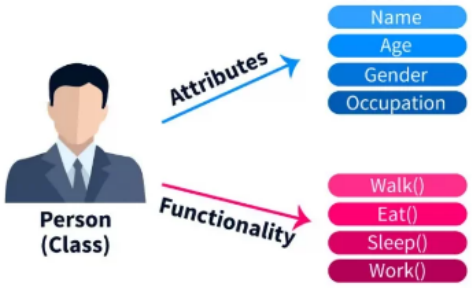
Day 3

Agenda:
Classes and objects, User-Defined Classes, Class Variables and Instance, Variables, Instance methods, Class method, static methods, constructor in python, parameterized constructor

Classes

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties(also called attributes) and methods.



- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.

Object

- Object is an entity that has a state and behavior associated with it.
- It may be any real-world object like a mouse, keyboard, chair, table, pen, etc.
- An object consists of:
    - State: It is represented by the attributes of an object.
    - Behavior: It is represented by the methods of an object.
    - Identity: It gives a unique name to an object.

Example: Create class and object

```
class Greeting:
    message = "Hello, World!"

    def say_hello(self):
        print(self.message)


# Create an object
greet = Greeting()
greet.say_hello()
```

```
Hello, World!
```

_ _init_ _() Method

- In python __init__() is a special method known as the constructor.
- The __init__() method is useful to initialize the variables of the class object.
- All classes have a method called __init__(), which is always executed when the class is being initiated.
- Use the __init__() Method to assign values to object properties, or other operations that are necessary to do when the object is being created.
- The __init__() Method is called automatically every time the class is being used to create a new object.

Example: Create a class named person with some attributes and properties.

```python
class Person:
    def __init__(self, name, age, gender): # __init__ is constructors.
        self.name=name
        self.age=age
        self.gender=gender
    def show_details(self):
        print('The name is {}, age is {} and gender is {}'.format(self.name,self.age,self.gender))
```

```python
# Create an object of Person class
p1=Person('Nimit', 35, 'Male')
p1.show_details()
```

```
The name is Nimit, age is 35 and gender is Male
```

Class Variables in Python

- Class Variables are declared inside the class definition (but outside any of the instance methods).
- They are not tied to any particular object of the class, hence shared across all the objects of the class.
- Modifying a class variable affects all object instances at the same time.

Instance Variable or Object Variable

- Object variable or instance variable owned by each object created for a class.
- Declared inside the constructor method of class (the __init__ method).
- They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.
- The object variable is not shared between objects.

Example:

```
class Car:
    wheel = 4    # <- Class variable
    def __init__(self, name):
        self.name = name    # <- Instance variable
```

```
mercedes=Car("mercedes")
print(mercedes.wheel) #Access of class variable through object
print(Car.wheel)  #Access of class variable through class
print(mercedes.name)  #Access of instance variable through object
```

```
4
4
mercedes
```

Example: Create a class name Employee with

i. a class variable office_name, and

ii. instance variables name and designation.

iii. Create a show_detail() method.

```
class Employee:
    office_name = "ABC Corporation" #Class variable
    def __init__(self,name,designation):
        self.name = name
        self.designation = designation
    def show_details(self):
        print(F" Company Name: {Employee.office_name}\n Name: {self.name} \n Designation: {self.designation}")
emp1 = Employee("Mohit", 35)
emp1.show_details()
```

```
 Company Name: ABC Corporation
 Name: Mohit
 Designation: 35
```

Difference Between Class and Instance Variables

Class variable	Instance variable
Class variable owned by class.	Instance variable owned by object.
All the objects of the class will share the class variable.	The instance variable is not shared between objects.
Any change made to the class variable by an object will be reflected in all other objects.	Any change made to the Instance variable by an object will not be reflected in all other objects.

- Generally , Class variable is used to define constant with a particular class or provide default attribute.
- Another use of class variable is to count the number of objects created.

self Parameter

- The self argument refers to the object itself.
- The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
- We can have __init__() with any number of parameters, but the first parameter will always be a variable called self.
- When a new class instance is created, the instance is automatically passed to the self parameter in __init__() so that new attributes can be defined on the object.

