Polymorphism

Polymorphism refers to a function having the same name but being used in different ways and different scenarios. This makes programming easier and more intuitive.

It can be implemented by Overloading and Overriding

Method Overriding

In [1]:
```
def add(a,b):
  print(a + b)
```

In [2]:
```
def add(a,b,c):
  print(a + b + c)
```

In [3]:
```
add(8,2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-1f5bee9b1427> in <module>
----> 1 add(8,2)

TypeError: add() missing 1 required positional argument: 'c'
```

In [4]:
```
add(3,4,5)
```

```
12
```

As seen above, the second add function with 3 parameters, overrides the first add function with two parameters

Operator Overloading

Python operators work for predefined data types like int, str, list, etc, but we can change the way an operator works depending on the types of operands that we use. We may use any inbuilt or user-defined operand. This is the feature of operator overloading in Python that allows the same built-in operator to behave differently according to the context of the implementation of a problem.

Operator overloading in Python provides the ability to override the functionality of a built-in operator in user-defined classes.

For example, the “*” operator can be overloaded not only as a multiplier for numbers but also as a repetition operator for lists or strings.

Operator overloading is also known as Operator Ad-hoc Polymorphism.

In [5]:

```
class Point:

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __str__(self):
    return 'X ' + str(self.x) + ' Y ' + str(self.y)

  def __add__(self, other):
    x = self.x + other.x
    y = self.y + other.y
    return Point(x, y)

  def __sub__(self, other):
    x = self.x - other.x
    y = self.y - other.y
    return Point(x, y)

  def __mul__(self, other):
    x = self.x * other.x
    y = self.y * other.y
    return Point(x, y)

  def __truediv__(self, other):
    x = self.x / other.x
    y = self.y / other.y
    return Point(x, y)

  def __floordiv__(self,other):
    x = self.x // other.x
    y = self.y // other.y
    return Point(x, y)

  def __mod__(self,other):
    x = self.x % other.x
    y = self.y % other.y
    return Point(x, y)

  def __pow__(self, other):
    x = self.x ** other.x
    y = self.y ** other.y
    return Point(x, y)

  def __lt__(self, other):
    if self.x < other.x:
      if self.y < other.y:
        return True
    return False

  def __gt__(self, other):
    if self.x > other.x:
      if self.y > other.y:
        return True
    return False

  def __ge__(self, other):
    if self.x >= other.x:
      if self.y >= other.y:
        return True
    return False
```

```
  def __le__(self, other):
    if self.x <= other.x:
      if self.y <= other.y:
        return True
    return False

  def __ne__(self, other):
    if self.x != other.x:
      if self.y != other.y:
        return True
    return False

  def __eq__(self, other):
    if self.x == other.x:
      if self.y == other.y:
        return True
    return False
```

In [6]:
```
p1 = Point(6,31)
p2 = Point(3,4)
```

In [7]:
```
print(p1)
```

```
X 6 Y 31
```

In [8]:
```
print(p2)
```

```
X 3 Y 4
```

In [9]:
```
p3 = p1 + p2
print(p3)
```

```
X 9 Y 35
```

In [10]:
```
p3 = p1 - p2
print(p3)
```

```
X 3 Y 27
```

In [11]:
```
p3 = p1 * p2
print(p3)
```

```
X 18 Y 124
```

In [12]:
```
p3 = p1 / p2
print(p3)
```

```
X 2.0 Y 7.75
```

In [13]:
```
p3 = p1 // p2
print(p3)
```

```
X 2 Y 7
```

In [14]:
```
p3 = p1 ** p2
print(p3)
```

```
X 216 Y 923521
```

In [15]:
```
p3 = p1 > p2
print(p3)
```

```
True
```

In [16]:
```
p3 = p1 < p2
print(p3)
```

```
False
```

In [17]:
```
p3 = p1 >= p2
print(p3)
```

```
True
```

In [18]:
```
p3 = p1 <= p2
print(p3)
```

```
False
```

In [19]:
```
p3 = p1 != p2
print(p3)
```

```
True
```

In [20]:
```
p3 = p1 == p2
print(p3)
```

```
False
```

Inheritance

Inheritance is the ability to 'inherit' features or attributes from already written classes into newer classes we make. These features and attributes are defined data structures and the functions we can perform with them, a.k.a. Methods. It promotes code reusability, which is considered one of the best industrial coding practices as it makes the codebase modular.

