Outline

- Class
- Object
- Self-Variables
- Constructors
- Types of Methods
- Access Modifiers

Python Object Oriented Programming

- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - **attributes**
  - **behavior**

Ex: A parrot is an object, as it has the following properties:
name, age, color as attributes
singing, dancing as behavior

Python Object Oriented Programming
(continued)

- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
- In Python, the concept of OOP follows some basic principles:
  - **Class**
  - **Object**
  - **Methods**
  - **Inheritance**
  - **Encapsulation**
  - **Polymorphism**

Python Object Oriented Programming
(continued)

Class

- A class is a blueprint for the object.
- We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.
- The example for class of parrot can be :

```
class Parrot:
    pass
```

Python Object Oriented Programming
(continued)

Object

- An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- The example for object of parrot class can be:

```
obj = Parrot()
```

Python Object Oriented Programming (continued)

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

1

2

3

4

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Somaiya TRUST

Python Object Oriented Programm (continued)

Methods

- Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

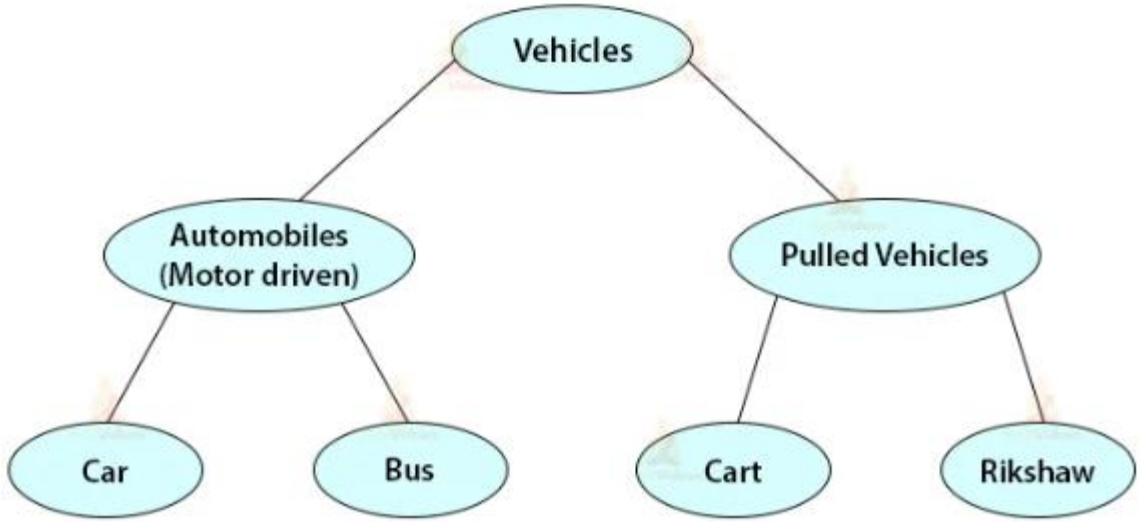
Output

```
Blu sings 'Happy'
Blu is now dancing
```

Python Object Oriented Programming
(continued)