Use other name in Place of self

Example: In the following program we are using default and abc in place of self.

```
class Person:
    def __init__(default, name, age):
        default.name = name
        default.age = age
    def myfunc(abc):
        print("Hello my name is " + abc.name)


p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

More class Example:

Example: Program modifying mutable type attributes.

```
class Number:
    evens = []
    odds = []
    def __init__(self, num):
        self.num = num
        if num % 2 == 0:
            Number.evens.append(num)
        else:
            Number.odds.append(num)
N1 = Number(21)
N2 = Number(32)
N3 = Number(43)
N4 = Number(54)
N5 = Number(65)
print("even Numbers are:", Number.evens)
print("odd Numbers are:", Number.odds)
```

```
even Numbers are: [32, 54]
odd Numbers are: [21, 43, 65]
```

__str__() method

- __str__() defines what will get printed if we print an object.
- It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing object information.
- We can alter how objects of a class are represented in strings by defining the __str__() method.
- If we do not define the __str__() method inside the class and we print the object using print. Then the default __str__() method is called.

  Example: In the following example we are altering the __str__() method.

```python
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def __str__(self):      # This will get called when we print the object
        return F"The name is {self.name} and age is {self.age}"


p = Person("Amit",35)
print(p)
```

```
The name is Amit and age is 35
```

Methods in Python

- We can also define a function inside a Python class. A Python Function defined inside a class is called a method.
- In python there are three different types of methods
    1. **Instance Methods**
    2. **Class methods**
    3. **Static Methods**
- Each one of them has different characteristics and should be used in different situations.

Instance Methods or Object Methods

- They are the most widely used methods.
- **Instance method receives the instance of the class as the first argument, which by convention is called self, and points to the instance of class.**
- However it can take any number of arguments.
- Using the self parameter, we can access the other attributes and methods on the same object and can change the object state.
- Therefore, instance methods gives us control of changing the object as well as the class state.
- A built-in example of an instance method is str.upper()

Syntax:

```
class my_class:
    def function_name(self, arguments):
        #Function Body
        return value
```

Instance Method Example

```
class Employee:
    # __init__ method to initialize the object attributes
    def __init__(self, name, position, salary):
        self.name = name
        self.position = position
        self.salary = salary

    # Instance method to display employee details
    def display_details(self):
        print(f"Employee Details:\nName: {self.name}\nPosition:
{self.position}\nSalary: {self.salary}")

    # Instance method to give a raise to the employee
    def give_raise(self, amount):
        self.salary += amount
        print(f"{self.name} received a raise of {amount}. New salary is
{self.salary}")

    # Instance method to update the employee's position
    def update_position(self, new_position):
        self.position = new_position
        print(f"{self.name}'s new position is {self.position}")
```

```
# Creating an instance of Employee
employee_1 = Employee("Alice", "Software Engineer", 70000)

# Displaying employee details using an instance method
employee_1.display_details()

# Giving a raise to the employee using an instance method
employee_1.give_raise(5000)

# Updating the employee's position using an instance method
employee_1.update_position("Senior Software Engineer")

# Displaying updated employee details
employee_1.display_details()
```

Employee Details:
Name: Alice
Position: Software Engineer
Salary: 70000
Alice received a raise of 5000. New salary is 75000
Alice's new position is Senior Software Engineer
Employee Details:
Name: Alice
Position: Senior Software Engineer
Salary: 75000

Class Method

- A class method is a method that is bound to a class rather than its object.
- It doesn't require creation of a class instance.
- A class method receives the class as an implicit first argument, just like an instance method receives the instance.
- **Class methods can be called by both class and object.**

- We use **@classmethod** decorator in Python to create a class method.
- Class methods are bound to the class and not to the object of the class. They can alter the class state that would apply across all instances of class but not the object state.

Note:

- Class methods are called by class.
- First argument of the class method **cls** , not the **self**.

Syntax:

```
class my_class:
    @classmethod
  def function_name(cls, arguments):
      #Function Body
      return value
```

Class Method Example

Write a program to count how many objects of a class are created.

