OOPs Concepts:

1. Data Abstraction:

- Hide internal details and show functionalities only.
- Abstraction can be achieved by using abstract classes and interfaces.

2. Encapsulation:

- Used to restrict access to methods and variables.
- Code and data are wrapped together within a single unit, to prevent from being modified accidently.

3. Inheritance:

- Inheritance is the capability of one class to derive or inherit the properties from another class.
- The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.
- Types of Inheritance:
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Multiple Inheritance
  - Hybrid Inheritance

4. Polymorphism:

- Polymorphism simply means having many forms.
- Means, declaring methods with same name, but performing different actions.

1. Abstraction:

In [1]:

```python
from abc import ABC, abstractmethod

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

class Rectangle(Shape):
    def __init__(self, l: float, b: float) -> None:
        super().__init__()
        self.l = l
        self.b = b
    def sides(self):
        super().sides()
        print('Length:',self.l,'Breadth:',self.b)
    def area(self):
        return self.l*self.b
    def perimeter(self):
        return 2*(self.l+self.b)

class Square(Shape):
    def __init__(self, s: float) -> None:
        super().__init__()
        self.s = s
    def sides(self):
        super().sides()
        print('Side:',self.s)
    def area(self):
        return self.s**2
    def perimeter(self):
        return 4*self.s

r = Rectangle(10, 20)
r.sides()
print('Perimeter', r.perimeter())
print('Area', r.area())

s = Square(20)
s.sides()
print('Perimeter', s.perimeter())
print('Area', s.area())

shape = Shape()
# TypeError: Can't instantiate abstract class Shape with abstract methods d
```

```
Length: 10 Breadth: 20
Perimeter 60
Area 200
Side: 20
Perimeter 80
Area 400
```

```
---------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
Cell In[1], line 51
     48 print('Perimeter', s.perimeter())
     49 print('Area', s.area())
---> 51 shape = Shape()

TypeError: Can't instantiate abstract class Shape with abstract methods a
rea, perimeter, sides
```

abstractproperty

In [2]:
```python
from abc import ABC, abstractmethod, abstractproperty

class Base(ABC):
    def __init__(self) -> None:
        super().__init__()
    @abstractproperty
    def wish(self) -> str:
        return 'Base class'

class Derived(Base):
    def __init__(self) -> None:
        super().__init__()
    @property
    def wish(self) -> str:
        return super().wish

d = Derived()
d.wish

# b = Base()
# b.wish
```

Out[2]:
```
'Base class'
```

2. Encapsulation:

In [ ]:
```python
class Base:
    # public data member
    pub = 0
    # protected data member
    _prot = 0
    # private data member
    __pvt = 0
    def __init__(self, pub: int, prot: int, pvt: int) -> None:
        self.pub = pub
        self._prot = prot
        self.__pvt = pvt
    def pub_method(self) -> int:
        return self.pub
    def _prot_method(self) -> int:
        return self._prot
    def __pvt_method(self) -> int:
        return self.__pvt
    def method(self):
        print(self.__pvt)
        print(self.__pvt_method())

b = Base(10, 20, 30)
print(b.__getstate__())
print(b.pub)
print(b._prot)
# print(b.__pvt)
# AttributeError: 'Base' object has no attribute '__pvt'

print(b.pub_method())
print(b._prot_method())
# print(b.__pvt_method())
# AttributeError: 'Base' object has no attribute '__pvt_method'

# Can access the private methods and attributes from public methods
# inside the class
b.method()
```

3. Inheritance:

Simple Inheritance Example:

In [3]:
```python
# Create the class Person and then inherit to Teacher and Student

# Parent Class
class Person:
    def __init__(self, id:int, name:str) -> None:
        self.id = id
        self.name = name
        print('Person called..')
    def display(self, name: str) -> None:
        print('Person', self.id, self.name)
    def state(self) -> None:
        print(self.__getstate__())

# Child1 Class
class Teacher(Person):
    # Have to take the params for parent class initialization
    # Also take params for child class, if needed
    def __init__(self, id: int, name: str, salary:float) -> None:
        # Call Parent class Constructor
        super().__init__(id, name)
        self.salary = salary
        print('Teacher called..')
    # Method overridden from the parent
    def display(self) -> None:
        print('Teacher', self.id, self.name, self.salary)

# Child2 Class
class Student(Person):
    def __init__(self, id: int, name: str, marks:float) -> None:
        super().__init__(id, name)
        self.marks = marks
        print('Student called..')
    # If overridden method not present, then call the method from parent cl
    # def display(self) -> None:
    #     print('Student', self.id, self.name, self.marks)


p1 = Person(1, 'Snehal')
p1.display('snehal')

t1 = Teacher(21, 'Shiv', 45000.50)
t1.display()
t1.state()

s1 = Student(11, 'Shubh', 89.60)
s1.display('asdf')
s1.state()
```

```
Person called..
Person 1 Snehal
Person called..
Teacher called..
Teacher 21 Shiv 45000.5
{'id': 21, 'name': 'Shiv', 'salary': 45000.5}
Person called..
Student called..
Person 11 Shubh
{'id': 11, 'name': 'Shubh', 'marks': 89.6}
```