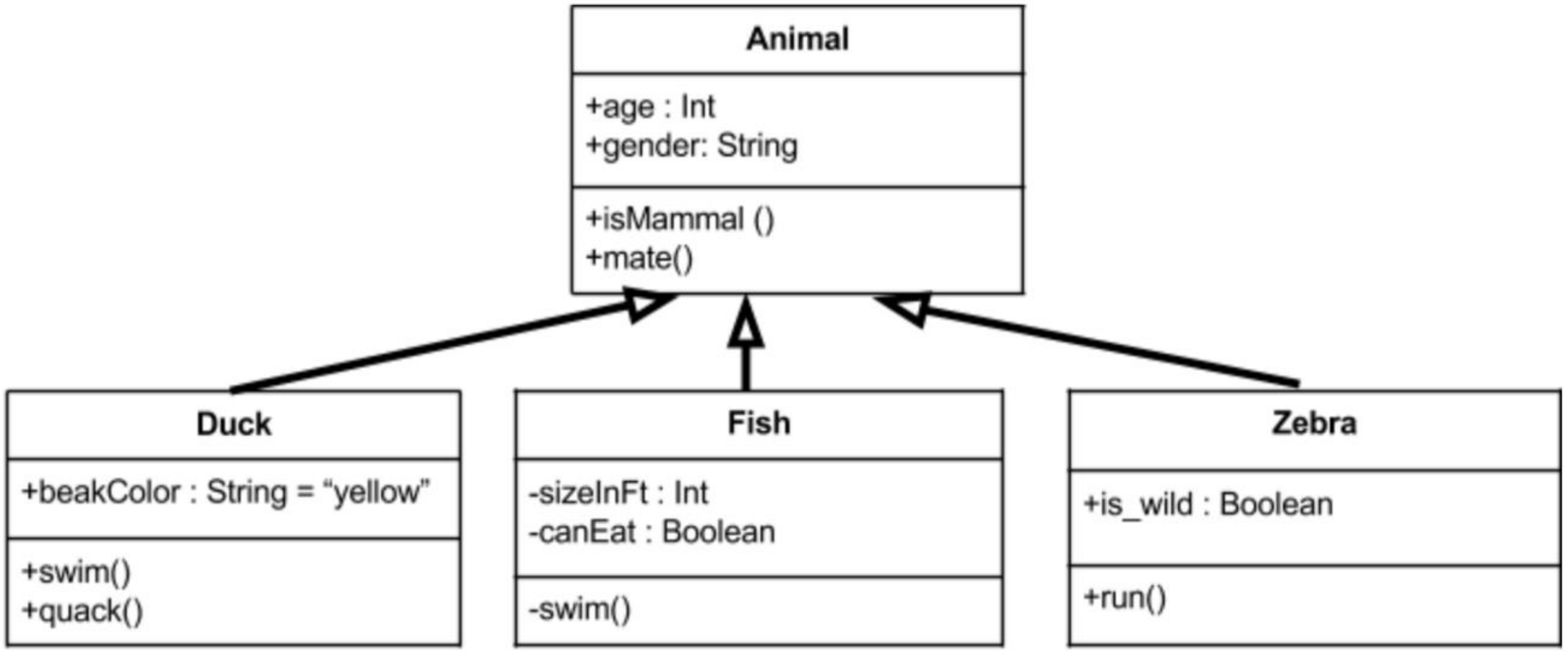
Inheritance

- Inheritance is a way of creating a new class for using details of an existing class without modifying it.
- The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).



Python Object Oriented Programming
(continued)

Inheritance

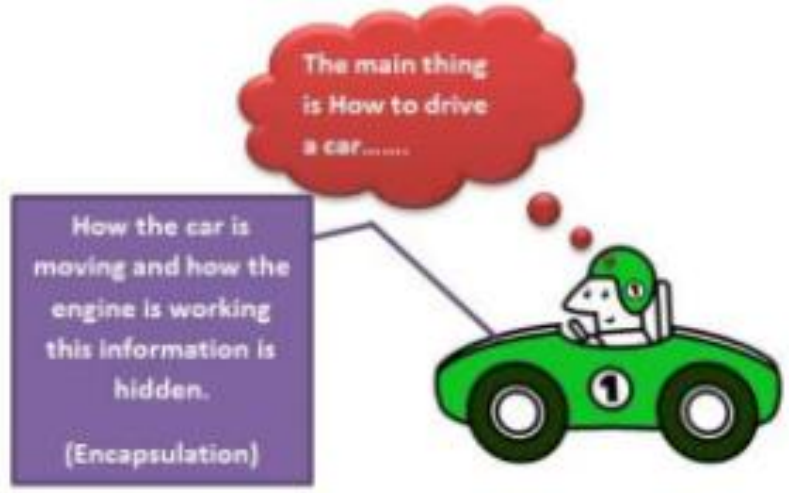


Somaiya TRUST

Python Object Oriented Programming
(continued)

