🧬 Inheritance in Python – Complete Explanation

Inheritance in Python is a fundamental concept in Object-Oriented Programming (OOP) that allows one class (called the child class or derived class) to acquire properties and behaviors (methods) from another class (called the parent class or base class).

Why is inheritance useful?

- **Code Reusability:** Avoid rewriting common code.
- **Hierarchy:** Establish a relationship between classes.
- **Extensibility:** Add or override features in the child class.

🔹 Parent Class and Child Class

In inheritance, there are mainly two types of classes:

- **Parent Class (Base Class)**
  The class whose properties and methods are inherited.
- **Child Class (Derived Class)**
  The class that inherits properties and methods from another class.

The child class can:

- Use parent class methods
- Add new methods
- Modify existing behavior

🔹 Why Do We Need Inheritance?

Inheritance is used to:

- Avoid writing the same code again and again
- Reduce redundancy
- Improve readability
- Build a real-world relationship between classes

Example:

A *Student* class and a *Teacher* class can inherit from a common *Person* class.

In [1]:
```
# init method --> constructors
class Employee:
    amt=1.20
    def __init__(self,fname,lname,desig='',sal=0):
        self.fname=fname
        self.lname=lname
        self.desig=desig
        self.sal=sal
        self.email=self.fname.lower()+self.lname.lower()+'@zh.du.ac.in'
    def info(self):
        print('Name of Emp\t',self.fname+' '+self.lname)
        print('Designation \t',self.desig)
        print('Email \t\t',self.email)
        print('Salary \t\t',self.sal)
    def apply_raise(self):
        self.sal=self.sal*self.amt
        print('Salary after appraisal\t',self.sal)
```

In [2]:
```
# plan of action
#employee--> child(Developer,Manager)
```

In [3]:
```
#Method overiding
class Developer(Employee):
    #pass
    def __init__(self,fname,lname,desig,sal,lang):
        super().__init__(fname,lname,desig,sal)
        self.lang=lang
    def info(self):
        print('-'*60)
        super().info()
        print('Language\t',self.lang)
        print('-'*60)
```

In [4]:
```
dev1=Developer(fname='Vipul',lname='Pandey',desig='Developer',sal=100000,lang=
dev1.info()
```

```
------------------------------------------------------------
Name of Emp         Vipul Pandey
Designation          Developer
Email                  vipulpandey@zh.du.ac.in
Salary                  100000
Language         Python
------------------------------------------------------------
```

In [5]:
```
 #manager

class Manager(Employee):
    def __init__(self,fname,lname,desig,sal,emp=None):
```

```
        super().__init__(fname,lname,desig,sal)
        self.emp = emp
        if emp==None:
            self.emp= []
        else:
            temp = None
            temp = self.emp
            self.emp = []
            self.emp.append(temp)
    def add_emp(self,cand):
        self.emp.append(cand)
        print(f'{cand} is successfully added to your list.')
    def remove_emp(self,cand):
        if cand not in self.emp:
            print(f'{cand} : Candidate not found.')
        else:
            self.emp.remove(cand)
            print(f'{cand} successfully removed.')
    def replace_emp(self,cand,replacement):
        if cand not in self.emp:
            print('Candidate not found')
        else:
            temp = self.emp.index(cand)
            self.emp[temp] = replacement
            print(f'successfully replaced {cand} with {replacement}')
    def show_all(self):
        count =1
        for i in self.emp:
            print(count,'.',i)
            count+=1
```

In [6]:
```
man1=Manager('Vipul','Pandey','Manager',100000,'Ram')
```

In [7]:
```
man1.emp
```

Out[7]:
```
['Ram']
```

In [8]:
```
man1.add_emp('Gaurav')
man1.add_emp('Raju')
man1.add_emp('Harsh')
```

```
Gaurav is successfully added to your list.
Raju is successfully added to your list.
Harsh is successfully added to your list.
```

