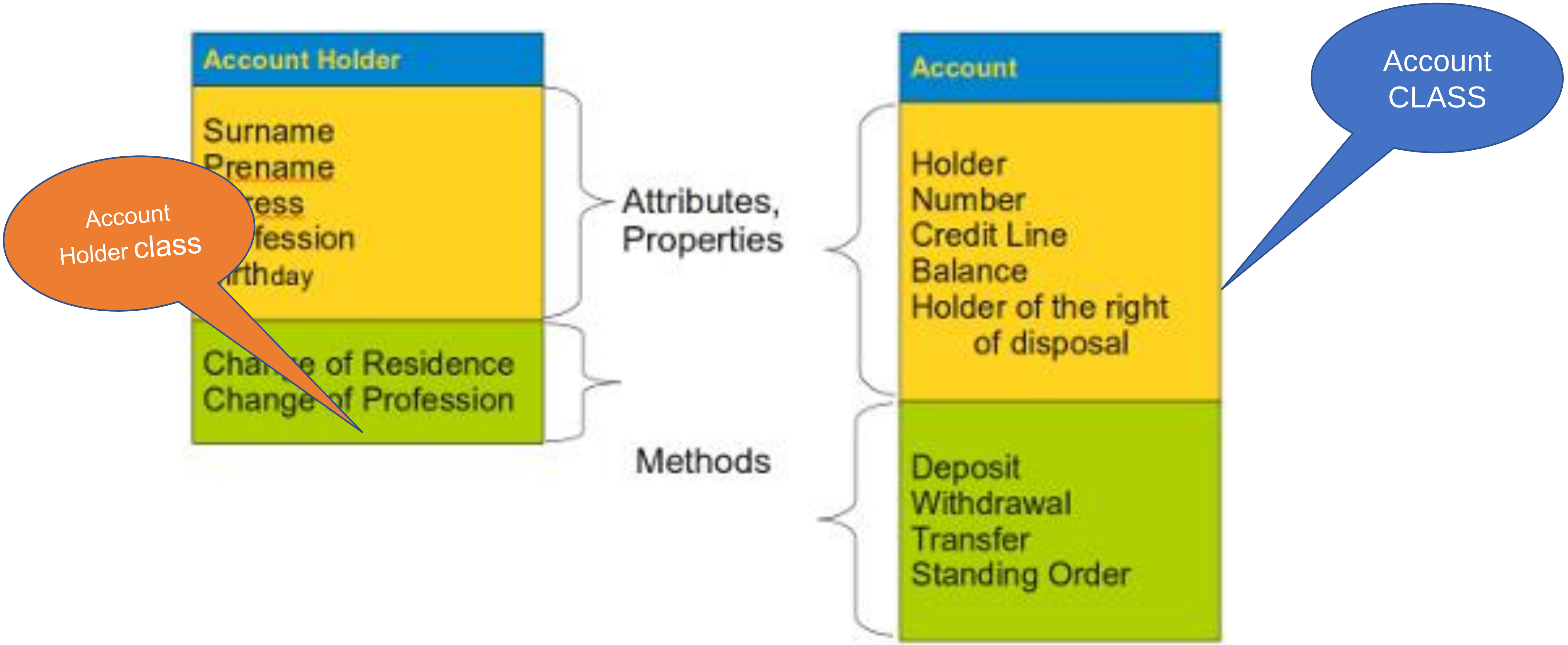
Incredients:
1 qt. fresh strawberries,
3/4 c. sugar
2 tsp. fresh lemon juice
6 baked tart shells
1 1/2 tbsp. cornstarch
1 c. water
1/4 tsp. vanilla

Method:
Wash and hull berries. Mix sugar, cornstarch and salt in a small saucepan.
...
Arrange whole strawberries, stem end down
in tart shells. Spoon glaze over the top.
...

Objects & Instances

- So what does a class do? It brings to life a Real-life ***THING.***
- This Thing is an ***"Object".***
- *An object created from a class at runtime is an **instance** of a class.*
- *The process of creating the instance is called **instantiation.***
- *Example, Creation of a Chocolate cake from the recipe for Baking Chocolate Deserts is Instantiation of a Chocolate cake Object from a Chocolate Class.*
- The object consists of state and the behavior that's defined in the object's classes.
- *An object is also an instance of a class.*
- *A class usually contains :-*
  - *Attributes.*
  - *Properties.*
  - *Methods.*
- *At class level, the variables are refered to as "Class Variables". At the instance level it is known as "Instance Variables"*

Objects & Instances



Creating classes and objects

- *Now that we have theorized a lot, lets get our hands dirty and actually work with the nuts and bolts.*

Creating classes :-

```
class Shark:
    def swim(self):
        print("The Shark is swimming \n")
    def be_awesome(self):
        print("The Shark is being awesome \n")
```

- *So there we are, we created a Class , which is nothing else but*

(Disclaimer : Don' try this at Home)

Creating classes and objects

- There were two functions inside the Class. These functions are called ***methods***.
- The argument to these functions is the word ***self.*** It is used as a reference to the objects that are made out of the class(will see in a while).
- *We have till now just created a Recipe. Now lets create the* ***Real Thing.***

Creating The Real World Thing – OBJECT

- An object is an instance of a class
- Create an object by enclosing the class in parantheses()

```
>>> sammy = Shark()
```

- *Object created, now lets see what this Object can do.*

```
>>> sammy.swim()
The Shark is swimming
>>> sammy.be_awesome()
The Shark is being awesome
```

Creating classes and objects

- The Shark object sammy is using the two methods swim() and be_awesome(). We called these using the dot operator (.), which is used to reference an attribute of the object. In this case, the attribute is a method and it's called with parentheses, like how you would also call with a function.
- Because the keyword self was a parameter of the methods as defined in the Shark class, the sammy object gets passed to the methods. The self parameter ensures that the methods have a way of referring to object attributes.

```
class Shark:
    def swim(self):
        print("The shark is swimming.")
    def be_awesome(self):
        print("The shark is being awesome.")
def main():
    sammy = Shark()
    sammy.swim()
    sammy.be_awesome()
if __name__ == "__main__":
    main()
```

Can you tell me what does _name_ and _main_ mean?

Bound & Unbound Method Calls

- How do you call methods from an object?
- There are two broad ways of doing it :-
  - ✓ Call the method from the instance.
  - ✓ Call the method from inside the class itself.
- Accordingly there are two ways of calling methods :-
  - ✓ Bound method calls.
  - ✓ Unbound method calls.

Bound method (instance call): To call the method we must provide an instance object explicitly as the first argument. In other words, a bound method object is a kind of object that remembers the self instance and the referenced function. So, a bound method may be called as a simple function without an instance later. Python automatically packages the instance with the function in the bound method object, so we don't need to pass an instance to call the method. In other words, when calling a bound method object, Python provides an instance for us automatically-the instance used to create the bound method object. This means that bound method objects are usually interchangeable with simple function objects, and makes them especially useful for interfaces originally written for functions such as Callback functions.

Bound & Unbound Method Calls

```
>>> class Callback:
...   def __init__(self, color):
...     self.color = color
...   def changeColor(self):
...     print(self.color)
... >>> obj = Callback('red')
>>> cb = obj.changeColor()
```

Unbound method (class call): Accessing a function attribute of a class by qualifying the class returns an unbound method object.

```
class B(object):
  def first(self):
    print("First method called")
  def second():
    print("Second method called")
ob = B()
B.first(ob)
```

Initializing attributes

- *What is an attribute of an object ?*
- *Well these are*

Initializing attributes

- Remember the ceil method we used in one of our sessions?

```
>>> import math
>>> math.ceil(1.5)
2
```

- What did we do here? We passed an attribute to a method.
- Whenever a Object is to be created, and we have to define some properties for it, we initialize these attributes by means of an ***__init__*** method.
- This is also known as **CONSTRUCTOR** Method.
- It is usually the first definition of a Class.
- Looks something like this

```
class Shark:
    def __init__(self):
        print("This is the constructor method.")
```

Initializing attributes

- Now, we want to assign a name to the Shark , instead of saying things like "The Shark".
- We will create two Sharks , "Jai", "Veeru" ☺

Try and Learn :-

➢ Create two Sharks Jai and Veeru, and print out the following by calling the methods **swim()** and **be_awesome()** :-

- Jai is Swimming
- Veeru is being Awesome()

Hint :- To receive an attribute apart from "self", we modify the __init__() method to receive the attribute

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

Class & Instance Attributes

Add , remove, modify Attributes of classes

- Instead of accessing attributes of a object, we can use the following functions as well :-
  - The **getattr(obj, name[, default])** − to access the attribute of object.
  - The **hasattr(obj,name)** − to check if an attribute exists or not.
  - The **setattr(obj,name,value)** − to set an attribute. If attribute does not exist, then it would be created.
  - The **delattr(obj, name)** − to delete an attribute.

Try and Learn :-

- Create an object **Jai** of class type **Shark**. The Shark should have an attribute **"Name"**, value should be **"Jai"**. Using the above functions find out if the name attribute exists, whats the value of the attribute, and then rename the value to **Veeru**.

Built in Class Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

Class & Instance attributes

Built in Class Attributes(contd)

- Every Python class keeps following built-in attributes and they can be accessed using dot operator
  like any other attribute −
  - ✓ **__dict__** − Dictionary containing the class's namespace.
  - ✓ **__doc__** − Class documentation string or none, if undefined.
  - ✓ **__name__** − Class name.
  - ✓ **__module__** − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
  - ✓ **__bases__** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.
- Class variables are defined within the class construction. Because they are owned by the class itself, class variables are shared by all instances of the class. They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.
- Defined outside of all the methods, class variables are, by convention, typically placed right below
  the class header and before the constructor method and other methods.

