OOP ppclass

① Encapsulation ② Abstraction ③ Inheritance
④ Polymorphism

ENCAPSULATION:

Wrapping data and methods into single unit.

_variable_name ⟶ #protected

__variable_name ⟶ #private

Object: basic real world run-time entity.
Also called as instance variable of a class.

TYPES of constructor: #no return

- copy # passing the reference
- default # with no arguments
- Parameterised # with arguments
- Dynamic constructor # Obj creation run time

copy constructor passes object as an argument.

```
def __init__(self):

class Student:

    def __init__(self):

        Print('constructor envoked,
                Object created")

S1 = Studen()
```

## Variables

*Instance variable :# Unique for each object
*class variable :# common for all object

### Instant Variable

defined using self.

```
class student:
    def __init__(self,name,age):
        self.name=name
        self.range=age

s1 = student("Daisy", 20)
Print(s1.name)
Print(s1.age)
```

### class variable

```
class Student:
    college = "SIET"
    def __init__(self,name):
        self.name=name
s1 = Student("Daisy")
Print(s1.college, s1.name)
```

### Methods or functions

```
class Student:
    def __init:__(self,name,rank):
        self.name=name
        self.rank=rank
    def display(self):
        Print("Name:", self.name)
        print("Rank:", self.rank)
s1=Student()
```

object variables (or) properties (or) attributes (or) fields (or) columns.

These are controlled with the help of methods and functions.

## constructors

### default constructor:

```
def __init__(self):
    self.name = "Daisy"
```

### Paramterised constructor:

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

### constructor Overloading:

# not Allowed in python

# Using default arguments method instead

```
class Student:
    def __init__(self, amount=0,
                 discount=0):
        self.price = amount
        self.discount = discount

P1 = Student()                  # P1.price,
P2 = Student(500, 50)           P1.discount
P3 = Student(500)

# OP
P1 => 0, 0
P2 => 500 50
P3 => 500 0
```

## *args in python

```
# passes the value as tuples
def Student:
    def __init__(self, *data)
        Print ("student data :", data)
        # Printing what passed through
data
S1 = Student ("Daisy")
S8 = Student ("Daisy", "EEE", "8.8", "20")

# OP
Student data : ('Daisy')
Student data : ('Daisy', 'EEE', 8.8, 20)
```

## **kwargs in python

```
#dictionary in simple words.
def Student:
    def __init__(self, **info):
        Print(info) # printing the data
                    Passed through
                    info.
S1 = Student() # not necessary
student ("name": "Daisy", age: 20, gender:
                                "Female")

# OP
{name: "Daisy", age: 20, gender: "Female"}
```

__new__ (real constructor)

# allocates memory for the object.

Syntax:

```
def __new__(cls):
    return sup().__new__(cls)
```

or

| __new__ | __init__ |
| --- | --- |
| (*static method<br>* creates object<br>* returns new instance or object<br>* called first | * instance method<br>* initializes already created object<br>* returns None<br>* called second |

- cls keyword => similar to "self"
- super().__new__(cls) => actually allocates new objects
- Must return the created objects
- we can modify only in __new__

## Encapsulation

Encapsulation: data + method bundled & controlled access.

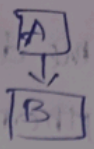
=> Everything is public until we protect it.

## Encapsulation

=> Getters and Setters
=> Private
=> Public
=> Protected
=> @ property
- @ detter
- @ setter
- @ docstring
- @ dynamic property

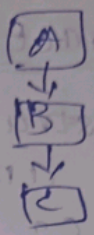
# Inheritance

=> Single Inheritance




```
class A:
    #
class B(A):
```

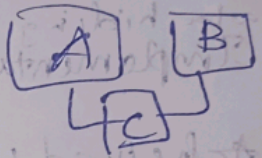
=> Multilevel Inheritance




```
class A:
    #
class B(A):
    #
class C(B):
    #
```

=> Multiple.

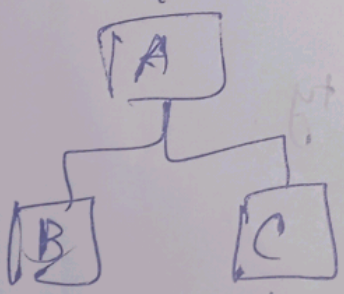
=> ~~Hierarchical~~ Inheritance.




```
class A:
    #
class B:
    #
class C(A, B):
    #
```

=> Multiple Inher~~ar~~

=> Hierarchical Inheritace




```
class A:
    #
class B(A):
    #
class C(A):
    #
```

## Polymorphism:

Same name different behaviours
refer python / oops / sofdev.

## Abstraction:

Abstraction hides implementation details and shows only essential features.

Expose: What it does
Hide: How it does

| Abstraction | Encapsulation |
|---|---|
| Data hiding Implementation<br>• Acheived using abstract classes<br>Focus on methods<br>Force subclass to Implement | ~~Implementing~~ Data hiding<br>Acheived using<br>• Private<br>• Protected<br>Focus on Variables<br>Prevent direct access |

Encapsulation: protect data
Abstraction: hide complexity

## Syntax

```
from abc import ABC
```

abc => Abstract Base classes

ABC => A special base class used to create abstract classes

Abstract Method :

A decorator used to declare abstract methods.

Abstract class cannot create objects