Encapsulation

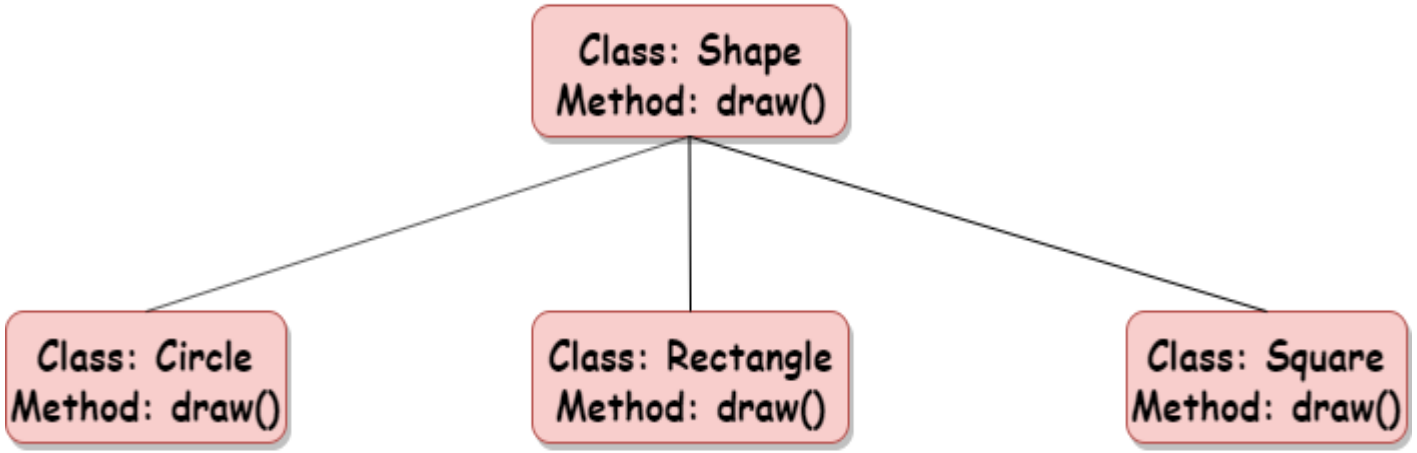
- Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation.
- In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.



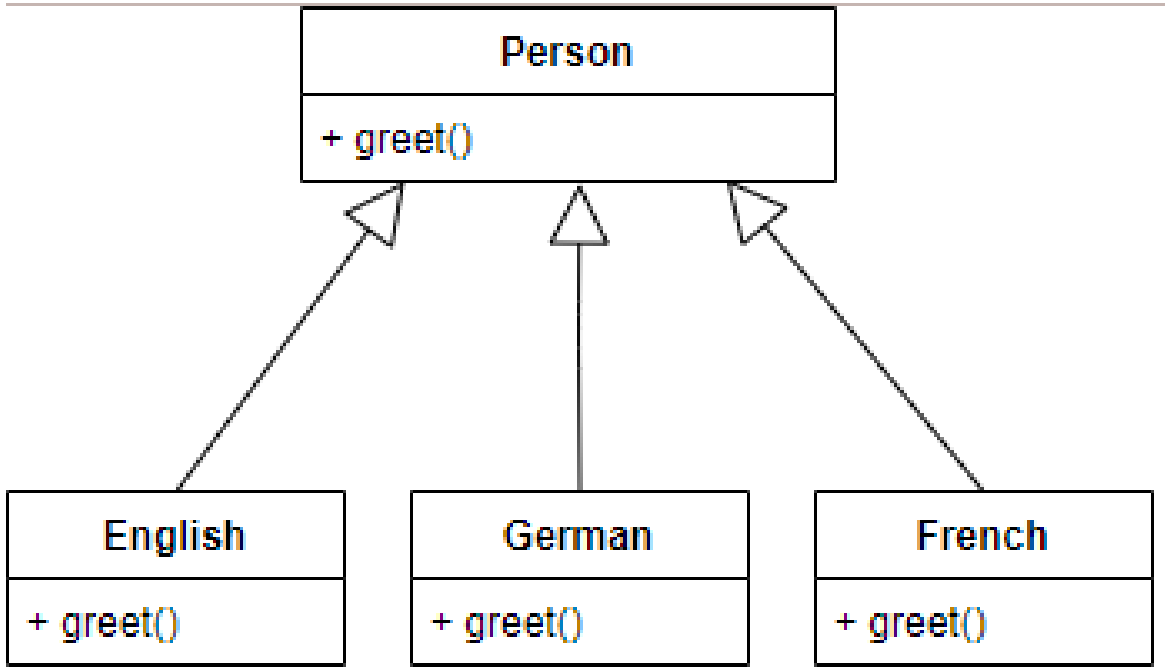
Python Object Oriented Programming
(continued)

