Object Oriented Programming 🔴

💡 ***Important for interviews***

- Before creating an object we create a class of that object.
- Object examples- lists, strings
- Class is a blueprint for creating objects.

Class & Object in Python

- Class is a blueprint for creating objects.

Creating Class

```
class Student:
    name= “karan”
```

- We start the class name with capital letter

Creating Object

- Object is also called instance.

`s1 = Student( )`

```
class Student:
    name = "karan"

s1= Student ()
print(s1)

Output:
<__main__.Student object at 0x0000028FB2B85A90>
```

```
class Teachers:
    pass

teacher1= Teachers()

teacher1.name= "Vikas"
teacher1.city= "Delhi"
print(teacher1.name)

Output:
Vikas
```

- For now, we don’t have anything to write in class, so we wrote pass
- To print the name, we can’t just write name.. we have to specify **whose name**.
- ***`teacher1.name`*** → name of teacher1
- Teacher1 from class Teachers have following attributes
    - name= Vikas
    - city=Delhi

- But if we have 10 Teachers, then we will have to write name, city, mobile number for every teacher and that will be a tedious task.

- Therefore we define these variables in **class** and we can call the funcion.

```
class Student:
    name = "karan"

s1= Student ()
print(s1.name)

s2 = Student ()
print(s2.name)

Output:
karan
karan
```

Another example:

```
class Car:
    color = "blue"
    brand = "tata"

car1 = Car()
print(Car.color)

car2= Car()
print (Car.brand)

Output:
blue
tata
```

__*init* __ Function

Constructor

- All classes have a function called **init**(), which is always executed when the object is being
  initiated.
- Invoked (executed) at the time of object creation.

```
class Student:
    name = "karan"
    def __init__(self):
        print("Adding new student")

s1= Student()
s2= Student()

Output:
Adding new student
Adding new student
```

- We have 2 objects- s1 & s2. Therefore the __init__ function was called 2 times and Adding new student was printed twice.

`self` refers to the object being created.

- Let's print `self`:

```
class Student:
    name = "karan"
    def __init__(self):
        print (self)
        print("Adding new student")

s1= Student()

Output:
```

```
<__main__.Student object at 0x00000160AB3A5C10>
Adding new student
```

- The newly created object `s1` and `self` both are same.
- Using “self” is not necessary. You can call it anything.
- It’s just an alias for “Object”.
    - `s1` = `self`

```
class Student:
    def __init__(self, fullname):
        self.name= fullname

s1= Student("Vijay")
print(s1.name)

Output:
Vijay
```

- “name” can be anything.
- `fullname` refers to the thing we entered in s1 i.e. *Vijay*

Add another student:

```
class Student:
    def __init__(self, fullname):
        self.name= fullname
        print("Adding new student")

s1= Student("Vijay")
print(s1.name)
```

```
s2= Student("Jeevan")
print (s2.name)

Output:
Adding new student
Vijay
Adding new student
Jeevan
```

- This stored data is known as Attribute (Variable).

```
class Student:
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks
        print("Adding new student")

s1= Student("Vijay", 98)
print(s1.name,s1.marks)


s2= Student("Jeevan", 65)
print (s2.name, s2.marks)

Output:
Adding new student
Vijay 98
Adding new student
Jeevan 65
```

- `self.name` = `s1.name`

Constructor with only 1 parameter is called **default constructor**:

```
def __init__(self):
    print("Adding new student")
```

- If we don’t create this, python will automatically create it.

Parameterized constructor:

```
def __init__(self, name, marks):
    self.name= name
    self.marks = marks
    print("Adding new student")
```

Default Parameter:

```
class Students:
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks
        self.followers= 0

studen1 = Students ("rahul", 89)

print (studen1.followers)

Ooutput: 0
```

Here, followers is a default value. For everyone, the value of followers will be 0 by default.

We can change it later .

How to change the default value later for individual objects?

```
# `Change the followers value`
`studen1.followers = 10  # Update followers to 10`

# `Print the updated number of followers`
`print(studen1.followers)  # Output: 10
```

Class & Instance Attributes

2 Types:

1. Class- Common for all class and object
    - Common for all
    - Ex. Name of College which is same for all students
    - Stored in memory for 1 time only.

    ```
    class Student:
        college= "ABCD"
        def __init__(self, name, marks):
            self.name= name
            self.marks = marks
            print("Adding new student")

    s1= Student("Vijay", 98)
    print(s1.college)
    print(Student.college)

    Output:
    Adding new student
    ABCD
    ABCD
    ```

2. Instance- Different for each object

  - Instance attributes are defined by `self.name`
  - It means every name is different.

```
class Student:
    college= "ABCD"
    name= "Unknown"
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks
        print("Adding new student")

s1= Student("Vijay", 98)
print(s1.name)

Output:
Adding new student
Vijay
```

- Although we have passed the value Unknown, it will print Vijay.
- Because **Object attribute > Class attribute**

METHODS:

- Class is a collection of 2 things:
  - Data (Attributes)
  - Methods
- Data= Your Properties
- Methods= what can you do?