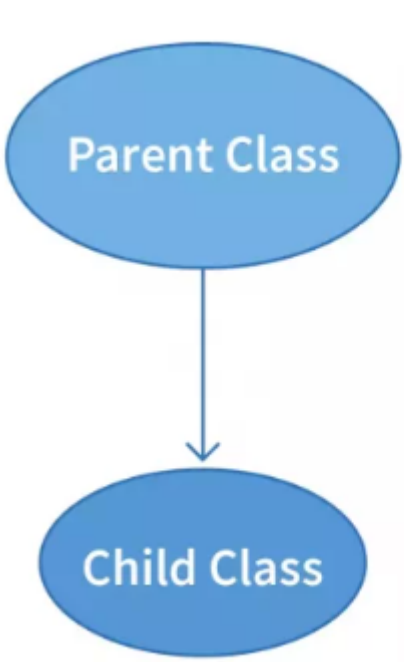
In python inheritance, new class/es inherits from older class/es. The new class/es copies all the older class's functions and attributes without rewriting the syntax in the new class/es. These new classes are called derived classes, and old ones are called base classes.

For example, inheritance is often used in biology to symbolize the transfer of genetic traits from parents to their children. Similarly, we have parent classes (Base classes) and child classes (derived classes). In Inheritance, we derive classes from other already existing classes. The existing classes are the parent/base classes from which the attributes and methods are inherited in the child classes.

Types of Inheritance

Single Inheritance

Single Inheritance is the simplest form of inheritance where a single child class is derived from a single parent class. Due to its candid nature, it is also known as Simple Inheritance.



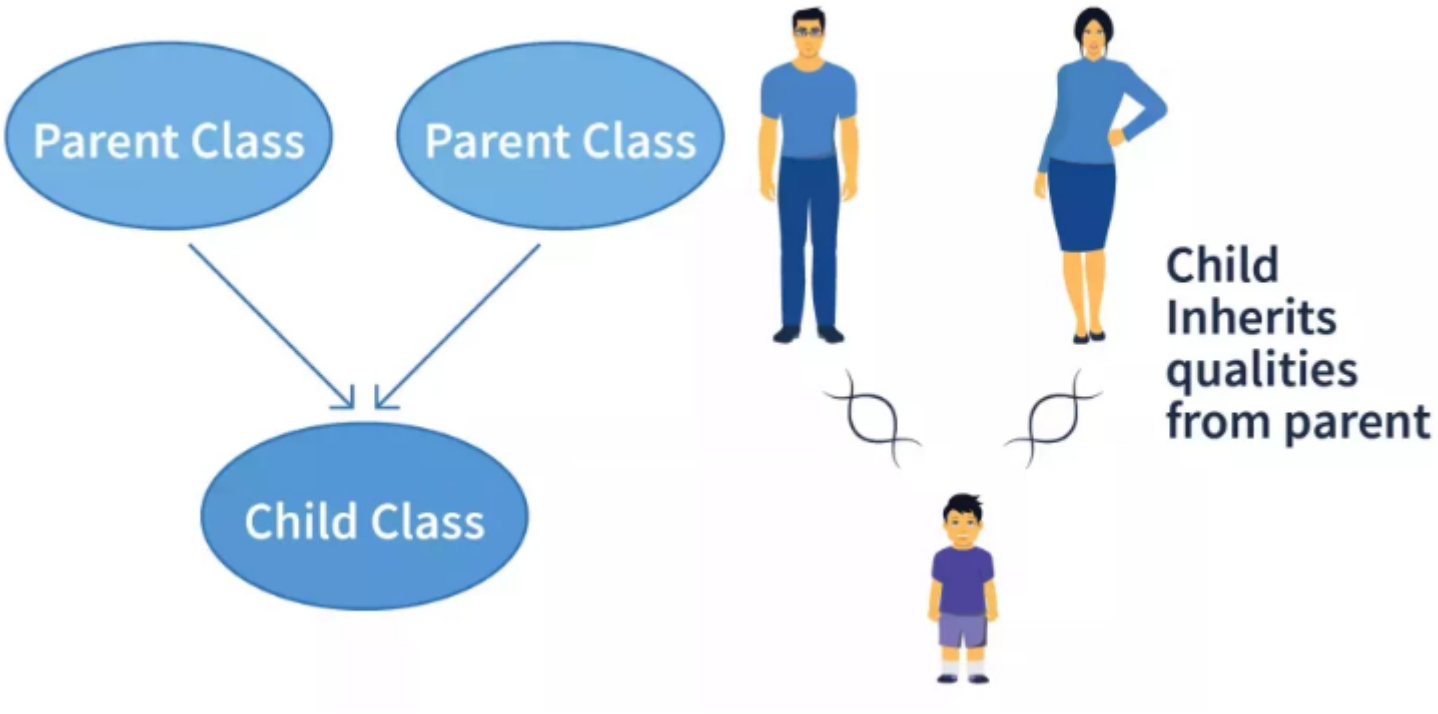
In [21]:
```
# python 3 syntax
# single inheritance example

class parent:                   # parent class
    def func1(self):
        print("Hello Parent")

class child(parent):
  # child class
    def func2(self):                # we include the parent class
        print("Hello Child")    # as an argument in the child
                                # class

# Driver Code
test = child()                  # object created
test.func1()                    # parent method called via child object
test.func2()                    # child method called
```

```
Hello Parent
Hello Child
```

Multiple Inheritance in Python

In multiple inheritance, a single child class is inherited from two or more parent classes. It means the child class has access to all the parent classes' methods and attributes.