Polymorphism

- Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).
- Suppose, we need to draw a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to draw any shape. This concept is called Polymorphism.



Python Object Oriented Programming
(continued)



Python Objects and Classes

- Python is an object-oriented programming language.Like procedure-oriented programming, where the main emphasis is on **functions**, object-oriented programming stresses on **objects**.
- An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.
- We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.
- As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class in Python

- Like function definitions begin with the **def** keyword in Python, class definitions begin with a **class** keyword.
- The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

Defining a Class in Python (continued)

- Once we define a class, a new class object is created with the same name.
- This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person class
```

Creating an Object in Python (continued)

- The class object could be used to access different attributes.
- It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> harry = Person()
```

Creating an Object in Python (continued)

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

Somaiya TRUST

Constructors in Python

- Class functions that begin with double underscore __ are 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.
- This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Constructors in Python (continued)

```python
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')


# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method
# Output: 2+3j
num1.get_data()

# Create another ComplexNumber object
# and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

# Output: (5, 0, 10)
print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

Output

```
2+3j
(5, 0, 10)
Traceback (most recent call last):
  File "<string>", line 27, in <module>
    print(num1.attr)
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object num2 and read it as well. But this does not create that attribute for object num1.

Deleting Attributes and Objects

- Any attribute of an object can be deleted anytime, using the del statement

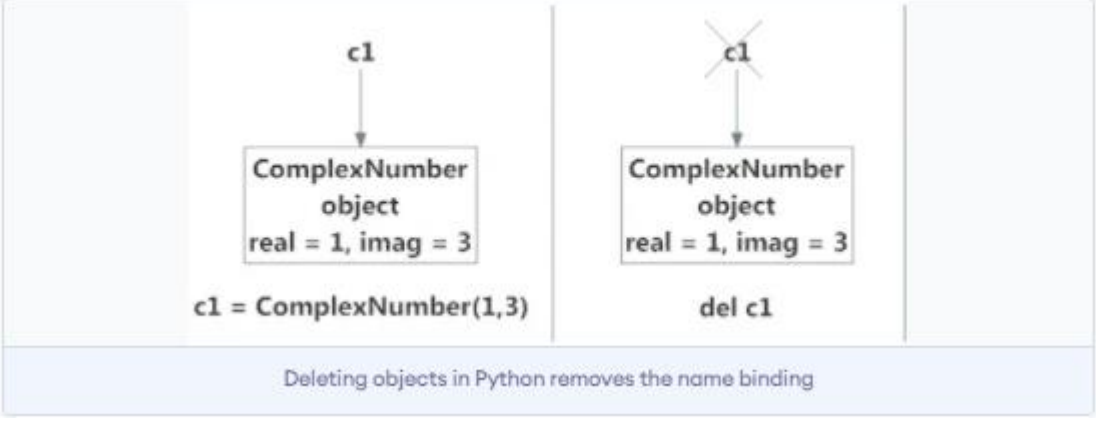
```
>>> num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

Deleting Attributes and Objects (continued)

- On the command del c1, this binding is removed and the name c1 is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
- This automatic destruction of unreferenced objects in Python is also called garbage collection.

We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```



Types of Methods

- There are three types of methods in Python:
  1. Instance Methods.
  2. Class Methods
  3. Static Methods

Types of Methods continued

Some key concepts before types of methods:

- **Class Variable:** A class variable is nothing but a variable that is defined outside the constructor. A class variable is also called as a **static variable**.
- **Accessor(Getters):** If you want to fetch the value from an instance variable we call them accessors.
- **Mutator(Setters):** If you want to modify the value we call them mutators.

1. Instance Method

- This is a very basic and easy method that we use regularly when we create classes in python.
- If we want to print an instance variable or instance method we must create an object of that required class.
- If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.
- As we are working with instance variables we use self keyword.
- **Note**: Instance variables are used with instance methods.

1. Instance Method continued

- Example:

```
# Instance Method Example in Python
class Student:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def avg(self):
        return (self.a + self.b) / 2

s1 = Student(10, 20)
print( s1.avg() )
```

Output:

```
15.0
```

- In the above program, a and b are instance variables and these get initialized when we create an object for the Student class.
- If we want to call avg() function which is an instance method, we must create an object for the class.
- If we clearly look at the program, the self keyword is used so that we can easily say that those are instance variables and methods.

2. Class Method

- classsmethod() function returns a class method as output for the given function.
- Here is the syntax for it:

classmethod(function)

- The classmethod() method takes only a function as an input parameter and converts that into a class method.
- There are two ways to create class methods in python:
  1. Using classmethod(function)
  2. Using @classmethod annotation

2. Class Method continued

Example 1: Create class method using classmethod()

```
class Person:
    age = 25

    def printAge(cls):
        print('The age is:', cls.age)

# create printAge class method
Person.printAge = classmethod(Person.printAge)

Person.printAge()
```

Output

```
The age is: 25
```

2. Class Method continued

Example 2: Create class method using @classmethod annotation

- As we are working with ClassMethod we use the cls keyword. Class variables are used with class methods.
- In the above example, name is a class variable.
- If we want to create a class method we must use @classmethod decorator and cls as a parameter for that function.

```
# Class Method Implementation in python
class Student:
    name = 'Student'
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @classmethod
    def info(cls):
        return cls.name

print(Student.info())
```

Output:

```
Student
```

3. Static Method

- A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.
- For example, If you want to print factorial of a number then we don't need to use class variables or instance variables to print the factorial of a number. We just simply pass a number to the static method that we have created and it returns the factorial.

3. Static Method continued

```
# Static Method Implementation in python
class Student:
    name = 'Student'
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @staticmethod
    def info():
        return "This is a student class"

print(Student.info())
```

Output

```
This a student class
```

Difference between Class and Static Methods

Class Method	Static Method
The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	@staticmethod decorator is used here.

Types of Methods continued

Key Takeaways

- Instance methods need a class instance and can access the instance through self.
- Class methods don’t need a class instance. They can’t access the instance (self) but they have access to the class itself via cls.
- Static methods don’t have access to cls or self. They work like regular functions but belong to the class’s namespace.

Access Modifiers

- In most of the object-oriented languages access modifiers are **used to limit the access to the variables and functions of a class**. Most of the languages use three types of access modifiers, they are - **private**, **public** and **protected**.
- Just like any other object oriented programming language, access to variables or functions can also be limited in python using the access modifiers.
- Python makes the use of **underscores** to specify the access modifier for a specific data member and member function in a class.

Access Modifiers continued

- Access modifiers play an important role to protect the data from unauthorized access as well as protecting it from getting manipulated.
- When inheritance is implemented there is a huge risk for the data to get destroyed(manipulated) due to transfer of unwanted data from the parent class to the child class.
- Therefore, it is very important to provide the right access modifiers for different data members and member functions depending upon the requirements.

Access Modifiers continued

- There are 3 types of access modifiers for a class in Python.
- These access modifiers define how the members of the class can be accessed.
- Of course, any member of a class is accessible inside any member function of that same class.

Access Modifier: Public

- The members declared as Public are accessible from outside the Class through an object of the class.

Access Modifier: Protected

- The members declared as Protected are accessible from outside the class but only in a class derived from it that is in the child or subclass.

Access Modifier: Private

- These members are only accessible from within the class. No outside Access is allowed.

public Access Modifier

- By default, all the variables and member functions of a class are public in a python program.

```
# defining a class Employee
class Employee:
    # constructor
    def __init__(self, name, sal):
        self.name = name;
        self.sal = sal;
```

All the member variables of the class in the above code will be by default public, hence we can access them as follows:

```
>>> emp = Employee("Ironman", 999000);
>>> emp.sal;
999000
```

protected Access Modifier

- According to Python convention adding a prefix _(single underscore) to a variable name makes it protected. Yes, no additional keyword required.

```
# defining a class Employee
class Employee:
    # constructor
    def __init__(self, name, sal):
        self._name = name;    # protected attribute
        self._sal = sal;      # protected attribute
```

In the code above we have made the class variables **name** and **sal** protected by adding an _(underscore) as a prefix, so now we can access them as follows:

```
>>> emp = Employee("Captain", 10000);
>>> emp._sal;
10000
```

protected Access Modifier continued

```
# defining a child class
class HR(Employee):

    # member function task
    def task(self):
        print ("We manage Employees")
```

```
>>> hrEmp = HR("Captain", 10000);
>>> hrEmp._sal;
10000
>>> hrEmp.task();
We manage Employees
```

private Access Modifier

- While the addition of prefix __(double underscore) results in a member variable or function becoming private.

```
# defining class Employee
class Employee:
    def __init__(self, name, sal):
        self.__name = name;    # private attribute
        self.__sal = sal;      # private attribute
```

If we want to access the **private** member variable, we will get an error.

```
>>> emp = Employee("Bill", 10000);
>>> emp.__sal;
```

```
OUTPUT:

AttributeError: 'employee' object has no attribute '__sal'
```

Example

```python
# define parent class Company
class Company:
    def __init__(self, name, proj):
        self.name = name      # name(name of company) is public
        self._proj = proj     # proj(current project) is protected
    # public function to show the details
    def show(self):
        print("The code of the company is = ",self.ccode)

# define child class Emp
class Emp(Company):
    def __init__(self, eName, sal, cName, proj):
        # calling parent class constructor
        Company.__init__(self, cName, proj)
        self.name = eName    # public member variable
        self.__sal = sal     # private member variable
    # public function to show salary details
    def show_sal(self):
        print("The salary of ",self.name," is ",self.__sal,)

# creating instance of Company class
c = Company("Stark Industries", "Mark 4")
# creating instance of Employee class
e = Emp("Steve", 9999999, c.name, c._proj)

print("Welcome to ", c.name)
print("Here ", e.name," is working on ",e._proj)
e.show_sal()
```

Output

```
Welcome to  Stark Industries
Here  Steve  is working on  Mark 4
The salary of  Steve  is  9999999
```

Getter and Setter

- The primary use of getters and setters is to ensure data encapsulation in object-oriented programs.
- We want to conceal an object class's attributes from other classes so that methods in other classes don't accidentally modify data.
- In OOPs languages, getters and setters are used to retrieve and update data.
- A getter retrieves an object's current attribute value, whereas a setter changes an object's attribute value.

Getter and Setter

What is Getter in Python?

- Getters are the methods that are used in Object-Oriented Programming (OOPS) to access a class's private attributes.

- What is Setter in Python?
  - The setter is a method that is used to set the property's value. It is very useful in object-oriented programming to set the value of private attributes in a class.

Getter and Setter

```
1  class Person:
2      def __init__(self, age = 0):
3          self._age = age
4        # using the getter method
5      def get_age(self):
6          return self._age
7        # using the setter method
8      def set_age(self, a):
9          self._age = a
10
11 John = Person()
12 John.set_age(19)    #using the setter function
13 print(John.get_age())    # using the getter function
14
15 print(John._age)
```

Output

```
19
19
```

Thank You!!