- Methods= **Functions** that belong to objects

```
class Student:
    college= "ABCD"
    name= "Unknown"
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks

    def welcome (self):
        print ("Welcome Students")

s1= Student("Vijay", 98)
s1.welcome()

Output:
Welcome Students
```

```
class Instructors:
    def __init__(self, name, address):
        self.name= name
        self.address= address
    def display (self):
        print("Marshall")

Instructor1 = Instructors("Jimmy", "USA")
print (Instructor1.name)
print (Instructor1.display())

Output:
Jimmy
```

```
Marshall
None
```

- None is printed because of the `print (Instructor1.display())`
- If we remove the print, it will vanish

Same code- print Hi, I am “name”:

```
class Instructors:
    def __init__(self, name, address):
        self.name= name
        self.address= address
    def display (self):
        print(f"Hi, I am {self.name}")

Instructor1 = Instructors("Jimmy", "USA")
Instructor1.display()

Output:
Hi, I am Jimmy
```

Add subject to above code:

```
class Instructors:
    def __init__(self, name, address):
        self.name= name
        self.address= address
    def display (self, sub):
        print(f"Hi, I am {self.name} and tech {sub}")

Instructor1 = Instructors("Jimmy", "USA")
Instructor1.display("Philosophy")
```

```
Output:
Hi, I am Jimmy and tech Philosophy
```

- Here, we don’t need to write self.sub.. `sub` will print the subject.
  - Because sub is not an attribute.
  - Name and address are the attributes, therefore it is mandatory to write `self.name`

```
class Student:
    college= "ABCD"
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks

    def welcome (self):
        print ("Welcome Student", self.name)

s1= Student("Vijay", 98)
s1.welcome()

Output:
Welcome Student Vijay
```

```
class Student:
    college= "ABCD"
    def __init__(self, name, marks):
        self.name= name
        self.marks = marks

    def welcome (self):
        print ("Welcome Student", self.name)

    def getmarks (self):
        return self.marks

s1= Student("Vijay", 98)
s1.welcome()
print (s1.getmarks())


Output:
Welcome Student Vijay
98
```

QUIZ

- Create student class that takes name & marks of 3 subjects as arguments in constructor.
  Then create a method to print the average.

  ▼

  ```
  class Student:
      def __init__(self, name, marks):
          self.name= name
          self.marks= marks

      def avg (self):
          sum= 0
          for i in self.marks:
              sum+=i
          print(f"Hi {self.name}, your average score is {sum/3}")

  s1 = Student("Tony", [55, 38, 87])
  s1.avg()
  ```

```
Output:
Hi Tony, your average score is 60.0
```

```
class Student:
    def __init__(self, name, marks):
        self.name= name
        self.marks= marks

    def avg (self):
        sum= 0
        for i in self.marks:
            sum+=i
        print(f"Hi {self.name}, your average score is {sum/3}")

s1 = Student("Tony", [55, 38, 87])
s1.avg()

s1.name= "Iron"
s1.avg()

Output:
Hi Tony, your average score is 60.0
Hi Iron, your average score is 60.0
```

Static Methods

- Methods that don’t use the self parameter (work at class level)
- These belong **class** level.
- You can call them directly on the class name without creating an object.
- They don't have access to instance (`self`) or class (`cls`) variables.
- **No Access to Instance or Class**: Static methods **don't take** `self` or `cls` as their first parameter, meaning they can't access instance-specific or class-specific

data.

`@staticmethod` → This is decorator

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

    @staticmethod
    def ola (): # We haven't written "self" inside the parenthesis
        print ("Oo la lala la aeee O")

s1= Person("Alex")

s1.ola ()
Person.ola()

Output:
Oo la lala la aeee O
Oo la lala la aeee O
```

4 Pillars of OOP:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction

- Hiding the implementation details of a class and only showing the essential features to the user.

- Abstract- Hidden
- Unnecessary things are hidden. Only the important things are shown to the user.
- **It is generalising things to hide the complex details.**

```
class Car:
    def __init__(self):
        self.acc = False
        self.brk = False
        self.clutch = False

    def start (self):
        self.clutch= True
        self.acc = True
        print ("Car startrd")

car1 = Car()
car1.start ()

Output:
Car startrd
```

Encapsulation

- Wrapping data and functions into a single unit (object).
- The attributes (`acc`, `brk`, `clutch`) are encapsulated within the `Car` class. This means they are not accessible directly from outside the class.

- We make a capsule of data and related functions.

> **Encapsulation** means **wrapping data and methods** together inside a class and **restricting direct access** to some of the object’s components.

Private Members (Weak Encapsulation)

- Prefix a variable/method with `_` (single underscore) to indicate **"protected"** (internal use).
- Prefix with `__` (double underscore) to **name-mangle** (makes it harder to access accidentally).

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # private variable

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
print(account.get_balance())  # ✅ Works (output: 1000)
print(account.__balance)      # ❌ Fails (AttributeError)
```

`self.__balance = balance`	**Private variable** (cannot access directly from outside)

🔒 Access Control in Python

Symbol	Access Level	Example
`public`	Anyone can access	`self.name`

Symbol	Access Level	Example
`_protected`	Internal use only	`self._salary`
`__private`	Class-only access	`self.__balance`

Getter & Setter Methods

- Getter retrieves values from private attributes.
- Setter safely sets/updates those values.
- Both help control access to internal class variables (encapsulation).