In [9]:
```
man1.emp
```

Out[9]:
```
['Ram', 'Gaurav', 'Raju', 'Harsh']
```

In [10]:
```
man1.remove_emp('soorya')
man1.remove_emp('Harsh')
```

```
soorya : Candidate not found.
Harsh successfully removed.
```

```
In [11]: man1.emp
```

```
Out[11]: ['Ram', 'Gaurav', 'Raju']
```

```
In [12]: man1.replace_emp('Raju','Anshul')
```

```
successfully replaced Raju with Anshul
```

```
In [13]: man1.emp
```

```
Out[13]: ['Ram', 'Gaurav', 'Anshul']
```

```
In [14]: man1.show_all()
```

```
1 . Ram
2 . Gaurav
3 . Anshul
```

Types of Inheritance in Python:

- **Single Inheritance** – One child inherits from one parent.
- **Multiple Inheritance** – One child inherits from multiple parents.
- **Multilevel Inheritance** – Child -> Parent -> Grandparent.
- **Hierarchical Inheritance** – Multiple children inherit from one parent.
- **Hybrid Inheritance** – Combination of the above.

🔹 Single Inheritance

Single inheritance occurs when:

- One child class inherits from **one parent class**

Features:

- Simple and easy to understand
- Improves code reuse
- Commonly used in beginner-level programs

Use case:

- A class extends basic functionality from one base class

🔹 Multiple Inheritance

Multiple inheritance occurs when:

- One child class inherits from **more than one parent class**

Features:

- Child class gets access to multiple class features
- Useful when combining different functionalities
- Must be handled carefully to avoid confusion

Important concept:

- Method Resolution Order (MRO) decides which method is called first

---

🔹 Multilevel Inheritance

Multilevel inheritance occurs when:

- A class inherits from a class that already inherits another class

Structure:

- Grandparent → Parent → Child

Features:

- Represents real-life hierarchy
- Each level adds more features
- Improves modularity

🔹 Hierarchical Inheritance

Hierarchical inheritance occurs when:

- Multiple child classes inherit from **one parent class**

Features:

- One base class shared by many subclasses

- Helps maintain common behavior
- Reduces duplication

Example use case:

- One base class for different types of users

🔹 Hybrid Inheritance

Hybrid inheritance is:

- A combination of two or more types of inheritance

Features:

- Complex structure
- Powerful but harder to manage
- Uses MRO to resolve conflicts

Used in advanced-level applications.

🔹 Method Overriding

Method overriding happens when:

- A child class defines a method with the **same name** as the parent class

Purpose:

- To change or extend parent class behavior
- Allows customization in child class

This supports **runtime polymorphism**.

🔹 Use of super()

The `super()` concept is used to:

- Access parent class methods and attributes
- Avoid code duplication

- Maintain proper inheritance flow

It ensures that parent class functionality is not lost.

🔹 Constructor in Inheritance

Constructors can also be inherited.

- Parent class constructor can be accessed
- Child class can have its own constructor
- Both can work together using proper calling

This helps in initializing data properly.

🔹 Method Resolution Order (MRO)

MRO defines:

- The order in which Python searches for methods in multiple inheritance

Important points:

- Python follows C3 linearization
- MRO avoids ambiguity
- Can be checked to understand execution flow

🔹 Advantages of Inheritance

- Code reuse
- Easy maintenance
- Logical class structure
- Faster development
- Better scalability

🔹 Disadvantages of Inheritance

- Increases complexity
- Tight coupling between classes

- Difficult debugging in deep inheritance

Use inheritance only when there is a **clear relationship**.

🔹 Real-World Use Cases

Inheritance is used in:

- Banking systems
- Employee management systems
- Game development
- Data science class structures
- Framework and library design

In [15]:
```
# single inheritance
class parent:
    print('parent')

class child(parent):
    pass
```

```
parent
```

In [16]:
```
# multiple inheritance

class papa:
    pass
class mummy:
    pass

class child(papa,mummy):
    pass

# help(child)
```

In [17]:
```
# mutlilevel inheritance

class grandparents:
    pass
class parents(grandparents):
    pass
class child(parents):
    pass
```

In [18]:
```
# Hierarchical Inheritance

class parent:
```

```
    pass

class child1(parent):
    pass
class child2(parent):
    pass
```

✅ Final Summary

Inheritance is a core pillar of **Object-Oriented Programming**.
It helps build structured, reusable, and maintainable code.
Understanding inheritance is essential for writing **professional and scalable Python applications**.

In [ ]: