* Python OOPs Concepts *

↳ Python is an OO Lang, so can design prog using classes & objects.

↳ OOPs concept focuses on writing the reusable code.

↳ Technique to solve problem by creating objects.

◉ Major Principles of OOP language :-

① **Class**: Defined as coll^n of objects.

↳ Serves as blue print for creating objects.

↳ Logical entity has some specific attr & methods

↳ **Ex**: Student class contains - id, name, class, mark attr & give_exam(), pay_fees(), attend_classes() as the methods of the class.

- **Syntax** -

```
class Name:
    # stmt1
    :
    # stmtn
```

② **Object**: Entity having state & behavior.

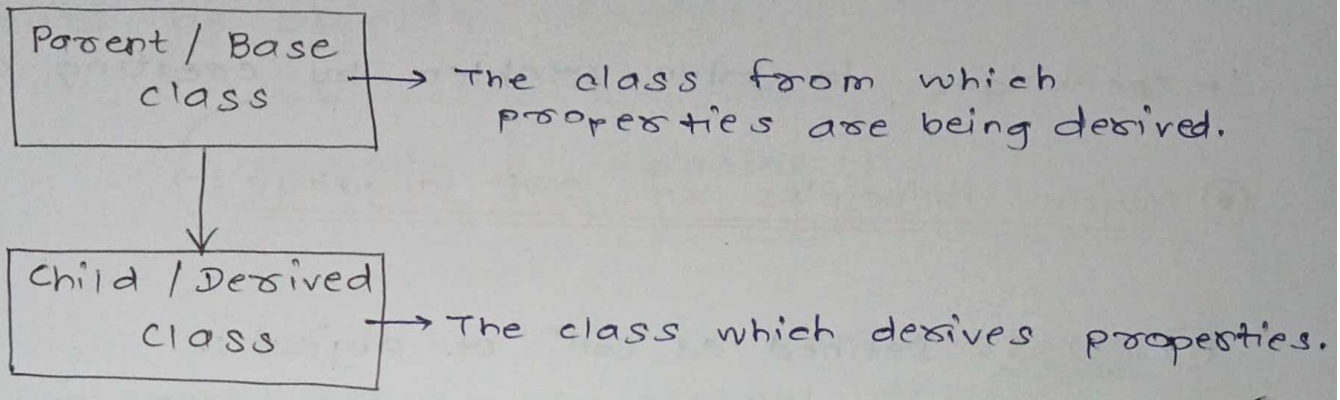
↳ May be real world object like - pen, pencil etc.

- **Syntax** -

```
obj = class_name(arg_list)
obj. variables  # state of obj.
obj.methods()   # Behavior of obj.
```

③ Method - Fun^n associated with an object.
↳ Not unique to class instances.

④ Inheritance: capability of one class to derive or inherit properties from another class.



↳ Provide code re-usability.

⑤ Polymorphism: poly + morphs
Many shapes.

↳ So, polymorphism simply means having many forms.

↳ Here, we can create methods with same name, but they perform different actions.

↳ Polymorphism is possible in 2 ways:

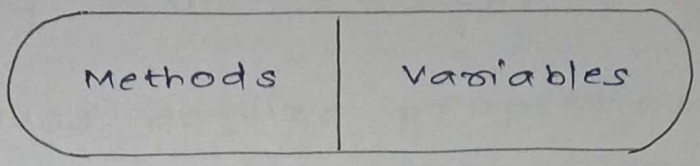
① Runtime Polymorphism - Ex. method overriding.
Meth with same name & param list, present in both parent & child classes.
↳ Python support method overridding.

② Compile-time polymorphism - Ex. meth overloading.
↳ Meths in sam class, but have different param list.
↳ Python does not support meth overloading.

⑥ Encapsulation: Restrict access to methods & variables from the outside world.

↳ Code & data are wrapped together within a single unit, so can't modified.

↳ To prevent accidental changes, var can only be changed by object's methods.



class

⑦ Data Abstraction: Hide internal details & show only the functionality.

↳ Give names to the things, so that name captures what a fun^n / the whole program does.

↳ Abstraction can be achieved by creating abstract classes.

1) Inheritance :-

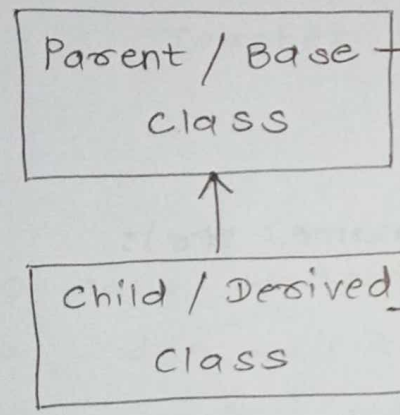
↳ In Python, every class inherits from built-in basic class called 'object'.

↳ Capability of one class to derive / inherit properties of another class.

↳ Provides code re-usability to prog, bcz can use existing class to create new class, instead of creating from scratch.