🎯 WHY DO WE NEED THEM?

When variables are **private** (like `__balance`), you **can’t access or change them directly**.

You must use **getters** and **setters** for **safe, controlled access**.

```
class Person:
    def __init__(self, name,age):
        self.__name=name #Private access modifier or variable
        self.__age=age #Private variable

    #Getter method for name
    def getname(self):
        return self.__name

        #Setter method for name
    def setname(self,name):
        self.__name=name
```

- `name` will be accessible inside the class only
- To **access** `name`, we use `getname(self)` (Getter)
- Modify name with the help of setter

    - `setname(self,name)`

```
person= Person("Jon", 38)
person.getname()

'Jon'
```

- 👆 Get name

Set Name:

```
person.setname('Moxley')
person.getname()

'Moxley'
```

QUIZ (IMP)

- Create Account class with 2 attributes - balance & account no. Create methods for debit, credit & printing the balance.

    ▼

    ```
    class Account:
        def __init__(self, balance, acc_no):
                self.balance = balance
                self.acc_no = acc_no

        def debit(self,n):
            self.balance -= n
            print(f"Rs.{n} debited. Current balance= {self.balance}")

        def credit (self, n):
    ```

```
        self.balance += n
        print(f"Rs.{n} credited. Current balance= {self.balance}")


    def getbal (self):
        return self.balance

acc1= Account(10000, 635498765784)
acc1.debit(1000)
acc1.credit (2000)
print (acc1.getbal())

Output:
Rs.1000 debited. Current balance= 9000
Rs.2000 credited. Current balance= 11000
11000
```

Delete (del)

- delete object properties or the object itself

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

s1= Person("Alex")

print(s1.name)

del s1.name

print(s1.name)
```

```
Output:
Alex
Error: 'Person' object has no attribute 'name'
```

Private(like) attributes and methods

- Private attributes are a way to restrict access to certain parts of a class, making them less visible to the outside world.
- Naming convention:
  - prefixing its name with an underscore (`_`). For example, `_name`.
- True Private:
  - two underscores (`__`)
  - For example, `__name`
- Private attributes are not accessible beyond class.

```
class Account:
    def __init__ (self, acc_no, passd):
        self.acc_no = acc_no
        self.__password= passd

acc1 = Account(12345, "abcde")

print(acc1.acc_no)
print(acc1.password)

Output:
12345
error: 'Account' object has no attribute 'password'.
```

- Here, account number got printed but not password as we have added __ prefix before password

BUT had we written a single underscore (_)→ `self._password= passd` , `print(acc1._password)` This would have printed the password.

```
class Person:
    __name = "Unk"

p1= Person ()
print(p1.__name)

Output: Error
```

```
class Person:
    __name = "Unk"

    def __hello (self):
        print ("Hello person")

    def welcome (self):
        self.__hello ()

p1= Person ()

print(p1.welcome()) # ✔

Output:
Hello person
None

print(p1.__hello()) #❌

Output:
Error: 'Person' object has no attribute '__hello
```

- `Welcome` can call `__hello`
- `p1` can call `Welcome`
- But `p1` can’t call `__hello` because p1 is outside the class.
- The reason you see `None` in the output is that the `welcome` method does not have a `return` statement.
    - To avoid printing `None`, you can simply call `p1.welcome()` without the `print()` statement:

INHERITANCE - IMP

- When we pass things from one class to another.

```
class Car:
    ...
class Toyota (Car):
```

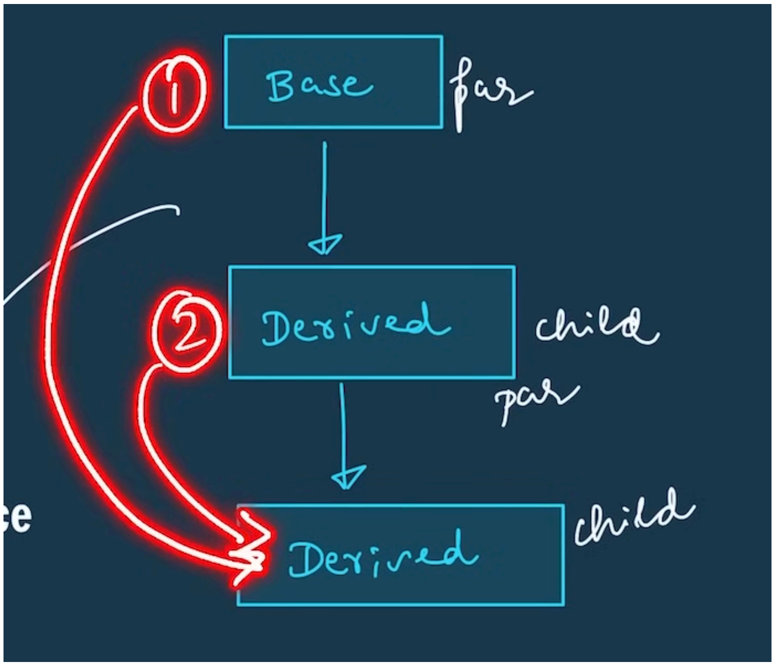
- Here, `Car` is the Parent class/ base class
- `Toyota` is the child class/ derived class