```
class Student:
    counter = 0
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
        Student.counter = Student.counter + 1
    def msg(self):
        print("Hello "+self.name + " you got ", self.marks, "% marks")
    @classmethod
    def object_count(cls):
        return cls.counter
s1 = Student("Abhishek", 65)
s2 = Student("Amit", 89)
print(Student.object_count())
print(s1.object_count())
```

```
2
2
```

Static Method

- A static method is marked with a **@staticmethod** decorator to flag it as static.
- It does not receive an implicit first argument **(neither self nor cls)**.
- It can also be put as **a method that "doesn't know its class"**.
- **A static method can be called by the class or an instance(object).**
- **Static method can be called without an object of that class.**
- Hence **static methods can neither modify the object state nor class state.** They are primarily a way to namespace our methods.

  Syntax:

```
class my_class:
    @staticmethod
    def function_name(arguments):
        #Function Body
        return value
```

Static Method Example

```
class MathOperations:
    @staticmethod
    def add_numbers(a, b):
        return a + b

    @staticmethod
    def subtract_numbers(a, b):
        return a - b

obj = MathOperations()
result_add = MathOperations.add_numbers(10, 5) #called by className
result_subtract = obj.subtract_numbers(10, 5)  #called by objectName

print(f"Addition: {result_add}")
print(f"Subtraction: {result_subtract}")
```

```
Addition: 15
Subtraction: 5
```

Class Methods vs 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.

Constructor in Python

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.
- In Python, the **__init__()** method is called the constructor and is always called when an object is created.

Syntax:

```
def __init__(self):
    # body of the constructor
```

Types of Constructor

Constructor can be **default constructor** or **parameterised constructor**.

Default constructor:

- The default constructor is simple constructor which doesn't accept any arguments.
- It's definition has only one argument which is a reference to the instance being constructed.

Parameterized constructor:

- Constructor with parameters is known as parameterized constructor.
- The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example of Default Constructor

```
class Student:
    # Default Constructor
    def __init__(self):
        print("This is a default constructor")
    def show(self, name):
        print("Hello", name)
student = Student()
student.show("Amit")
```

```
This is a default constructor
Hello Amit
```

Parameterised Constructor Example

```
class Addition:
    first = 0
    second = 0
    answer = 0
    def __init__(self, f, s):  #parameterized constructor
        self.first = f
        self.second = s
    def display(self):
        print("First number = ", self.first)
        print("Second number = ", self.second)
        print("Addition of two numbers = ", self.answer)
    def calculate(self):
        self.answer = self.first + self.second
obj = Addition(1000, 2000)
obj.calculate()
obj.display()
```

```
First number =  1000
Second number =  2000
Addition of two numbers =  3000
```

More than One Constructor in Class

- Internally, **the object of the class will always call the last constructor if the class has multiple constructors**.
- **The constructor overloading is not allowed in Python**.
- In the following code, the object st called the second constructor whereas both have the same configuration.
- The first method is not accessible by the st object.

```
class Student:
    def __init__(self):
        print("The First Constructor")
    def __init__(self):
        print("The second Constructor")
st = Student()
```

```
The second Constructor
```

Built-in Class Functions

All classes in Python have some default built-in class functions.

Function	Description	Syntax
getattr(obj,name)	It is used to access the attribute of an object.	getattr(obj, 'answer')
setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.	setattr(obj, 'answer', 100)
delattr(obj, name)	It is used to delete a specific attribute.	delattr(obj, 'answer')
hasattr(obj, name)	It returns true if the object contains some specific attribute.	hasattr(obj, 'name')

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create an instance of Person
person = Person("Alice", 30)

# Use getattr to get the value of an attribute
name = getattr(person, "name")
print(f"Name: {name}")  # Output: Name: Alice

# Use setattr to set the value of an attribute
setattr(person, "age", 31)
print(f"Updated Age: {person.age}")  # Output: Updated Age: 31

# Use hasattr to check if an attribute exists
has_name = hasattr(person, "name")
print(f"Has attribute 'name': {has_name}")  # Output: Has attribute 'name': True

# Use delattr to delete an attribute
delattr(person, "age")
# Check if the attribute 'age' still exists after deletion
has_age = hasattr(person, "age")
print(f"Has attribute 'age' after deletion: {has_age}")  # Output: Has attribute 'age' after deletion: False
```

```
Name: Alice
Updated Age: 31
Has attribute 'name': True
Has attribute 'age' after deletion: False
```