↳ child class acquire properties & can access all var & methods of parent class.

↳ child can also add its own var & meth.



→ Class from which properties are being derived.

→ Class which derives properties.

- Benefits of Inheritance :-

↳ Allow inherit base class props into derived.

↳ Represent real-world relationship.

↳ Provide code re-usability.

↳ Less development & expenses result.

↳ Offers simple structure.

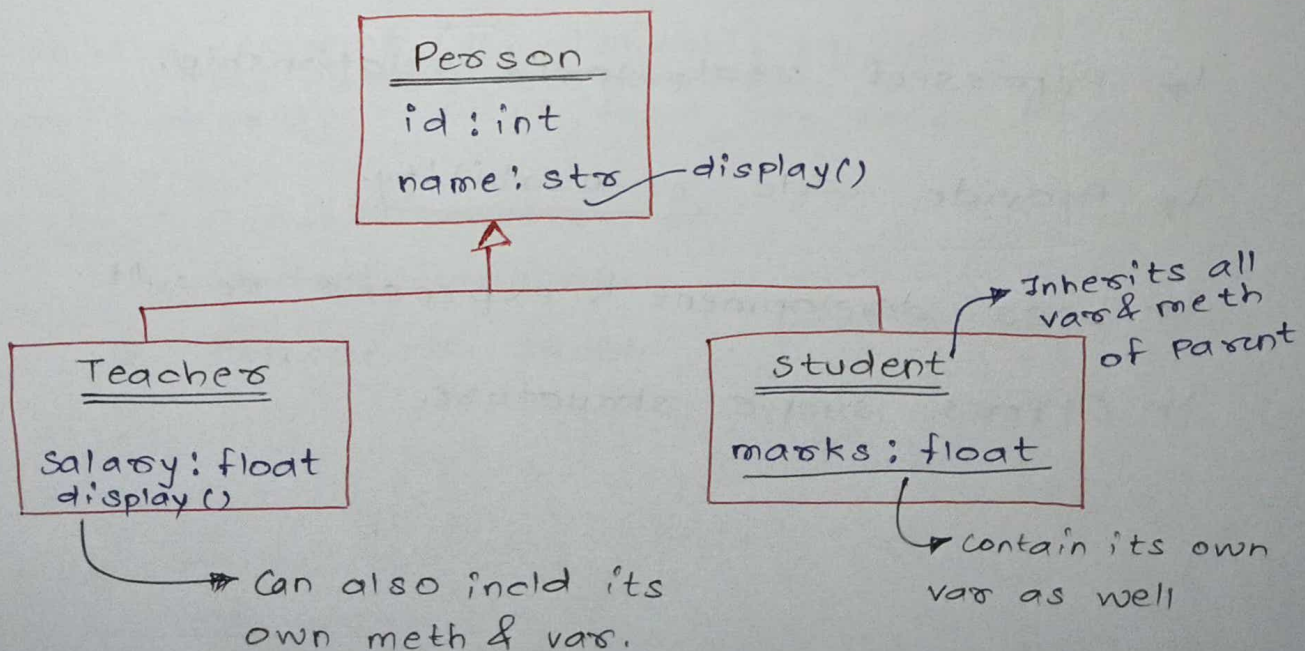
- **Syntax:**

```
class BaseClass:
    def __init__(self):
        .....
    {Other Body}

class Inherited class (BaseClass):
    def __init__(self):
        super().__init__(self).
```

- **Simple Inheritance Example:**

```
>>> class Person:
>>>     def __init__(self, id: int, name: str):
>>>         self.id = id
>>>         self.name = name
>>>         print('Person constructor...')
>>>     def display(self) -> None:
>>>         print('Person', self.id, self.name)
```



- The super() Keyword :-

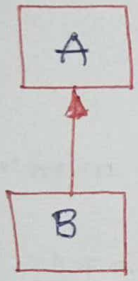
↳ When used in child class, returns objects that represent parent class.

↳ It allows to access parent class's meth & attr in the child class.

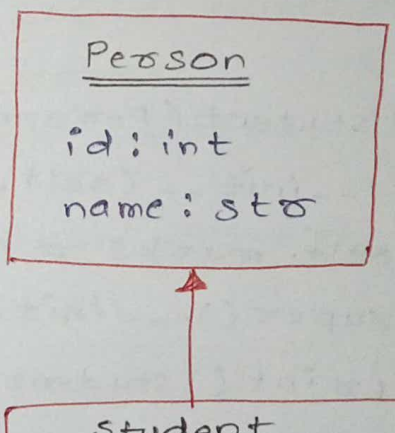
⊙ Different types of Python Inheritance :-

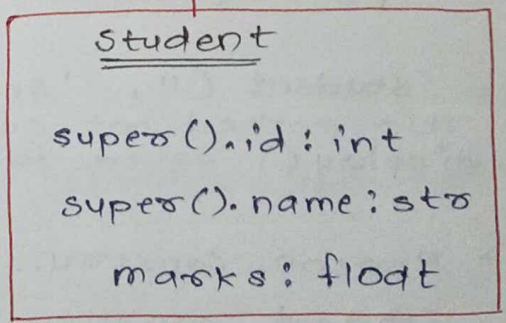
① Single Inheritance :-

↳ child inherit from only one parent class.