Class & Instance attributes

```
class Shark:
    "Welcome to Base Class"
    "Base class for Shark Objects"
        def __init__(self,name):
        print("This is the constructor  method.")
      self.name = name
    def swim(self):
    print("The Shark  %s is swimming \n"%
(self.name))
    def be_awesome(self):
        print("The Shark %s is being awesome \n"%
(self.name))
```

```
def main():
  Jai = Shark('Jai')
  Jai.swim()
  Jai.be_awesome()
  Veeru = Shark('Veeru')
  Veeru.swim()
  Veeru.be_awesome()
  print ("Shark.__doc__:", Shark.__doc__)
  print ("Shark.__name__:", Shark.__name__)
  print ("Shark.__module__:",
          Shark.__module__)
  print ("Shark.__bases__:", Shark.__bases__)
  print ("Shark.__dict__:", Shark.__dict__ )
```

Class & instance variables Co-Inhabitation

```
class Shark:
    # Class variables
    animal_type = "fish"
    location = "ocean"
    # Constructor method with instance variables name and age
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Method with instance variable followers
    def set_followers(self, followers):
        print("This user has " + str(followers) + " followers")
def main():
    # First object, set up instance variables of constructor method
    sammy = Shark("Sammy", 5)
    # Print out instance variable name
    print(sammy.name)
    # Print out class variable location
    print(sammy.location)
```

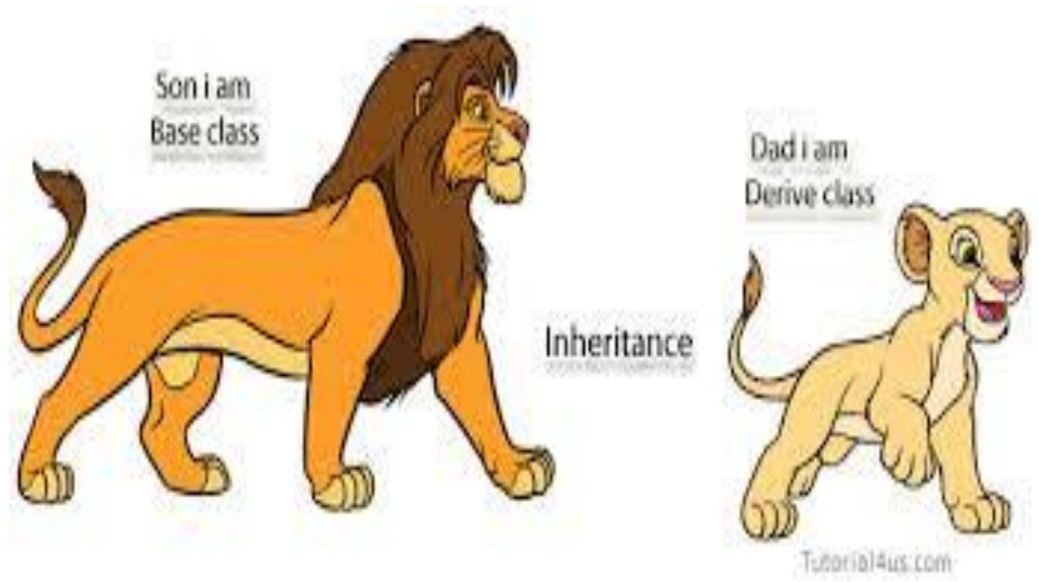
Class & Instance variables Co-Inhabitation

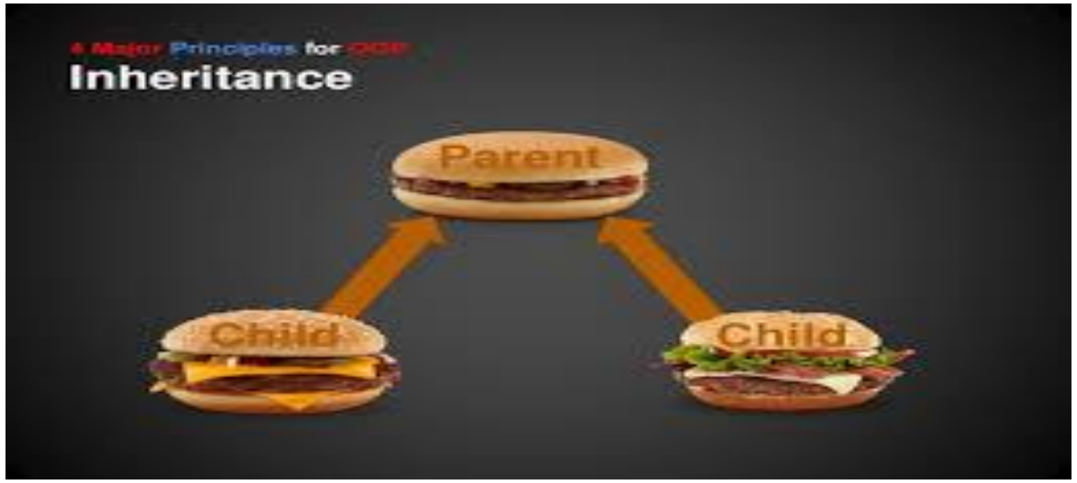
```
# Second object
    stevie = Shark("Stevie", 8)
    # Print out instance variable name
    print(stevie.name)
  # Use set_followers method and pass followers instance variable
    stevie.set_followers(77)
    # Print out class variable animal_type
    print(stevie.animal_type)
if __name__ == "__main__":
    main()
```

Output :

```
Sammy
ocean
Stevie
This user has 77 followers
fish
```

Class : Inheritance





- Object-oriented programming creates reusable patterns of code to curtail redundancy in development projects. One way that object-oriented programming achieves recyclable code is through inheritance, when one subclass can leverage code from another base class.
- **Inheritance** is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents.
- Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.
- Because the Child subclass is inheriting from the Parent base class, the Child class can reuse the code of Parent, allowing the programmer to use fewer lines of code and decrease redundancy.