Types of Inheritance:

1. Single Inheritance:

In [4]:
```
class Person:
    def __init__(self, id:int, name:str) -> None:
        self.id = id
        self.name = name
        print('Person called..')
    def __str__(self) -> str:
        return f'{self.id}, {self.name}'

class Student(Person):
    def __init__(self, id: int, name: str, marks: float) -> None:
        super().__init__(id, name)
        self.marks = marks
        print('Student called..')
    def __str__(self) -> str:
        return f'{self.id}, {self.name}, {self.marks}'

p = Person(1, 'Snehal')
print(p)
print(p.__getstate__())

s = Student(2, 'Shubh', 95.50)
print(s)
print(s.__getstate__())
```

```
Person called..
1, Snehal
{'id': 1, 'name': 'Snehal'}
Person called..
Student called..
2, Shubh, 95.5
{'id': 2, 'name': 'Shubh', 'marks': 95.5}
```

2. Multiple Inheritance:

In [5]:
```python
# Create 2 base classes and inherit to one child class

class Father:
    def __init__(self, f_name:str) -> None:
        self.f_name = f_name
        print('Father constructor..')

class Mother:
    def __init__(self, m_name:str) -> None:
        self.m_name = m_name
        print('Mother constructor..')

class Child(Father, Mother):
    def __init__(self, f_name: str, m_name: str, c_name: str) -> None:
        Father.__init__(self, f_name)
        Mother.__init__(self, m_name)
        self.c_name = c_name
        print('Child constructor..')

f = Father('Sanjay')
print(f.__getstate__())

m = Mother('Savita')
print(m.__getstate__())

c = Child('Sanjay', 'Savita', 'Snehal')
print(c.__getstate__())
```

```
Father constructor..
{'f_name': 'Sanjay'}
Mother constructor..
{'m_name': 'Savita'}
Father constructor..
Mother constructor..
Child constructor..
{'f_name': 'Sanjay', 'm_name': 'Savita', 'c_name': 'Snehal'}
```

3. Multilevel Inheritance :

In [6]:
```
class Parent:
    def __init__(self, p_name: str) -> None:
        self.p_name = p_name
        print('Parent constructor..')
    def get_p_name(self) -> str:
        print('Parent class name..', self.p_name)
        return self.p_name


class Child(Parent):
    def __init__(self, p_name: str, c_name) -> None:
        super().__init__(p_name)
        self.c_name = c_name
        print('Child constructor..')
    def get_c_name(self) -> str:
        print('Child class name..', self.c_name)
        return self.c_name


class GrandChild(Child):
    def __init__(self, p_name: str, c_name, g_name) -> None:
        super().__init__(p_name, c_name)
        self.g_name = g_name
        print('GrandChild constructor..')
    def get_g_name(self) -> str:
        print('GrandChild class name..', self.g_name)
        return self.g_name
    def get(self) -> str:
        return super().get_p_name()


g = GrandChild('mankar', 'sanjay', 'snehal')
g.get_p_name()
g.get_c_name()
g.get_g_name()
g.get()
```

```
Parent constructor..
Child constructor..
GrandChild constructor..
Parent class name.. mankar
Child class name.. sanjay
GrandChild class name.. snehal
Parent class name.. mankar
```

Out[6]:
```
'mankar'
```

4. Hierarchical Inheritance:

In [7]:
```python
# More than one derived class can be created from a single base.

class Person:
    def __init__(self, id: int, name: str) -> None:
        self.id = id
        self.name = name

class Teacher(Person):
    def __init__(self, id: int, name: str, salary: float) -> None:
        super().__init__(id, name)
        self.salary = salary

class Student(Person):
    def __init__(self, id: int, name: str, marks: float) -> None:
        super().__init__(id, name)
        self.marks = marks

class Assistant(Person):
    def __init__(self, id: int, name: str, dept: str) -> None:
        super().__init__(id, name)
        self.dept = dept

class Principal(Person):
    def __init__(self, id: int, name: str, exper: int) -> None:
        super().__init__(id, name)
        self.experience = exper

p = Principal(1, 'Snehal', 10)
print(p.__getstate__())

a = Assistant(2, 'Shubh', 'CSE')
print(a.__getstate__())

s = Student(3, 'Kal', 89.48)
print(s.__getstate__())

t = Teacher(4, 'Shiv', 67000)
print(t.__getstate__())
```

```
{'id': 1, 'name': 'Snehal', 'experience': 10}
{'id': 2, 'name': 'Shubh', 'dept': 'CSE'}
{'id': 3, 'name': 'Kal', 'marks': 89.48}
{'id': 4, 'name': 'Shiv', 'salary': 67000}
```