Single Inheritance





- Example

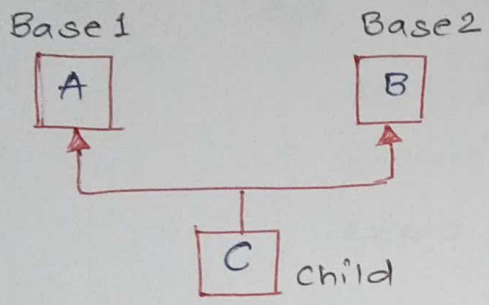
- Syntax -

```
class A:
    def __init__(self):
        # stmts
    def meth(self):
        :
class B(A):
    def __init__(self):
        super().__init__()
```

② Multiple Inheritance –

↳ Derive a class from more than one base classes.

↳ All features of all the base classes are inherited into the derived class.

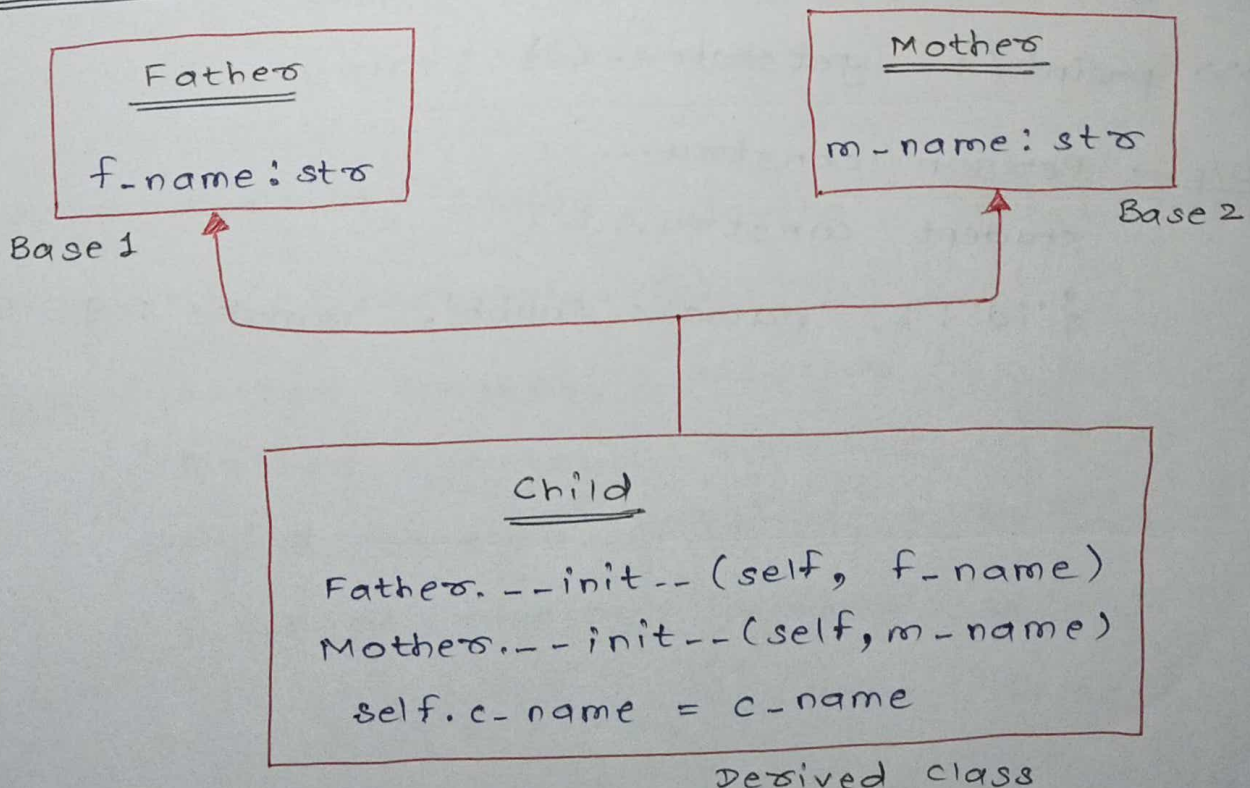


Multiple Inheritance

- Syntax –

```
class A:
    def __init__(self):
        ....
class B:
    def __init__(self):
        ....
class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
```

- Example –

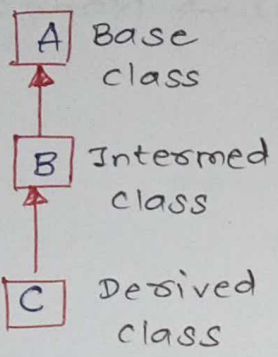


③ Multilevel Inheritance -

↳ Derived class inherits from another derived.

↳ No limit on no. of levels up to which, the multi-level inheritance is achieved.

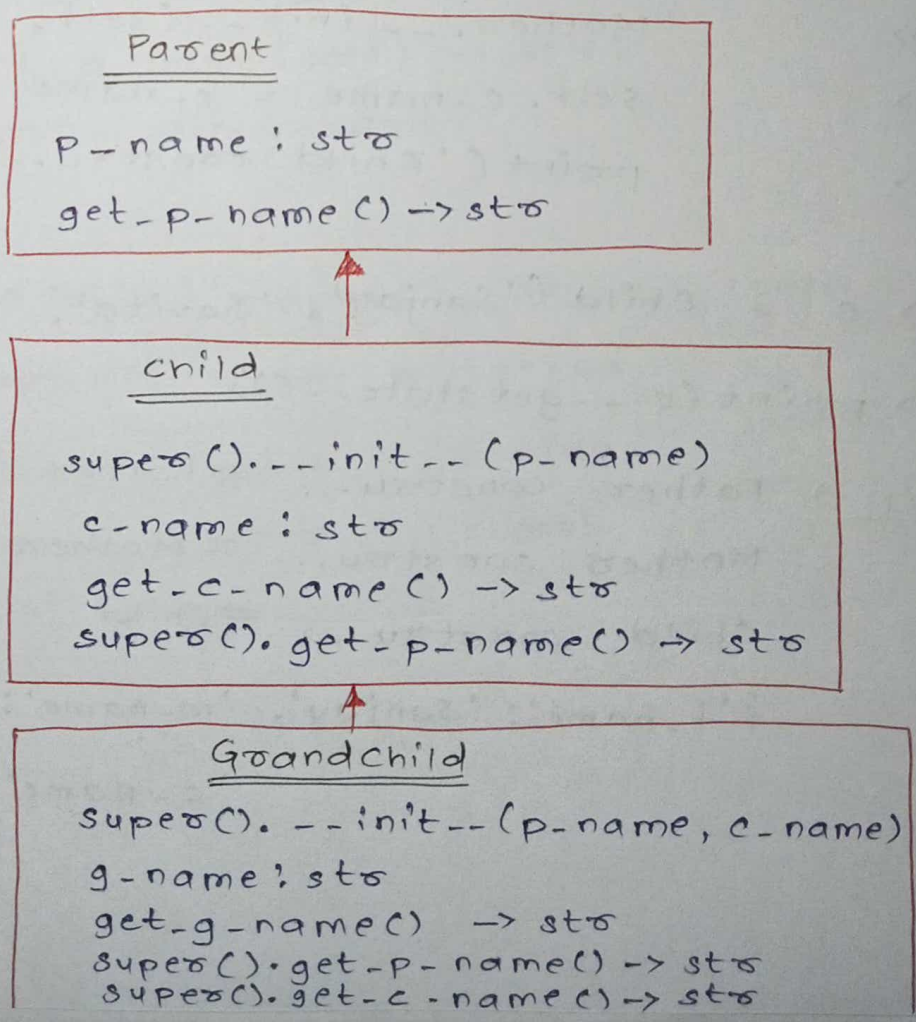
↳ we have, relationship. `Parent ← Child ← Grandchild`



- Syntax -

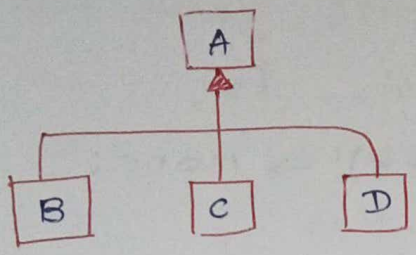
```
class A:
    def __init(self):
        .....
class B(A):
    def __init__(self):
        super().__init__(self)
class C(B):
    def __init__(self):
        super().__init__(self)
```

- Example -



④ Hierarchical Inheritance –

↳ More than 1 derived class created from single base.