Class : Inheritance

PARENT CLASS

- Parent or base classes create a pattern out of which child or subclasses can be based on. Parent classes allow us to create child classes through inheritance without having to write the same code over again each time. Any class can be made into a parent class, so they are each fully functional classes in their own right, rather than just templates.
- Let's say we have a general **Bank_account** parent class that has **Personal_account** and **Business_account** child classes. Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of Bank_account. The **Business_account** subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an **employee_identification_number** variable.
- Similarly, an Animal class may have eating() and sleeping() methods, and a Snake subclass may include its own specific hissing() and slithering() methods.
- Let's create a Fish parent class that we will later use to construct types of fish as its subclasses. Each of these fish will have first names and last names in addition to characteristics.
- We'll create a new file called fish.py and start with the __init__() constructor method, which we'll populate with first_name and last_name class variables for each Fish object or subclass.

Class : Inheritance

PARENT CLASS

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name
```

- Since most of the fish we'll be creating are considered to be bony fish (as in they have a skeleton made out of bone) rather than cartilaginous fish (as in they have a skeleton made out of cartilage), we can add a few more attributes to the __init__() method:

```
class Fish:
    def __init__(self, first_name, last_name="Fish",
                 skeleton="bone", eyelids=False):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids
    def swim(self):
        print("The fish is swimming.")
    def swim_backwards(self):
        print("The fish can swim backwards.")
```

Class : Inheritance

Child Classes

- Child or subclasses are classes that will inherit from the parent class. That means that each child class will be able to make use of the methods and variables of the parent class.
- For example, a Goldfish child class that subclasses the Fish class will be able to make use of the swim() method declared in Fish without needing to declare it.
- We can think of each child class as being a class of the parent class. That is, if we have a child class called Rhombus and a parent class called Parallelogram, we can say that a Rhombus is a Parallelogram, just as a Goldfish is a Fish.
- The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

"Now we will develop a child class, goldfish. Lets see what all it can do"

```
class GoldFish(Fish):
        pass
```

- The **GoldFish** class is a child of the **Fish** class. We know this because of the inclusion of the word Fish in parentheses.

Class : Inheritance

Child Classes

- The GoldFish class will just inherit all the characteristics of its parents, and lets see how .

Try and Learn

```
def main():
    Nemo = GoldFish("Nemo")
    print("%s is a %s  with %s"%(Nemo.first_name,Nemo.last_name,Nemo.skeleton))
    Nemo.swim()
    Nemo.swim_backwards()
if __name__== "__main__":
    main()
```

- With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the pass keyword, which we'll do in this case:
- Let's create one more class called **clownfish** which has some methods of its own.

Class : Inheritance

Child Classes

```
class clownfish(Fish):
    def live_with_anemone(self):
        print("%s lives with anemone"%self.first_name)

def main():
    Nemo = GoldFish("Nemo")
    print("%s is a %s  with %s"%(Nemo.first_name,Nemo.last_name,Nemo.skeleton))
    Nemo.swim()
    Nemo.swim_backwards()
    clown= clownfish("Jazzy")
    clown.live_with_anemone()
if __name__== "__main__":
    main()
```

Class : inheritance

Overriding Parent Methods

- So far, we have looked at the child class **GoldFish** that made use of the pass keyword to inherit all of the parent class **Fish** behaviors, and another child class **Clownfish** that inherited all of the parent class behaviors and also created its own unique method that is specific to the child class. Sometimes, however, we will want to make use of some of the parent class behaviors but not all of them. When we change parent class methods we override them.
- When constructing parent and child classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.
- We'll create a **Shark** child class of the **Fish** parent class. Because we created the **Fish** class with the idea that we would be creating primarily bony fish, we'll have to make adjustments for the Shark class that is instead a cartilaginous fish. In terms of program design, if we had more than one non-bony fish, we would most likely want to make separate classes for each of these two types of fish.
- Sharks, unlike bony fish, have skeletons made of cartilage instead of bone. They also have eyelids and are unable to swim backwards. Sharks can, however, maneuver themselves backwards by sinking.
- In light of this, we'll be overriding the **__init__()** constructor method and the **swim_backwards()** method. We don't need to modify the swim() method since sharks are fish that can swim. Let's take a look at this child class:

Class : inheritance

Overriding Parent Methods(Example)

```
class Shark(Fish):
    def __init__(self,first_name,last_name="Shark",skeleton="Cartilege",eyelids="True"):
        self.first_name = first_name
        self.last_name =  last_name
        self.skeleton = skeleton
        self.eyelids = eyelids
    def swim_backwards(self):
        print("Sharks cant swim backwards directly. They sink and swim back")
```

Super() Function :

- With the super() function, you can gain access to inherited methods that have been overwritten in a class object.
- When we use the super() function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.

Class : Inheritance

Super() Function(contd) :

- With the super() function, you can gain access to inherited methods that have been overwritten in a class object.
- When we use the super() function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.
- The rest of the attributes, that were supposed to be part of the parent class, now automatically comes into the childclass as part of inheritance.

Class : Inheritance

Super() Function(contd) :

- In a program that grades students, we may want to have a child class for Weighted_grade that inherits from the Grade parent class. In the child class Weighted_grade, we may want to override a calculate_grade() method of the parent class in order to include functionality to calculate a weighted grade, but still keep the rest of the functionality of the original class. By invoking the super() function we would be able to achieve this.
- The super() function is most commonly used within the __init__() method because that is where you will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

```
class Trout(Fish):
    def __init__(self, water = "freshwater"):
        self.water = water
        super().__init__(self)
```

- We have overridden the __init__() method in the Trout child class, providing a different implementation of the __init__() that is already defined by its parent class Fish. Within the __init__() method of our Trout class we have explicitly invoked the __init__() method of the Fish class.
- Because we have overridden the method, we no longer need to pass first_name in as a parameter to Trout, and if we did pass in a parameter, we would reset freshwater instead. We will therefore initialize the first_name by calling the variable in our object instance.

Class : inheritance

Super() Function(contd) :

- Now we can invoke the initialized variables of the parent class and also make use of the unique child variable. Let's refer the example below

```
class Account():
        def __init__(self,name):
                self.name = name
class Locker(Account):
        def __init__(self,locker,name):
                self.locker = locker
                super().__init__(name)
>>> lock = Locker("1001","Arun")
>>> lock.name
'Arun'
>>> lock.locker
'1001'
>>>
```

Class : Multiple Inheritance

- When a class inherits from two or more parent class, it is known as Multiple inheritance.
- Can reduce lot of redundancy, but can introduce multiple ambiguity.

Lets create a **Coral_reef** child class than inherits from a **Coral** class and a **Sea_anemone** class. We can create a method in each and then use the pass keyword in the **Coral_reef** child class:

```
class Coral:
    def community(self):
        print("Coral lives in a community.")
class Anemone:
    def protect_clownfish(self):
        print("The anemone is protecting the clownfish.")
class Coral_reef(Coral,Anemone):
    pass

great_barrier = Coral_reef()
great_barrier.community()
great_barrier.protect_clownfish()
```

Class : Inheritance Functions

- Use the **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.

Format :-

- The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class

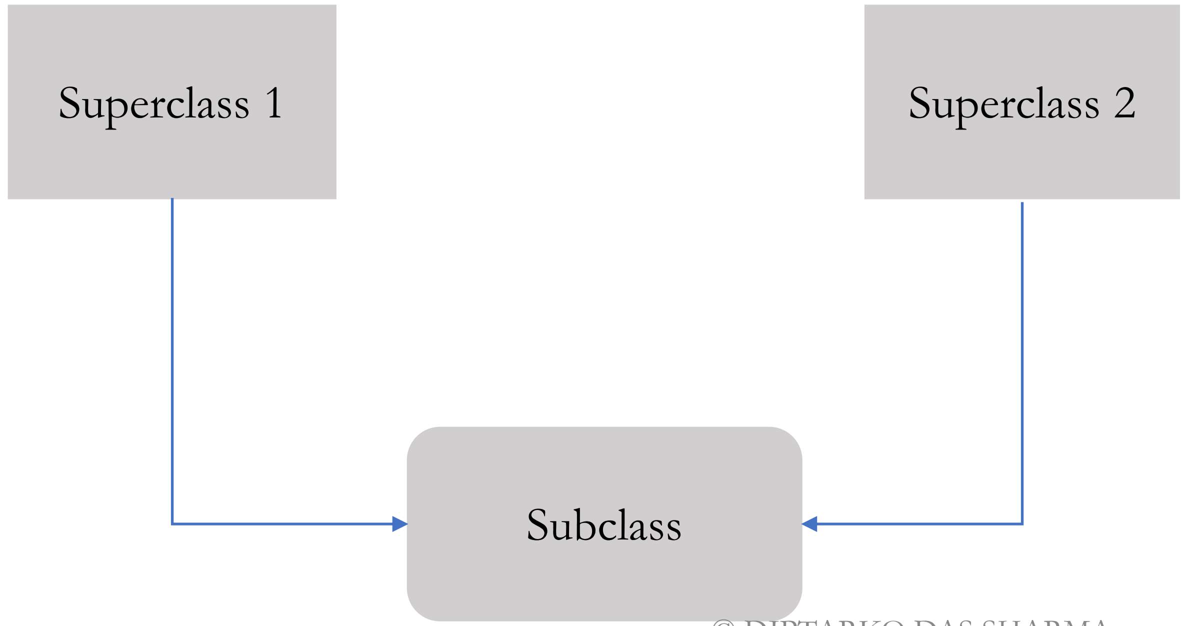
Example:-

```
issubclass(Coral_reef,Anemone)
True
isinstance(great_barrier,Coral)
True
```

Class : Multiple inheritance & MRO

super() in Multiple Inheritance :

Now that we have worked through an overview and some examples of super() and single inheritance, we will be introduced to an overview and some examples that will demonstrate how multiple inheritance works and how super() enables that functionality.



Class : Multiple inheritance & MRO

super() in Multiple Inheritance :

Let's use the below code to understand how multiple inheritance works in Python.