```
class Car:
    @staticmethod
    def start ():
        print ("Car started")
    @staticmethod
    def stop ():
          print ("Car stopped")

class Toyota (Car):
     def __init__(self, name):
          self.name= name

car1= Toyota ("Fortuner")
```

```
car2= Toyota ("Prius")

print(car1.name)
car2.start()

Output:
Fortuner
Car started
```

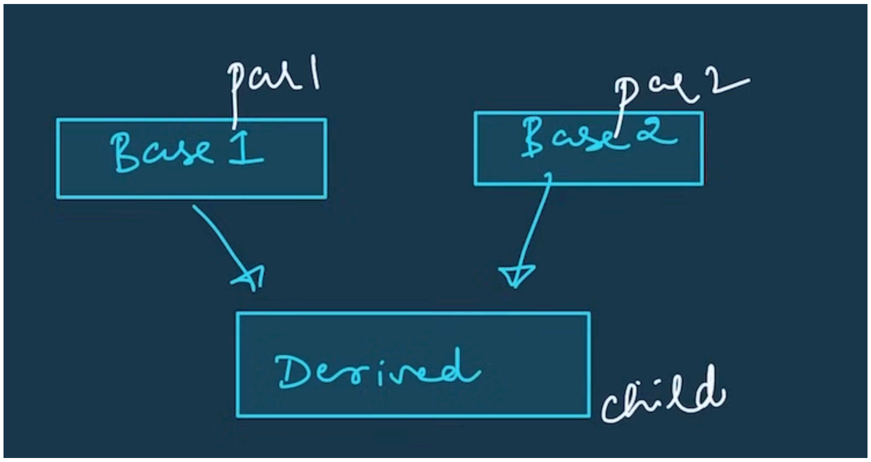
Type of Inheritance:

- Single Inheritance
    - 1 base class at 1 child class
- Multi-Level Inheritance
    - 1 base class and multiple child classes
    - Properties of both the classes go into the 3rd one.



```
class Car:
    @staticmethod
    def start ():
        print ("Car started")
    @staticmethod
    def stop ():
          print ("Car stopped")


class Toyota (Car):
     def __init__(self, brand):
          self.brand= brand


class fortuner (Toyota):
     def __init__(self, type):
          self.type= type

car1= fortuner ("deasel")
car1.start()

Output:
Car started
```

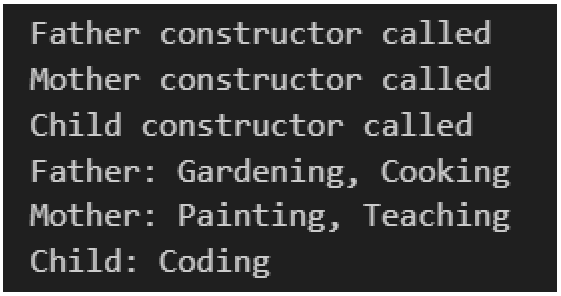
- Multiple Inheritance
    - 1 derived class can inherit properties of multiple classes.



```
class A:
    var1= "Welcome to class A."

class B:
    var2= "Welcome to class B."

class C (A,B):
    var3= "Welcome to class C."

c1= C()
print (c1.var2)

Output:
Welcome to class B.
```

```
# First parent class
class Father:
    def __init__(self):
        print("Father constructor called")

    def skills(self):
        print("Father: Gardening, Cooking")
```

```
# Second parent class
class Mother:
    def __init__(self):
        print("Mother constructor called")

    def skills(self):
        print("Mother: Painting, Teaching")

# Child class inherits from both
class Child(Father, Mother):
    def __init__(self):
        # Call both parent constructors
        Father.__init__(self)
        Mother.__init__(self)
        print("Child constructor called")

    def skills(self):
        Father.skills(self)
        Mother.skills(self)
        print("Child: Coding")

# Create object
c = Child()
c.skills()
```



```
Father constructor called
Mother constructor called
Child constructor called
Father: Gardening, Cooking
Mother: Painting, Teaching
Child: Coding
```

- Each class has its own `__init__()` constructor.

- The child class **manually calls both parent constructors** using `ClassName.__init__(self)` — this is important in multiple inheritance.
- If you don’t call both, only the first parent's constructor (from left to right in inheritance) runs due to Python’s **Method Resolution Order (MRO)**.
    - If you write `super().__init__()` instead of `Father.__init__(self)` **and** `Mother.__init__(self)` **, only** `Father.__init__(self)` will be called.

💡 **Do no write `super().__init__()` in case of multiple inheritence.**

Super Method

- *super()* → Parent inside inheritance
- Accesses methods of the parent class.
- **Method Overriding:** If a child class defines a method with the same name as a method in its parent class, it overrides the parent's method.
- `super()` allows you to call the overridden parent method from within the child class.