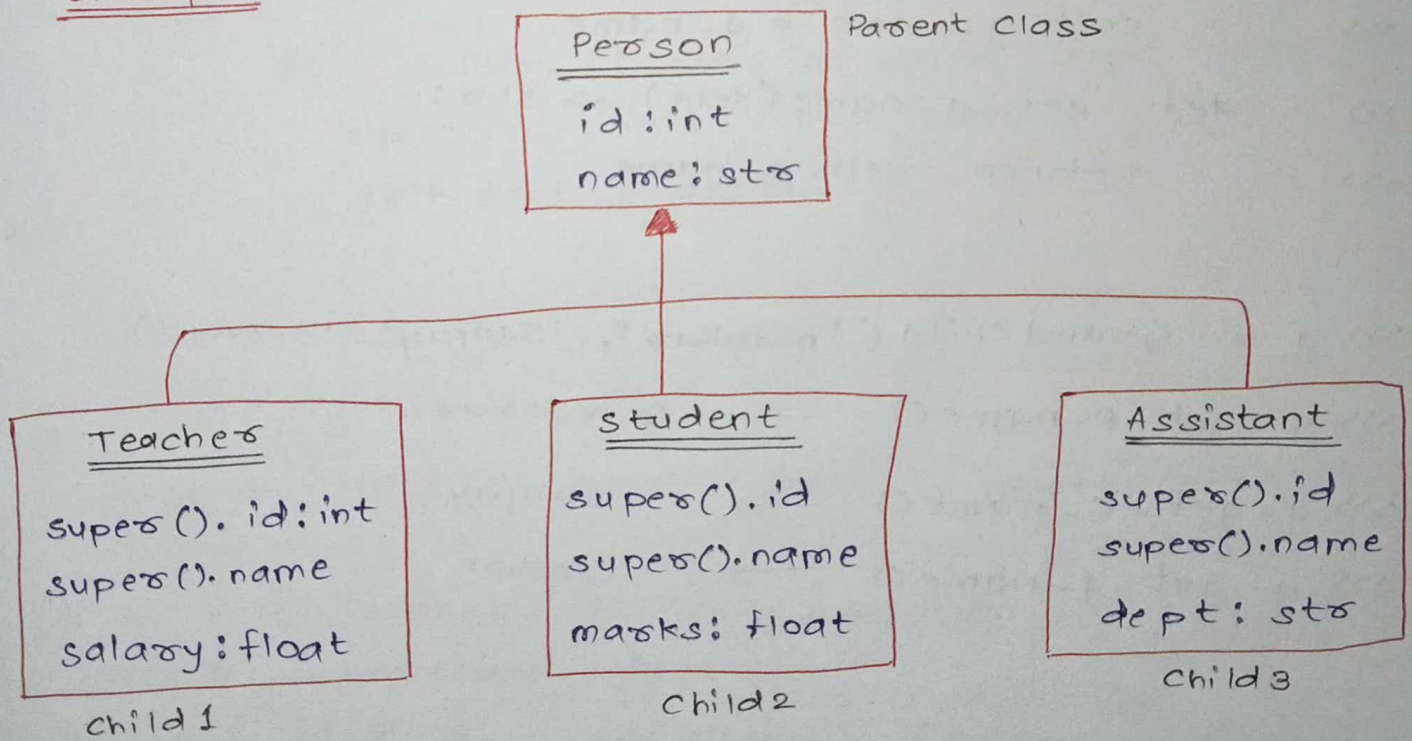


Hierarchical Inheritance

- Syntax –

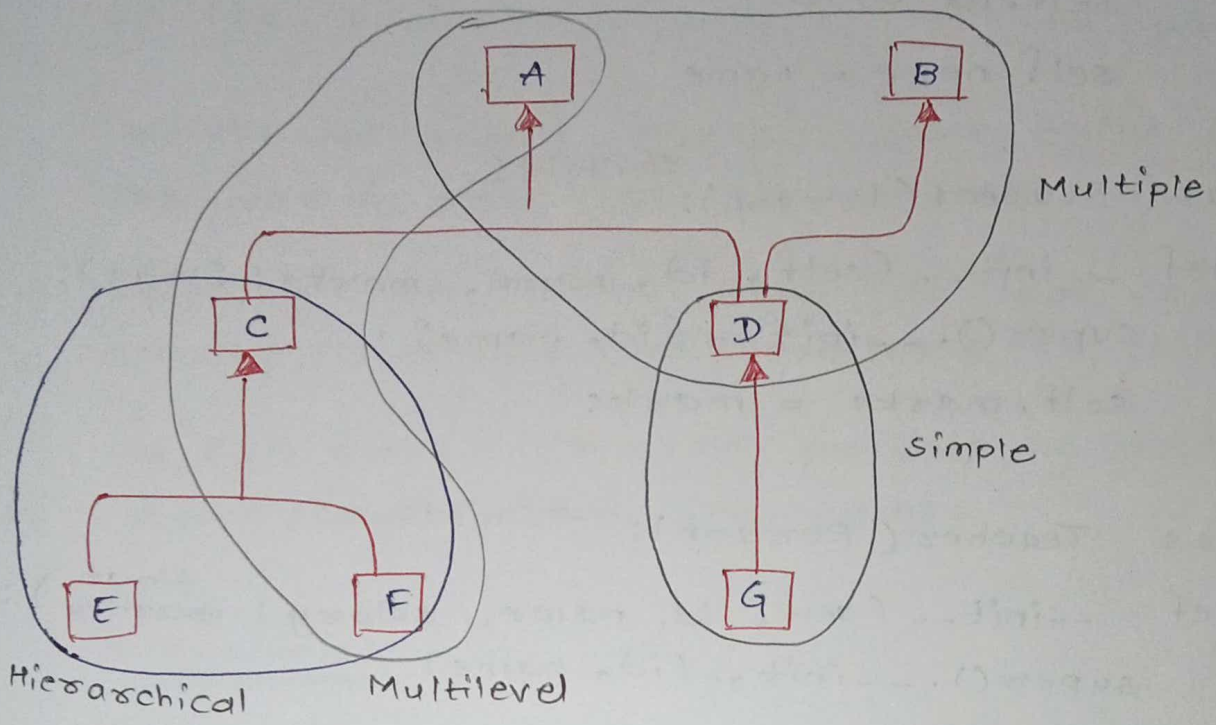
```
class A:
    def __init__(self):
        ...
class B(A):
    def __init__(self):
        super().__init__()
class C(A):
    def __init__(self):
        super().__init__()
class D(A):
    def __init__(self):
        super().__init__()
```

- Example –

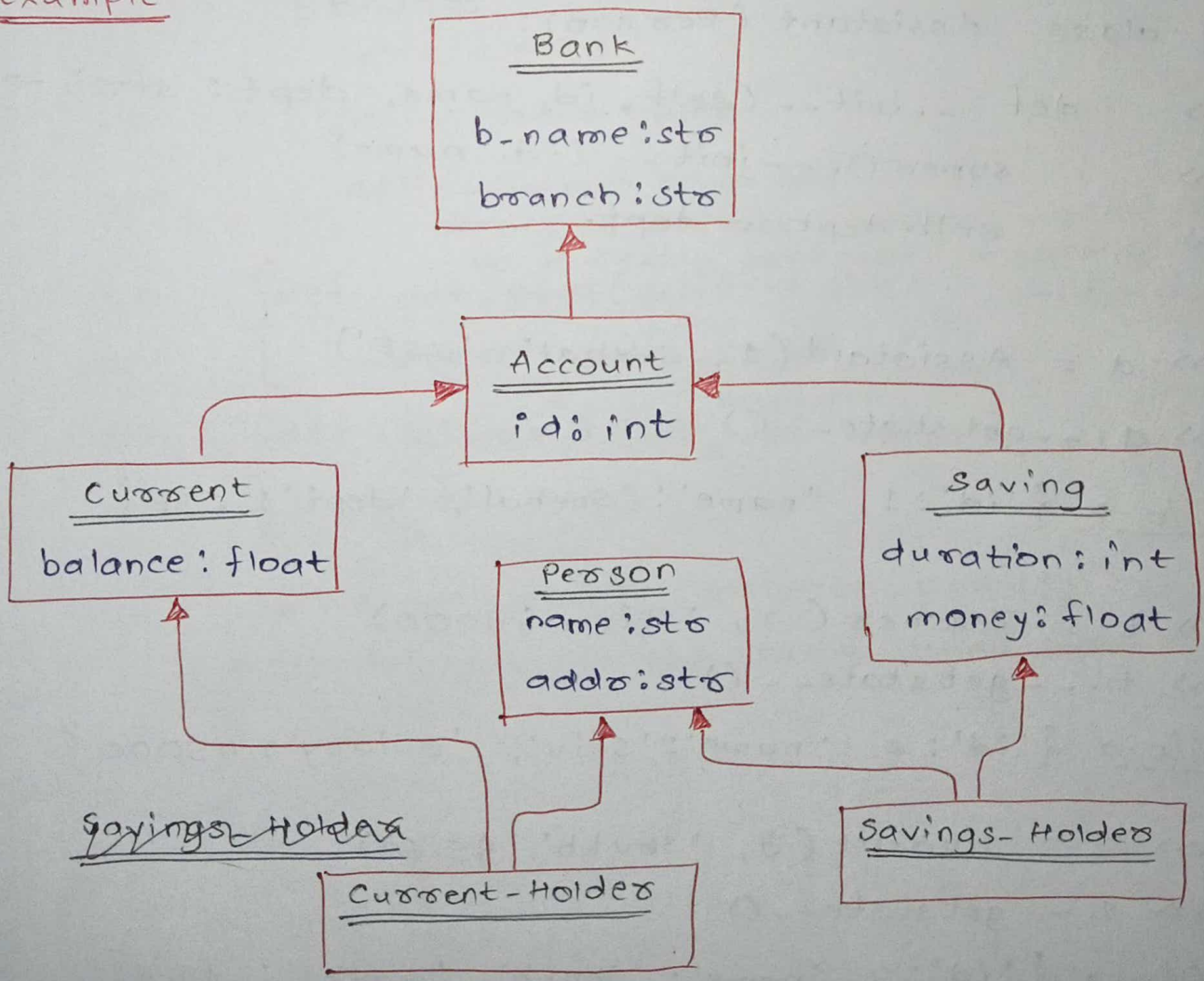


⑤ Hybrid Inheritance :-

↳ Combines more than 1 form of inheritance.