```
#Create Class A , which has an attribute "name'
class A:
        def __init__(self,name):
                self.name = name
# Create class B, which has an attribute surname
class B:
        def __init__(self,surname):
                self.surname = surname
# Child class C,  which inherits from both A & B
class C(A,B):
        def __init__(self,accntName,name,surname):
                self.accntName = accntName
                super().__init__('Mr.Singh')
```

Class : Multiple inheritance & MRO

super() in Multiple Inheritance :

If we use , the super() Function, the constructor method of parent ***A*** will be selected. How did that happen? Why was not the constructor for ***B*** selected?

The method resolution order (or MRO) tells Python how to search for inherited methods. This comes in handy when you're using super() because the MRO tells you exactly where Python will look for a method you're calling with super() and in what order.

Every class has an ***.__mro__*** attribute that allows us to inspect the order, so let's do that:

>> C.__mro__

(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)

This tells us that methods will be searched first in ***C***, then in ***A***, then in ***B***, then Object(***c***).

Now what to do, if we want to inherit from the parent ***B.***

Class : Multiple inheritance & MRO

```
class Coral():
    def community(self):
        print('Coral lives in a community')
class Anemone():
    def print_clownfish(self):
        print('Anemones protect clownfish')
    def community(self):
        print("Anemones dont live in a community")

class Coral_reef(Anemone,Coral):
    pass
```

Class : POLYMORPHISM

- Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.
- Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them.
- When several classes or subclasses have the same method names, but different implementations for these same methods, the classes are polymorphic because they are using a single interface to use with entities of different types. A function will be able to evaluate these polymorphic methods without knowing which classes are invoked.

Class : POLYMORPHISM

Lets create two classes, who use same method names, but with different functionalities.

- ✓ #Creating Polymorphic Classes
- ✓ ''' Create two Fish classes, One is a Shark and the other is a clwon fish. Clownfish can swim forwards as well as Backwards,
- ✓ while Shark one cant do so. Clownfish is a bony skeleton, while shark will have a cartilaginous one'''

```
class Shark:
    def swim(self):
        print("The Shark is swimming")
    def swim_backwards(self):
        print("Sharks cant swim backwards")
    def skeleton(self):
        print("Sharks skeleton is made of 
cartilege")
```

```
class Clownfish:
    def swim(self):
        print("The clownfish is swimming")
    def swim_backwards(self):
        print("clownfish can swim backwards")
    def skeleton(self):
        print("Clownfish's skeleton is made of  bones")
```

Class : POLYMORPHISM

'''Creating Object instances of the class'''

sammy = Shark()

cassey = Clownfish()

'''Now here's polymorphism in action, We will call the same method from the different objects and get different results. First the Shark'''

sammy.swim()

sammy. swim_backwards()

sammy.skeleton()

Output :-

The Shark is swimming

Sharks cant swim backwards

Sharks skeleton is made of cartilege

Same method, different outputs

'''Now the clownfish'''

cassey.swim()

cassey.swim_backwards()

cassey.skeleton()

Output :-

The clownfish is swimming

clownfish can swim backwards

Clownfish's skeleton is made of bones

Class : POLYMORPHISM With Class Methods

To show how Python can use each of these different class types in the same way, we can first create a for loop that iterates through a tuple of objects. Then we can call the methods without being concerned about which class type each object is. We will only assume that these methods actually exist in each class.

```
for fish in (cassey,sammy):
    fish.swim()
    fish.swim_backwards()
    fish.skeleton()
```

Output :-

```
The clownfish is swimming
clownfish can swim backwards
Clownfish's skeleton is made of bones
The Shark is swimming
Sharks cant swim backwards
Sharks skeleton is made of cartilege
```

Same method, different outputs

Class : POLYMORPHISM WITH FUNCTIONS

We can also create a function that can take any object, allowing for polymorphism.

```
#Polymorphism with Functions
'''Let's create a function called in_the_pacific() which takes in an object we can call fish.
   Though we are using the name fish, any instantiated object will be able to be called into this function:'''
def in_the_pacific(fish):
    fish.swim()
    fish. Swim_backwards()
    fish. Skeleton()
in_the_pacific(cassey)
in_the_pacific(sammy)
```

Output:-

```
The clownfish is swimming
clownfish can swim backwards
Clownfish's skeleton is made of bones
The Shark is swimming
Sharks cant swim backwards
Sharks skeleton is made of cartilege
```

Same method, different outputs

Class : COMPOSITION

In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other objects. Talking in terms of pseudocode you may say that

```
class GenericClass:
    define some attributes and methods

class ASpecificClass:
    Instance_variable_of_generic_class = GenericClass

# use this instance somewhere in the class
    some_method(Instance_variable_of_generic_class)
```

So you will instantiate the base class and then use the instance variable for any business logic.

Class : COMPOSITION

How to Achieve composition in Python?

To achieve composition you can instantiate other objects in the class and then use those instances. For example in the below example we instantiate the Rocket class using self.rocket and then using self.rocket in the method get_maker.

```
class Rocket:
    def __init__(self, name, distance):
        self.name = name
        self.distance = distance

    def launch(self):
        return "%s has reached %s" % (self.name, self.distance)
```

Class : COMPOSITION

How to Achieve composition in Python?

```
class MarsRoverComp():
    def __init__(self, name, distance, maker):
        self.rocket = Rocket(name, distance) # instantiating the base
        self.maker = maker


    def get_maker(self):
        return "%s Launched by %s" % (self.rocket.name, self.maker)
```

Output **:-**

```
>>> z = MarsRover('rover2','till Mars','ISRO')
>>> z.get_maker()
'rover2 Launched by ISRO'
>>> z.rocket.launch()
```

Class : COMPOSITION

How to Achieve composition in Python?

Output *:-*

```
>>> z = MarsRover('rover2','till Mars','ISRO')
>>> z.get_maker()
'rover2 Launched by ISRO'
>>> z.rocket.launch()
'rover2 has reached till Mars'
```

Class : Abstract Classes

How to Achieve composition in Python?

- An abstract class can be considered a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class that contains one or more abstract methods is called an **abstract class.** An **abstract method** is a method that has a declaration but does not have an implementation.
- We use an abstract class while we are designing large functional units or when we want to provide a common interface for different implementations of a component.

Class : Abstract Classes

Abstract Base Classes in Python

- By defining an abstract base class, you can define a common **Application Program Interface(API)** for a set of subclasses. This capability is especially useful in situations where a third party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code base where keeping all classes in your mind is difficult or not possible.
- By default, Python does not provide **abstract classes**. Python comes with a module that provides the base for defining A**bstract Base classes(ABC)** and that module name is **ABC**.
- **ABC** works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

Class : Abstract Classes

Abstract Base Classes in Python

```
# Python program showing
# abstract base class work
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noofsides(self):
        pass
class Triangle(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")
```

Class : Abstract Classes

Abstract Base Classes in Python

```
class Hexagon(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")
class Quadrilateral(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")
# Driver code
R = Triangle()
R.noofsides()
```

Class : Abstract Classes

Abstract Base Classes in Python

K = Quadrilateral()

K.noofsides()

R = Pentagon()

R.noofsides()

K = Hexagon()

K.noofsides()

Output
```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

Class : SPECIAL METHODS

Special methods are Functions which are enabled for some special work like del, finding length of and object, finding sizeof an object etc.

- ***Suppose you want to print out a list. What do you do?***

```
>>> l = [1,2,3]
>>> print(l)
[1, 2, 3]
```

Now, lets create an object and see if we can do the same with the object instead of printing out the parameters explicitly.

```
'''Creating a class Book, passing attributes like Book Name,
Author, Page.
'''
class Book():
    def __init__(self,book_name,author,page):
        self.book_name = book_name
        self.author = author
        self.page = page
```

Class : SPECIAL METHODS

- *I created a class.*
- *Initialized the attributes.*
- *Will create an object out of it, and try to print out the object.*

```
>>> print(bk)
<__main__.Book object at 0x021F1FB0>
```

- *I didn't get any meaningful output here, just got a reference to the object created.*
- *Suppose, I want to know the title of the book, the Author and the pages of the book,and I don't want to use any print statement.*
- *In comes the special method __str__. The string method, which is part of any class, is used to create a string reference to an object.It is used to create an informal string representation.*
- *Lets add this below piece of code to our Class definition and re-run our code.*

```
def __str__(self):
        return "Title : {}, Author:{}, Pages
{}".format(self.book_name,self.author,self.page)
```

Class : SPECIAL METHODS

```
>>> bk = Book("Python","Dip",300)
>>> print(bk)
        Title : Python, Author:Dip, Pages 300
```

- The print, thus invoked the __str__() special method in the Object.
- Now lets see one more special method : __del__()
- Remember the method del used to delete a number, a list, a string etc?

```
>>> str = "Hello World"
>>> del str
```

- The del method invokes invokes the __del__() special method.
- So lets add the __del__() special method to our code and invoke it.

```
'''Delete an object'''
  def  __del__(self):
     print("A book is deleted")
```

Class : SPECIAL METHODS

```
>>> bk = Book("Python","Dip",300)
>>> del bk
A book is deleted
```

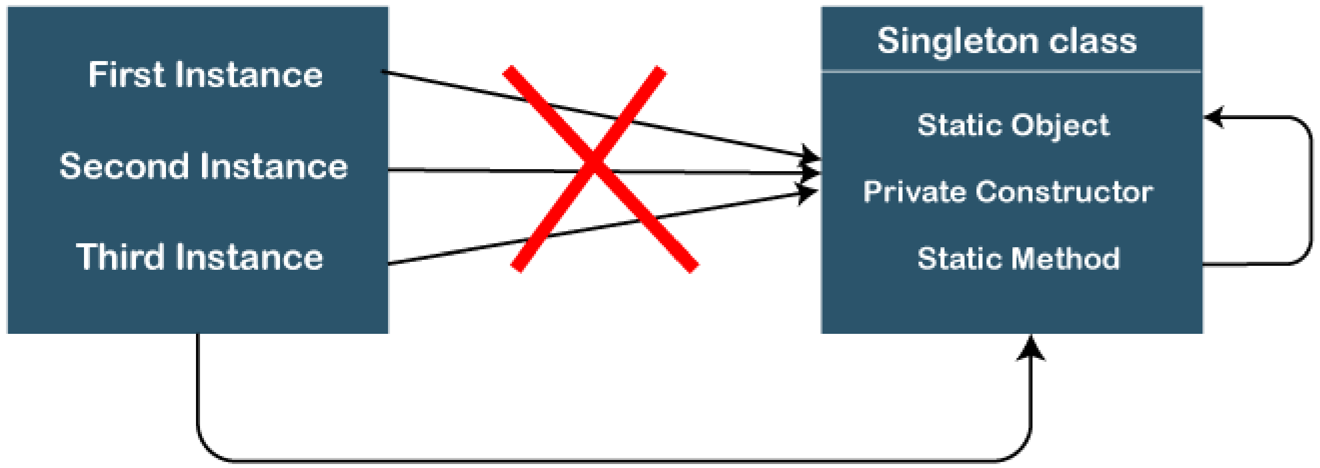
- *The del method thus invokes the __del__() special method.*

Here's a complete guide to special methods :-

http://www.informit.com/articles/article.aspx?p=453682&seqNum=6

Singleton Design Pattern

Singleton design pattern is one of the Credential Design Pattern and it is easiest to implement. As the name describes itself, it is a way to provide one object of a particular type. It is used to describe the formation of a single instance of class while offering a single global access point to the object.



It prevents the creation of multiple objects of the single class. The newly created object will be shared globally in an application. We can understand it with the simple example of Data connectivity. While setting up the database connection, we generate an exclusive Database connection object to work with the Database. We can perform every operation regarding database using that single connection object. This process is called a **Single design pattern**.

Singleton Design Pattern

- Motivation
- **Singleton design patterns** are specially used in application types that need mechanisms over access to a mutual resource. As we have discussed earlier, a single class can be used to define a single instance.

One of the best benefits of using a singleton pattern is that we can restrict the shared resource and maintain integrity. It also prevents the overwriting in the current code by the other classes ensuing perilous or incompetent code.
We can call the same object at multiple points of programs without worrying that it may be overwritten in the same points.

Implementation

To implement the singleton pattern, we use the static method. We create the **getInstance()** method that can return the shared resources. When we call the static method, either it gives the unique singleton object or an error singling an instantiated object's existence. It restricts to create the multiple objects of a defined class and maintain integrity.
We can take an example of the simple analogy - A county has a single central government that controls and accesses the country's operation. No one can create another government in a certain period.
We can implement this analogy using the singleton pattern.

Factory Design Pattern

The Factory design pattern is a creational design pattern that provides a way to create objects without explicitly specifying the exact class to be instantiated. It centralizes object creation logic and allows for more flexibility and decoupling in your code.

Here's a deeper dive into the Factory pattern:

Core Concept:

Instead of directly creating objects using constructors, you delegate object creation to a factory class or function.

This factory takes care of deciding which concrete class to instantiate based on certain criteria (like type, configuration, or arguments).

Benefits:

Decoupling: Factory patterns decouple object creation from the code that uses the objects. This makes your code more flexible and easier to test. You can modify the factory logic without affecting the code that consumes the objects.

Flexibility: Adding new types of objects becomes easier. You simply extend the factory to handle the new object type without modifying existing code.

Open/Closed Principle: The factory adheres to the Open/Closed Principle, where you can extend the functionality (adding new object types) without modifying existing code.

Stimulants

1. _____ represents an entity in the real world with its identity and behavior.
a) A method
b) An object
c) A class
d) An operator
2. _____ is used to create an object.
a) class
b) constructor
c) User-defined functions
d) In-built functions

3.What is the output of the following code?

```
class test:
     def __init__(self,a="Hello World"):
         self.a=a
    def display(self):
         print(self.a)
obj=test()
obj.display()
```

a) The program has an error because constructor can’t have default arguments
b) Nothing is displayed
c) “Hello World” is displayed
d) The program has an error display function doesn’t have parameters

4. What is setattr() used for?
a) To access the attribute of the object
b) To set an attribute
c) To check if an attribute exists or not
d) To delete an attribute

5. What is getattr() used for?
a) To access the attribute of the object
b) To delete an attribute
c) To check if an attribute exists or not
d) To set an attribute

6. What is the output of the following code?

```
class Demo:
    def __init__(self):
        pass
    def test(self):
        print(__name__)
obj = Demo()
obj.test()
```

a) Exception is thrown
b) __main__
c) Demo
d) test

7.Is the following piece of code valid?

```
class B(object):
  def first(self):
    print("First method called")
  def second():
    print("Second method called")
ob = B()
B.first(ob)
```

a) It isn’t as the object declaration isn’t right
b) It isn’t as there isn’t any __init__ method for initializing class members
c) Yes, this method of calling is called unbounded method call
d) Yes, this method of calling is called bounded method call.

Acknowledgements :-

1. Digital Ocean Tutorials
2. Tutorial point website
3. You tube