```
class Car:
    def __init__(self, type):
        self.type= type

    @staticmethod
    def start ():
       print ("Car started")
    @staticmethod
    def stop ():
         print ("Car stopped")


class Toyota (Car):
     def __init__(self, name, type):
        super().__init__(type) #Gives type for the class “Car”.
        self.brand= name
        super().start()

car1= Toyota("Prius", "electric")
print(car1.type)



Output:
Car started
electric
```

- If we write `print(car1.type)` without writing `super().__init__(type)`, it will give error.
    - because we have to write `self.type` inside Toyota.. and we haven’t written it.
- If we write `self.type=type`, it will give us type attribute for Toyota. But we want type for the class “Car”.
    - Because this is the child class that overrides the Car class’s type method.
- So we have to call the constructor of the parent class inside Toyota.
    - And we do that with- `super().__init__(type)`

Another example

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

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def __init__(self, name, breed):
        # Calling the parent class's __init__ method
        👇
        super().__init__(name)
        self.breed = breed

    def speak(self):
        # Calling the parent class's speak method
        👇
        super().speak()
        print(f"The dog is a {self.breed}.")

# Create an instance of Dog
dog = Dog("Buddy", "Golden Retriever")
dog.speak()

Output:
Buddy makes a sound.
The dog is a Golden Retriever.
```

There are 2 statements under `dog.speak()`:

1. `print(f"{` `self.name` `} makes a sound.")` (from the parent class).
2. `print(f"The dog is a {self.breed}.")` from the Dog child class.

Class Methods

```
class Person:
    name="anonymous"
    def changename (self, name):
        self.name =name

p1= Person()
p1.changename ("Ajay")

print(p1.name)
print(Person.name)
```

```
Output:
Ajay
anonymus
```

- **Problem with above code:**
    - `print(p1.name)` prints the newly added name. However, when we call `print(Person.name)`, it prints “anonymous”.
    - Our aim was to change the “anonymous” to → “Ajay”
- @staticmethod cannot access or modify class state.
- Class method is bound to class and receives the class as an implicit first argument.

1 way to change the “anonymous” to → “Ajay”:

change the `self.name =name` → `Person.name =name`

```
class Person:
    name="anonymus"
    def changename (self, name):
        Person.name =name

p1= Person()
p1.changename ("Ajay")

print(p1.name)
print(Person.name)

Output:
Ajay
Ajay
```

2nd Method:

- Write `__class__`
- We will be able to access the class by writing this

```
class Person:
    name="anonymus"
    def changename (self, name):
        self.__class__.name= "Ajay Menon"

p1= Person()
p1.changename ("Ajay")

print(p1.name)
print(Person.name)

Output:
Ajay Menon
Ajay Menon
```

- When you call `p1.changename("Ajay")`, you are not setting an instance variable. Instead, you are modifying the class variable `name`, which is why both `print(p1.name)` and `print(Person.name)` yield `"Ajay Menon"`.

- `self.__` **`class__`** `.name` :
    - `self`: Refers to the current object (`p1` in this case).
    - `.__class__`: Accesses the class of the current object (`Person`).
    - `.name`: Assigns the new value "Ajay Menon" to the class attribute `name`.

3rd Method: ***@classmethod:***

```
class Person:
    name="anonymus"
    @classmethod
    def changename(cls, name):
        cls.name=name

p1= Person()
p1.changename ("Ajay")

print(p1.name)
print(Person.name)

Output:
Ajay
Ajay
```

- Instead of self, we write- `cls`
- `cls.name` changes the class attribute directly

Property

- @property
- `@property` decorator is used to transform a method into a property.

This is normal formula to calculate percentage or Average:

```
class Student:
    def __init__(self, phy, chem, math):
        self.phy = phy
        self.chem=chem
        self.math=math
        self.percent= str ((self.phy+self.chem+self.math)/3) +"%"

stu1 = Student(80,68,98)
print(stu1.percent)
```

```
Output:
82.0%
```

- If we want to change the marks later:

```
class Student:
    def __init__(self, phy, chem, math):
        self.phy = phy
        self.chem=chem
        self.math=math
        self.percent= str ((self.phy+self.chem+self.math)/3) +"%"

stu1 = Student(80,68,98)
print(stu1.percent)

stu1.phy=88
print(stu1.phy)
print(stu1.percent)

Output:
82.0%
88
82.0%
```

- The marks change but percentage don’t.
- To change the percentage, we can create a method

```
class Student:
    def __init__(self, phy, chem, math):
        self.phy = phy
        self.chem=chem
        self.math=math
        self.percent= str ((self.phy+self.chem+self.math)/3) +"%"
    def calper (self):
        self.percent= str ((self.phy+self.chem+self.math)/3) +"%"

stu1 = Student(80,68,98)
print(stu1.percent)

stu1.phy=88
print(stu1.phy)

stu1.calper()
print(stu1.percent)

Output:
82.0%
88
84.66666666666667%
```

- This updates the percentage.
- But there’s a simpler method- @property
- In this, me make the function our attribute.