- Example -



⊙ Private members of the class:-

↳ Inheritance allows the child class to access all the variables & methods of the parent class.

↳ But, we won't always want child to inherit the var & meth of the parent class.

↳ so, we can name some of inst var of parent class private, won't available to child class.

↳ Can make inst var private by adding double underscore before its name.

- Example:

```
>>> class Account:
>>>     def __init__(self, id:int, name:str, pwd:str):
>>>         self.id = id
>>>         self.name = name
>>>         self.__pwd = pwd
                #This is private variable, which can't
                 be directly accessed / modified by
>>>     def get_pwd(self) -> str:      child class.
>>>         return self.__pwd
>>>     def set_pwd(self, pwd:str) -> None:
>>>         self.__pwd = pwd.
```

↳ we can modify private var by public methods only, in the same class only.

⊙ issubclass (sub_class, super_class) method :-

↳ Used to check relationship bet^n specified class.

↳ Return True if first subclass of second class, and False otherwise.

```
>>> class A:
>>>     pass
>>> class B(A):
>>>     pass
>>> print(issubclass(A, B))  ⟶ False
>>> print(issubclass(B, A))  ⟶ True
```

sub-class, super-class

⊙ isinstance (object, class) method :-

↳ Used to check relations between objects & classes.

↳ Return true if object is an instance of class.

```
>>> print(isinstance(a, A))  ⟶ True
>>> print(isinstance(a, B))  ⟶ False
```

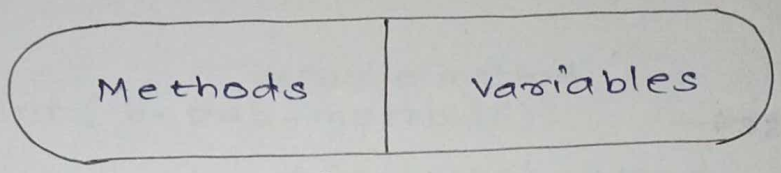
Object, Class

2) Encapsulation :-

↳ Wrapping data (var) & methods that work on data within one unit.

↳ Put restriction on var accessing directly & can prevent accidental modificat° of data.

↳ To prevent accidental change, object's variable can only changed by obj methods, those cannot be accessed / modified directly. These cla private var.



Class

↳ Encapsula° also hides data.

① Public members :

↳ All the variables & methods are default 'public'.

↳ Public attributes are can be accessed in the same class, in its sub-class & anywhere outside the class as well.

② Protected members :

↳ Members that can't be accessed outside class, but accessed from within the class & its sub-class.

↳ Protected members can be declared, by using one underscore before the name of variable. (single)

↳ As protected attr can accessed outside class as well in derived class (can also modified), it is just a convention not to access outside class body.

③ Private members:

↳ Similar to protected members, but these can't be accessed outside the class / in the base class.

↳ There is no existence of Private inst variables that can't be accessed except inside a class.

↳ Private members can be declared using the double underscore (__) before the attr name.

- **Example –**

```
>>> class Base:
>>>     pub = 0          # public variable
>>>     _prt = 0         # Protected variable
>>>     __pvt = 0        # Private variable
>>>
>>>     def __init__(self, pub:int, prt:int, pvt:int):
>>>         self.pub = pub
>>>         self._prt = prt
>>>         self.__pvt = pvt
```

} can initialize pub, prot, pvt var inside constr.

```
>>>     def pub_method(self) -> int:
>>>         return self.pub
```

} Public method

```
>>>     def _prot_method(self) -> int:
>>>         return self._prt
```

} Protected method

```
>>>     def __pvt_method(self) -> int:
>>>         return self.__pvt
```

} Private method, not accessible outside class.

```
>>>     def method(self):
>>>         print(self.__pvt)
>>>         print(self.__pvt_method())
```

} can access pvt var & meth using public method.

```
>>> b = Base(10, 20, 30)
>>> print(b.--getstate--())
```

O/p → {'pub': 10, '_prot': 20, '_Base--pvt': 30}

```
                    #Public var
>>> print(b.pub)          → 10
                    #Protected var
>>> print(b._prot)        → 20
```

} can be accessible outside class.

```
>>> print(b.--pvt)  # Private var not accessible.
```

O/p → AttributeError: 'Base' object has no attribute '--pvt'

```
                              #Public method
>>> print(b.pub_method())          → 10
                              #Protected method
>>> print(b._prot_method())        → 20
```

} can be access outside class.

```
>>> print(b.--pvt_method())  # Private method not accessible outside class.
```

O/p → AttributeError: 'Base' object has no attribute '--pvt_method'

```
>>> b.method()   # can access private methods & variables outside the class using the public methods.
```

O/p → 30
30

⊙ Access Modifiers: Public, Protected & Private :-

↳ OOP languages control access modifications which are used to restrict access to var & methods of the class.

↳ Python use '_' symbol to determine access control for specific data member / members fun^ of a class.

↳ Access specifiers are imp to securing data from unauthorized access & prevent from data exposing.

① Public Access Modifier:

- Easily accessible from any part of the prog.
- All var & meth of class public by default.

② Protected Access Modifier:

- Only accessible to a class derived from it.
- Declare protected by adding single underscore (_) before name of the data member.

③ Private Access Modifier:

- Accessible within the class only, are most secure.
- Declare private by adding double underscore '__' before name of the data member.

3) Data Abstraction :-

↳ Hide internal functionality of fun^n from the users.

↳ Users only interact with basic impl^n of fun^n, but inner working is hidden.

↳ User is familier with that "what fun^n does", but they don't know "how it does".

↳ Example - we use smartphone & familier with camera, voice-recorder, call-dialing, etc, but we don't know these operations happening in the background.

- Why Abstraction Important ?

↳ Hide irrelevant data/class in order to reduce complexity.

↳ Also enhance appl^n efficiency.

- ⊙ Abstraction classes in Python :-

↳ In Python, abstr^n achieved by using abstract classes & interfaces.

↳ Abstract method - Not contain their impl^n.

↳ Abstract classes - consist of one/more abstr meth.

↳ Abstract classes serves as blueprint for other classes. They can be inherited by subclass & abstr method gets defini^n in subclass.

↳ Python provides abc module to use the abstraction in the Python program.

- Syntax:

```
from abc import ABC
class Base(ABC):
    def method1(self):
        pass
    .
    .
    .
```

Abstract Base Classes (ABC):-

↳ ABC is common appln program of interface for set of subclasses.

↳ It is beneficial, when we work with large code-base hard to remember all the classes.

Working of Abstract Classes:-

↳ Python doesn't provide abstract class itself, have to import abc module, which provides base for defining Abstract Base classes (ABC).

↳ It registers concrete classes as impln of the abstract base.

↳ we use @abstractmethod decorator to define abstract method / if we don't provide definin to the method, it automatically becomes abstract method.

- Example –

```
>>> from abc import ABC, abstractmethod

>>> class Shape(ABC):
>>>     def __init__(self) -> None:
>>>         pass
>>>     @abstractmethod
>>>     def sides(self) -> None:
>>>         pass

>>> shape = Shape()
```

This is abstr meth with on impl? & contains the @ abstractmethod decorator.

O/p → TypeError: Can't instantiate abstract class Shape with abstract method sides

```
>>> class Square(Shape):
>>>     def __init__(self, s: float) -> None:
>>>         super().__init__()
>>>         self.s = s

>>>     def sides(self) -> None:
>>>         super().sides()
>>>         print('side:', self.s)

>>> sq = Square(20)
>>> sq.sides()
```

This is impl^n of the abstract method from the Parent class.

O/p → side : 20

4) Polymorphism :-

Poly (Many) + Morphs (Forms)

⊙ What is Polymorphism: Polymorphism simply means one name, but many forms, being used for different types.

↳ The difference is only data types & no. of args used in fun^n.

↳ Here, we can create methods with same name, but they perform different actions.

↳ Polymorphism is possible in 2 ways:

① Runtime polymorphism – Ex. method overriding.
- Method with same name & param list, present in both Parent & child classes.
- Python support method overriding.

② Compile-time Polymorphism – Ex. Meth overloading
- Methods in same class, but having diff params list & data types.
- Python does not support meth overloading.

• Note: Python does not support method overloading.

- Example –

```
#Method overridding:
>>> class Bird:
>>>     def __init__(self, name: str) -> None:
>>>         self.name = name
>>>     def flight(self) -> None:
>>>         print('Some birds cannot fly...')

>>> class Sparrow(Bird):
>>>     def __init__(self, name: str) -> None:
>>>         super().__init__(name)
>>>     def flight(self) -> None:
>>>         super().flight()
>>>         print(self.name, 'can fly...')
```

This is the overridden method from the base class.

```
>>> s = Sparrow('sparrow')
>>> s.flight()
```

O/p → Some birds cannot fly....
Sparrow can fly...

```
#Method Overloading
>>> def Ostrich(Bird):
>>>     def __init__(self, name:str) -> None:
>>>         super().__init__(name)
>>>
>>> ①   def flight(self) -> None:
>>>         super().flight()
>>>         print('No param')
>>>
>>> ②   def flight(self, name):
>>>         super().flight()
>>>         print(name, 'param')
```

These both are overloaded methods.

↳ When we declare overloaded methods, Python just considers the method declared at the very bottom.

```
>>> o = Ostrich('Ostrich')
>>> o.flight()  #cannot call the method declared earlier.
```

O/p → TypeError: Ostrich.flight() missing 1 positional argument: 'name'

```
>>> o.flight('snehal')  #Can call the only method present at the bottom.
```

O/p → snehal param