5. Hybrid Inheritance:

In [8]:
```
# This form combines more than one form of inheritance.
# Basically, it is a blend of more than one type of inheritance.

class Bank:
    def __init__(self, bank_name: str, branch: str) -> None:
        self.bank_name = bank_name
        self.branch = branch

class Account(Bank):
    def __init__(self, bank_name: str, branch: str, id: int) -> None:
        super().__init__(bank_name, branch)
        self.id = id

class Savings(Account):
    def __init__(self, bank_name: str, branch: str, id: int, duration: int,
        super().__init__(bank_name, branch, id)
        self.duration = duration
        self.money = money

class Current(Account):
    def __init__(self, bank_name: str, branch: str, id: int, balance: float
        super().__init__(bank_name, branch, id)
        self.balance = balance

class Person:
    def __init__(self, name: str, addr: str) -> None:
        self.name = name
        self.addr = addr

class Savings_Holder(Person, Savings):
    def __init__(self, name: str, addr: str, bank_name: str, branch: str, i
        Person.__init__(self, name, addr)
        Savings.__init__(self, bank_name, branch, id, duration, money)

class Current_Holder(Person, Current):
    def __init__(self, name: str, addr: str, bank_name: str, branch: str, i
        Person.__init__(self, name, addr)
        Current.__init__(self, bank_name, branch, id, balance)
```

Private members of the class:

In [9]:

```python
class Account:
    def __init__(self, id: int, name: str, pwd: str) -> None:
        self.id = id
        self.name = name
        self.__pwd = pwd
    def get_pwd(self) -> str:
        return self.__pwd
    def set_pwd(self, pwd: str) -> None:
        self.__pwd = pwd

class Saving_Account(Account):
    type = 'Saving'
    def __init__(self, id: int, name: str, pwd: str) -> None:
        super().__init__(id, name, pwd)
    def get_pwd(self) -> str:
        return self.__pwd


acct = Account(1, 'Snehal', 'act_pwd')
print(acct.__getstate__())
print(acct.id)
print(acct.name)
# print(acct.__pwd)
# AttributeError: 'Account' object has no attribute '__pwd'
print(acct.get_pwd())
acct.set_pwd('account')
print(acct.get_pwd())


acct2 = Saving_Account(1, 'Snehal', 'sav_pwd')
print(acct2.__getstate__())
print(acct2.id)
print(acct2.name)
# print(acct2.__pwd)
# AttributeError: 'Saving_Account' object has no attribute '__pwd'
print(acct2.get_pwd())
acct2.set_pwd('saving')
print(acct2.get_pwd())
```

```
{'id': 1, 'name': 'Snehal', '_Account__pwd': 'act_pwd'}
1
Snehal
act_pwd
account
{'id': 1, 'name': 'Snehal', '_Account__pwd': 'sav_pwd'}
1
Snehal
```

```
---------------------------------------------------------------------
--
AttributeError                            Traceback (most recent call las
t)
Cell In[9], line 36
     33 print(acct2.name)
     34 # print(acct2.__pwd)
     35 # AttributeError: 'Saving_Account' object has no attribute '__pw
d'
---> 36 print(acct2.get_pwd())
     37 acct2.set_pwd('saving')
     38 print(acct2.get_pwd())

Cell In[9], line 16, in Saving_Account.get_pwd(self)
     15 def get_pwd(self) -> str:
---> 16     return self.__pwd

AttributeError: 'Saving_Account' object has no attribute '_Saving_Account
__pwd'
```

4. Polymorphism:

In [10]:
```
class Bird:
    def __init__(self, name: str) -> None:
        self.name = name
    def flight(self) -> None:
        print('Birds can fly, but some cannot..')

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

class Ostrich(Bird):
    def __init__(self, name: str) -> None:
        super().__init__(name)
    def flight(self) -> None:
        super().flight()
        print(self.name, 'cannot fly..')
    def flight(self, name) -> None:
        super().flight()
        print('Hii', name, self.name, 'cannot fly..')

s = Sparrow('Sparrow')
s.flight()

p = Ostrich('Ostrich')
# p.flight()
# TypeError: Ostrich.flight() missing 1 required positional argument: 'name
p.flight('snehal')
```

```
Birds can fly, but some cannot..
Sparrow can fly..
Birds can fly, but some cannot..
Hii snehal Ostrich cannot fly..
```

issubclass() and isinstance():

In [11]:
```
class A:
    pass

class B(A):
    pass

a= A()

b = B()

print(issubclass(A,B))
print(issubclass(B,A))

print(isinstance(a,A))
print(isinstance(a,B))
```

```
False
True
True
False
```

In [ ]:
```

```