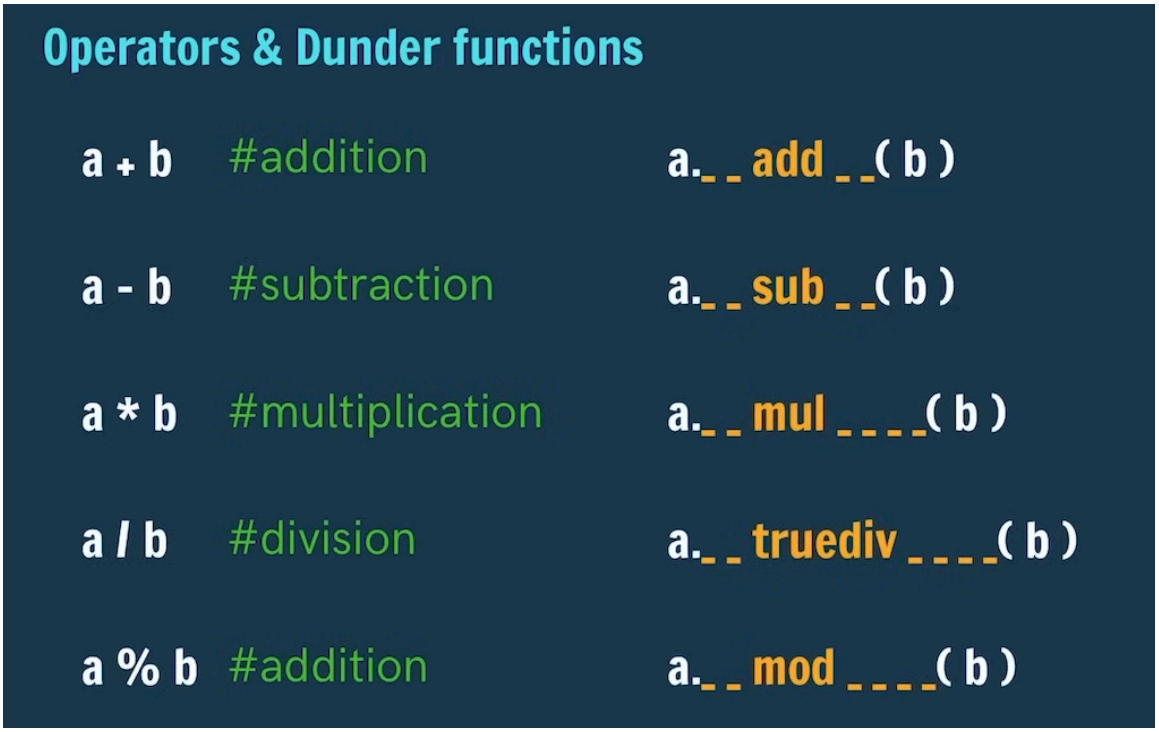
```
class Student:
    def __init__(self, phy, chem, math):
        self.phy = phy
        self.chem=chem
        self.math=math

    @property
    def calper (self):
        return str ((self.phy+self.chem+self.math)/3) +"%"

stu1 = Student(80,68,98)
print(stu1.calper)

stu1.phy=88
```

```
print(stu1.calper)

Output:
82.0%
84.66666666666667%
```

- When we’ll access percentage, we will get latest results.

Polymorphism/: Operator Overloading

- When same operator is allowed to have different meaning according ro the context.



Operators & Dunder functions

a + b	#addition	a.__add__(b)
a - b	#subtraction	a.__sub__(b)
a * b	#multiplication	a.__mul____(b)
a / b	#division	a.__truediv____(b)
a % b	#addition	a.__mod____(b)

```
print (1+2)
print ("slim"+ " shady")
print ([1,2,3]+ [4,5,6])

Output:
3
```

```
slim shady
[1, 2, 3, 3, 4, 5, 6]
```

- The `+` function behaves differently in different context.

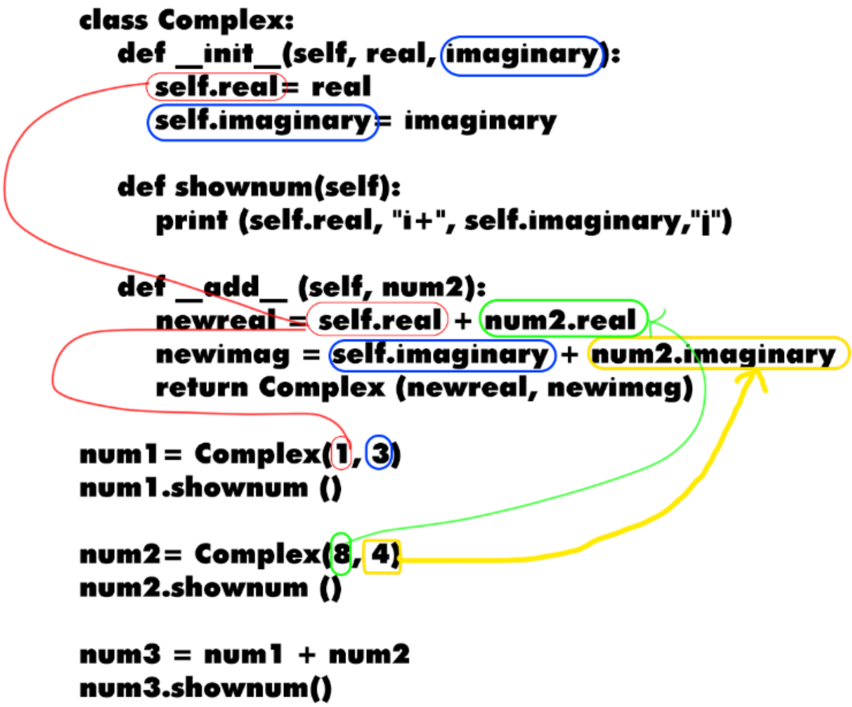
LET’S CREATE COMPLEX NUMBER:

```
class Complex:
    def __init__(self, real, imaginary):
        self.real= real
        self.imaginary= imaginary

    def shownum(self):
        print (self.real, "i+", self.imaginary,"j")

num1= Complex(1, 3)
num1.shownum ()

Output: 1 i+ 3 j
```

