Inheritance

- In general we human beings always know about inheritance.
- In programming it is almost the same. When a class inherits another class it inherits all features (like variables and methods) of the parent class.
- This helps in reusing codes.
- The way we inherited a few qualities from our parents similarly, a class can also inherit the qualities from a parent class.
- For eg: A Phone Class can have two Child Classes: 1) TelePhone and 2) MobilePhone. Both can inherit the "calling" behaviour.

Parent-class

- The class from which a class inherits is called the parent-class.
- Parent-class are often referred with different names - `parent-class` / `super-class` / `base-class` / `ancestor-class`

Child-class

- A class which inherits from a parent-class is called a child-class.
- A child-class are often referred with different names - `child-class` / `sub-class` / `derived-class` / `heir class`

For example in following diagram

- *class* `Vehicle` *is super-class of all the sub-classes (Bikes, Cars, Buses, Trains etc).*
- *class* `Cars` *is super-class for sub-classes* `Hatchbacks` *and* `Sedans`

```
graph TD;
Vehicle --> Bikes;
```

```
Vehicle --> Cars --> Hatchbacks;
Vehicle --> Cars --> Sedans;
Vehicle --> Buses;
Vehicle --> Trains;
```

Syntax

```
# The syntax for a subclass definition looks like this:

class ChildClass:              # super-class derived from default `object` class
    pass

class ParentClass(ChildClass): # sub-class derived from parent-class
    pass
```

Basic Inheritance example

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

    def greetings(self):
        print("Good morning, I am " + self.name)


class Engineer(Person):
    pass
```

```
>>> x = Person("Prashant")
>>> y = Engineer("Rahul")
>>> print(x, type(x))
<__main__.Person object at 0x1024d2220> <class '__main__.Person'>

>>> print(y, type(y))
<__main__.Engineer object at 0x1023c3d00> <class '__main__.Engineer'>

>>> y.greetings()
Good morning, I am Rahul
```

💡 When there isn't any method defined in the `child-class`, Python looks-up for the method definition in upward direction first in class hierarchy.

`Overriding` vs `Overloading`

- `Method overloading` is defining function with same name in both `parent-class` and `child-class` but with different `function signature`.
- Meaning, function signature in child-class may have **different number of parameters than parent-class**.
- *Below is the example of function overloading -*

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

    def greetings(self, nick_name, age):
        print(
            f"Good morning, I am  {self.name} ({nick_name} {age})"
        )


class Engineer(Person):
    def greetings(self, nick_name):
        print(
            f"Good morning, I am  {self.name} ({nick_name})"
        )
```

```
>>> y = Engineer("Virat")
>>> y.greetings("Viru")  # func `greetings` from child will be called

Good morning, I am  Virat (Viru)
```

- `Method overriding` is defining a function with same name and same number of parameters in both `parent-class` and `child-class`
- `Function signature` is same in both `parent-class` and `child-class`

- Essentially in method overriding, we just re-define the function under derived-class (which is already defined in the derived-class).
- *Below is the example of function overriding -*

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

    def greetings(self):
        print("Good morning", self.name)


class Engineer(Person):
    def greetings(self):
        print("hey - GOOD EVENING,", self.name)
```

```
>>> y = Engineer("Rahul")
>>> y.greetings()           # message from child will be printed
hey - GOOD EVENING, Rahul
```

Multiple inheritance

- When child-class is derived from multiple base-classes, it is called as multiple inheritance
- Syntax -

```
class ChildClass(BaseClass1, BaseClass2, BaseClass3):
    pass
```

Diamond Problem

- Diamond problem occurs when two or more child-classes have common ancestor-class.

- It means there are multiple paths to reach to the parent-class.
- In such case, `Python` follows certain algorithm (`C3 linearization algorithm`)to resolve method calls.
- In `python3`, the highest super-class is visited when all local classes underneath it are already visited.

```
graph TD;

Person--> Mother --> Child;
Person--> Father--> Child;
```

```
class GrantParent:
    def get_details(self):
        print("I will be only called once all my child classes are visited")


class Mother(GrantParent):
    def __init__(self):
        self.height = "tall"
        self.color = "white"


class Father(GrantParent):
    def __init__(self):
        self.height = "short"
        self.color = "brown"


class Child(Mother, Father):
    pass
```

```
>>> child = Child()
>>> child.get_details()
I will be only called once all my child classes are visited
```

Let's understand `MRO` better.

Method Resolution Order (MRO)

```
graph TD;

Person--> GrandFather_Mother --> Mother --> Child;
Person--> GrandFather_Father --> Father --> Child;
```

- Way of performing look-up for parent class methods in multiple inheritance.
- To keep the base classes from being accessed more than once, the dynamic algorithm **linearizes** the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once.
- `Python` follows algorithm called `C3 linearization algorithm` to resolve method calls.
- Python2.7 had DFLR algorith to perform MRO
- Python3.* has algorithm for MRO called "C3 Linearization algorithm"
- It works almost similar to DFLR depth-first, left to right except for the following difference -
- highest super-class will be only visited when all local classes are visited.

```
class Person:
    def get_details(self):
        print("method from class - Person")

class GrandFather_Father(Person):
    pass

class GrandFather_Mother(Person):
    pass

class Mother(GrandFather_Mother):
    pass

class Father(GrandFather_Father):
    def get_details(self):
        print("Father")

class Child(Mother, Father):
    pass
```

```
>>> child = Child()
>>> child.get_details()
Father
```

- **How to get MRO (Method Resolution Order) associated with a class?**

Let's see considering above example -

```
>>> Child.mro()
[__main__.Child,
 __main__.Mother,
 __main__.GrandFather_Mother,
 __main__.Father,
 __main__.GrandFather_Father,
 __main__.Person,
 object]
```