- Now we will create 2 complex numbers and add them.
- We do it with the help of **Dunder Function**
    - Dunder = Double Underscore

```
class Complex:
    def __init__(self, real, imaginary):
        self.real= real
        self.imaginary= imaginary

    def shownum(self):
        print (self.real, "i+", self.imaginary,"j")

    def add (self, num2):
        newreal = self.real + num2.real
        newimag = self.imaginary + num2.imaginary
        return Complex (newreal, newimag)

num1= Complex(1, 3)
num1.shownum () # Output: 1 i + 3j

num2= Complex(8, 4)
num2.shownum () # Output: 8 i + 4j

# Use the addcomplex method to add set1 and set2
num3 = num1.add(num2) # This calls the addcomplex method
num3.shownum()  # This should output the result of the addition

Output:
1 i+ 3 j
8 i+ 4 j
9 i+ 7 j
```

- `newreal = self.real + num2.real` → This takes the real part of current number i.e. 1 to real part of num2 i.e. 8.
    - It stores it into `newreal`
    - So `newreal` will contain → 1 +8 =9
- `self.real` refers to 1st number in the object being called
    - `num2.real` refers to 1st number in the object num2
- `self.imaginary` refers to 2nd number in the object being called
    - `num2.imaginary` refers to 2nd number in the object num2



```
class Complex:
    def __init__(self, real, imaginary):
        self.real= real
        self.imaginary= imaginary

    def shownum(self):
        print (self.real, "i+", self.imaginary,"j")

    def __add__ (self, num2):
        newreal = self.real + num2.real
        newimag = self.imaginary + num2.imaginary
        return Complex (newreal, newimag)

num1= Complex(1, 3)
num1.shownum ()

num2= Complex(8, 4)
num2.shownum ()

num3 = num1 + num2
num3.shownum()
```

- But instead of writing this, we want just a simple function `num1 + num2` that will do the addition of complex numbers.
- We do it with the help of Dunder function.
    - we just convert the `add` → `__add__`

```
class Complex:
    def __init__(self, real, imaginary):
        self.real= real
        self.imaginary= imaginary

    def shownum(self):
        print (self.real, "i+", self.imaginary,"j")

    def __add__ (self, num2):
        newreal = self.real + num2.real
        newimag = self.imaginary + num2.imaginary
        return Complex (newreal, newimag)

num1= Complex(1, 3)
num1.shownum ()

num2= Complex(8, 4)
num2.shownum ()

num3 = num1 + num2
num3.shownum()

Output:
1 i+ 3 j
8 i+ 4 j
9 i+ 7 j
```

- Output will be the same.

```
a=6
print (a.__add__(4))

print (6+4)
```

- In this example, when we write `6+4`, it calls `a.__add__(4)`

Best video for this: https://www.youtube.com/watch?v=QttxcLq-Pcs

For subtraction:

```
class Complex:
    def __init__(self, real, imaginary):
        self.real= real
        self.imaginary= imaginary

    def shownum(self):
        print (self.real, "i+", self.imaginary,"j")

    def __sub__ (self, num2):
        newreal = self.real - num2.real
        newimag = self.imaginary - num2.imaginary
        return Complex (newreal, newimag)

num1= Complex(1, 3)
num1.shownum ()

num2= Complex(8, 4)
num2.shownum ()

num3 = num1 - num2
num3.shownum()

Output:
1 i+ 3 j
8 i+ 4 j
-7 i+ -1 j
```

Dunder (Double Underscore) Methods

- Starts and end with `__`
- Enable **operator overloading**, **object initialization**, and other core behaviors.

Why Use Dunder Methods?

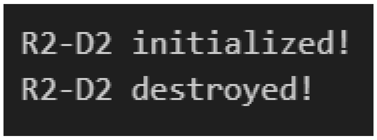
- Make classes behave like built-in types (`int`, `str`, `list`).
- Define how objects respond to operators (`+`, , `==`).
- Control object lifecycle (`__init__`, `__del__`).
- Enable Pythonic features (e.g., iteration, context managers).

`__init__` – The Constructor

```
class Soldier:
    def __init__(self, name):  # called when object is created
        self.name = name
```

✅ Creates the object and sets up its initial data.

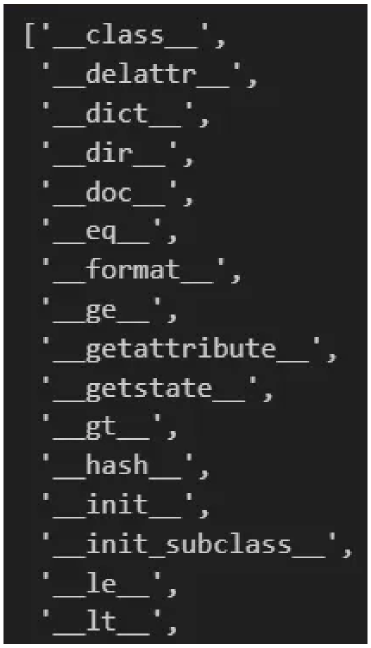
```
class Robot:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} initialized!")

    def __del__(self):
        print(f"{self.name} destroyed!")

r = Robot("R2-D2")  # Output: "R2-D2 initialized!"
del r               # Output: "R2-D2 destroyed!"
```



```
R2-D2 initialized!
R2-D2 destroyed!
```

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"{self.title} ({self.pages} pages)"

    def __len__(self):
        return self.pages

    def __eq__(self, other):
        return self.pages == other.pages
```

```
class Person():
    pass

p= Person()

dir(p)
```



```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
```

💡 **You can change the default behaviour of these by writing your own function.**

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1,2)
v2 = Vector(3,4)
print(v1+v2)

Output:
Vector(4, 6)
```

QUIZ

- Define a Circle class to create a circle with radius r using the constructor.
    - Define an Area() method of the class which calculates the area of the circle.
    - Define a Perimeter() method of the class which allows you to calculate the perimeter of the circle.

    ▼

    ```
    class Circle ():
        def __init__ (self, r):
            self.radius = r

        def area(self):
            self.area = (22/7) * self.radius**2
            print (self.area)
    ```

```
    def parameter (self):
        self.parameter= 2*(22/7)*self.radius
        print (self.parameter)

circle1 = Circle(5)
circle1.area ()
circle1.parameter ()

Output:
78.57142857142857
31.428571428571427
```

- Define an Employee class with attributes role, department & salary.

  This class also has a ***showDetails()*** method.

  Create an Engineer class that inherits properties from Employee & F attributes: name & age.

  ▼

```
class Employee ():
    def __init__ (self,role, department, salary):
        self.role = role
        self.department = department
        self.salary = salary

    def showDetails(self):
        print (f"Role: {self.role}, Department: {self.department}, salary: {self.

employee1 = Employee("Specialist", "Pharmacy", 20000)
employee1.showDetails()

Output:
Role: Specialist, Department: Pharmacy, salary: 20000
```

- Create an Engineer class that inherits properties from Employee & has additional attributes: name & age.

    ▼

    ```
    class Employee ():
        def __init__ (self,role, department, salary):
            self.role = role
            self.department = department
            self.salary = salary

        def showDetails(self):
            print (f"Role: {self.role}\nDepartment: {self.department}\nsalary: {se

    class Engineer (Employee):
        def __init__ (self, name, age):
            self.name=name
            self.age= age
            super().__init__ ("Engineer", "IT", "75,000" )

    eng1 = Engineer ("Sapeksha", 25)
    eng1.showDetails()
    print (eng1.age)
    print (eng1.name)

    Output:
    Role: Engineer
    Department: IT
    salary: 75,000
    25
    Sapeksha
    ```

    - use of `super().__init__` :
        - When you create an `Engineer`, you also want to set up those same properties (like role and department) from `Employee`.

    - **Using `super()`**: By writing `super().__init__("Engineer", "IT", "75,000")`, you're telling Python to run the `Employee` setup for the `Engineer` class. This way, `Engineer` gets its role, department, and salary without having to write that code again.

- Create a class called Order which stores item & its price.
    - Use Dunder function__gt__() to convey that:

  order1 > order2 if price of order1 > price of order2

    ▼

    - `__gt__` is greater than function
    - **When it’s Used**: Whenever you use `object1 > object2`, Python calls `object1.__gt__(object2)` behind the scenes.

```
class Order ():
    def __init__ (self, item, price):
        self.item= item
        self.price= price

    def __gt__ (self, order2):
        return self.price > order2.price

order1= Order("Tea", 10)
order2= Order("Samosa", 20)

print(order1 > order2) #Is my order1 greater than order2?

Output: False
```

    - `return self.price > order2.price`: This line does the actual comparison.
        - `self.price`: This accesses the `price` attribute of the current instance (`self`).

    - `order2.price`: This accesses the `price` attribute of the `order2` instance.
    - `self.price > order2.price`: This checks if the `price` of the current instance is greater than the `price` of the `order2` instance.
    - If `self.price` is greater than `order2.price`, it returns `True`. Otherwise, it returns `False`.

ABC (Abstract Base Class)

> **Abstract Base Class** = A class that **can't be used on its own.** It's a **blueprint** for other classes.

⚙️ Why Use It?

- Forces developers to follow rules:

  > "Every animal must have a sound() method."

- Prevents creating incomplete classes

📦 Code Example

```
from abc import ABC, abstractmethod

# Abstract base class
class Animal(ABC):

    @abstractmethod
    def sound(self):
        pass  # must be implemented in child

# Child classes must implement 'sound'
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"
```

> 💡 “sound” needs to be defined for every inherited class.

⚠ What Happens If You Don’t Implement?

```
class Cow(Animal):
    pass

cow = Cow()  # ❌ ERROR: Can't create Cow because it didn’t define sound()
```

✅ Python prevents creation unless **all abstract methods are defined**.

💡 Trivia

- `abc` module is built-in
- You can define multiple abstract methods
- ABC is a common tool in **Object-Oriented Programming (OOP)**