However, if two parents have the same “named” methods, the child class performs the method of the first parent in order of reference. To better understand which class’s methods shall be executed first, we can use the Method Resolution Order function (mro). It tells the order in which the child class is interpreted to visit the other classes.



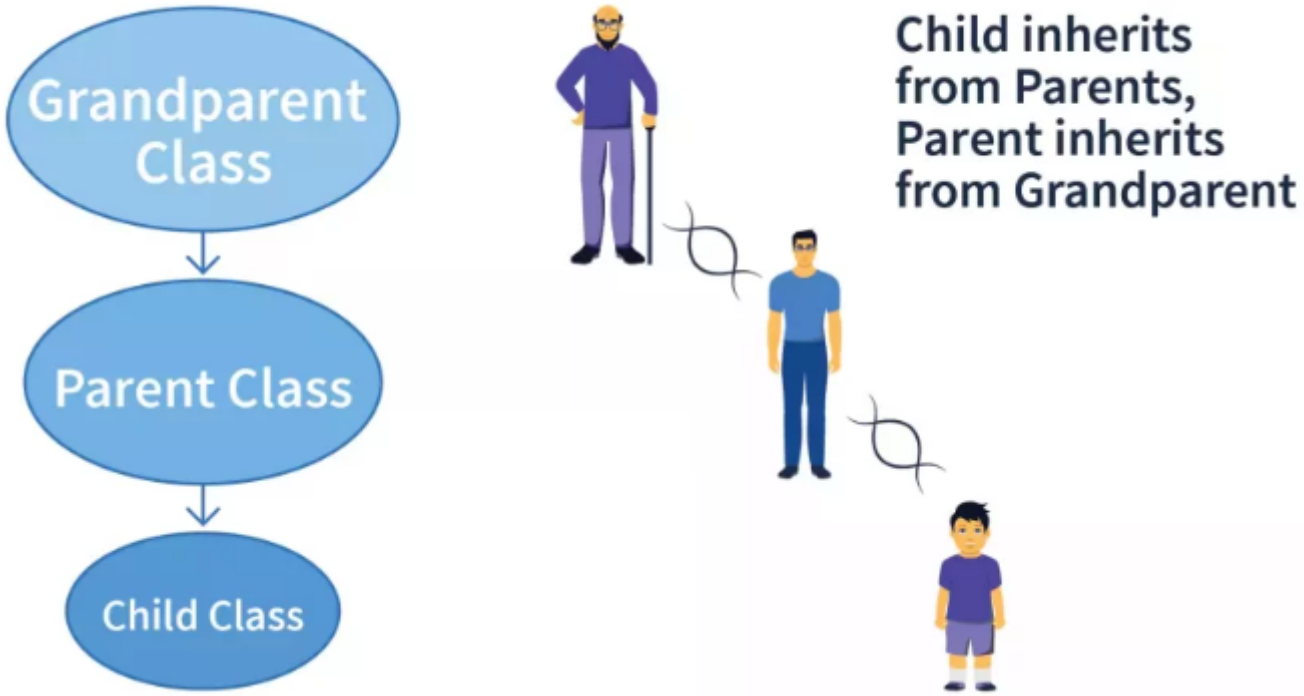
In [22]:
```
# python 3 syntax
# multiple inheritance example

class parent1:                      # first parent class
    def func1(self):
        print("Hello Parent1")

class parent2:                      # second parent class
    def func2(self):
        print("Hello Parent2")

class parent3:                      # third parent class
    def func2(self):                    # the function name is same as parent2
        print("Hello Parent3")

class child(parent1, parent2, parent3):     # child class
    def func3(self):                    # we include the parent classes
        print("Hello Child")        # as an argument comma separated

# Driver Code
test = child()          # object created
test.func1()            # parent1 method called via child
test.func2()            # parent2 method called via child instead of parent3
test.func3()            # child method called

# to find the order of classes visited by the child class, we use __mro__ on the chil
d class
print(child.__mro__)
```

```
Hello Parent1
Hello Parent2
Hello Child
(<class '__main__.child'>, <class '__main__.parent1'>, <class '__main__.parent2'>, <
class '__main__.parent3'>, <class 'object'>)
```

As we can see with the help of mro, the child class first visits itself, then the first parent class, referenced before the second parent class. Similarly, it visits the second parent class before the third parent class, and that's why it performs the second parent's function rather than the third parent's. Finally, it visits any objects that may have been created.

Multilevel Inheritance in Python

In multilevel inheritance, we go beyond just a parent-child relation. We introduce grandchildren, great-grandchildren, grandparents, etc. We have seen only two levels of inheritance with a superior parent class/es and a derived class/es, but here we can have multiple levels where the parent class/es itself is derived from another class/es.

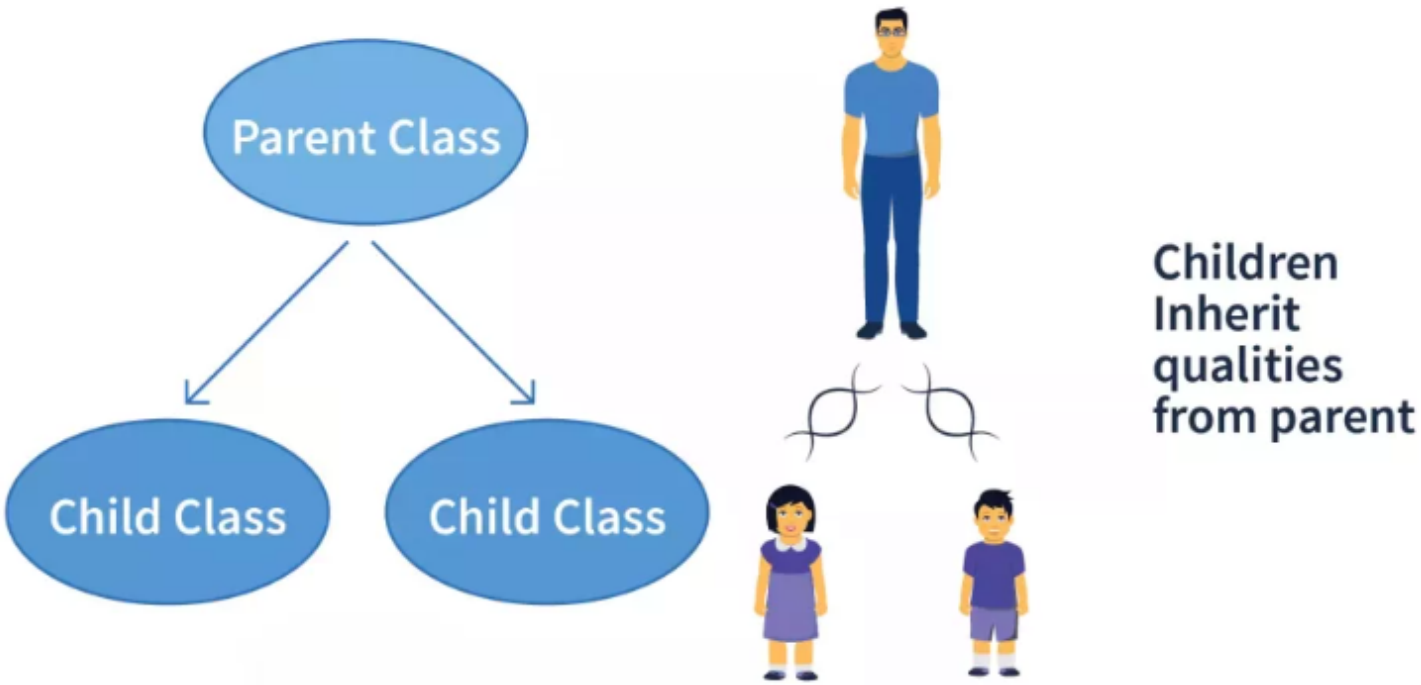


In [24]:
```
class grandparent:                  # first level
    def func1(self):
        print("Hello Grandparent")

class parent(grandparent):          # second level
    def func2(self):
        print("Hello Parent")

class child(parent):                # third level
    def func3(self):
        print("Hello Child")


# Driver Code
test = child()                      # object created
test.func1()                        # 3rd level calls 1st level
test.func2()                        # 3rd level calls 2nd level
test.func3()                        # 3rd level calls 3rd level

print(child.__mro__)
```

```
Hello Grandparent
Hello Parent
Hello Child
(<class '__main__.child'>, <class '__main__.parent'>, <class '__main__.grandparen
t'>, <class 'object'>)
```

Hierarchical Inheritance in Python

Hierarchical Inheritance is the right opposite of multiple inheritance. It means that, there are multiple derived child classes from a single parent class.

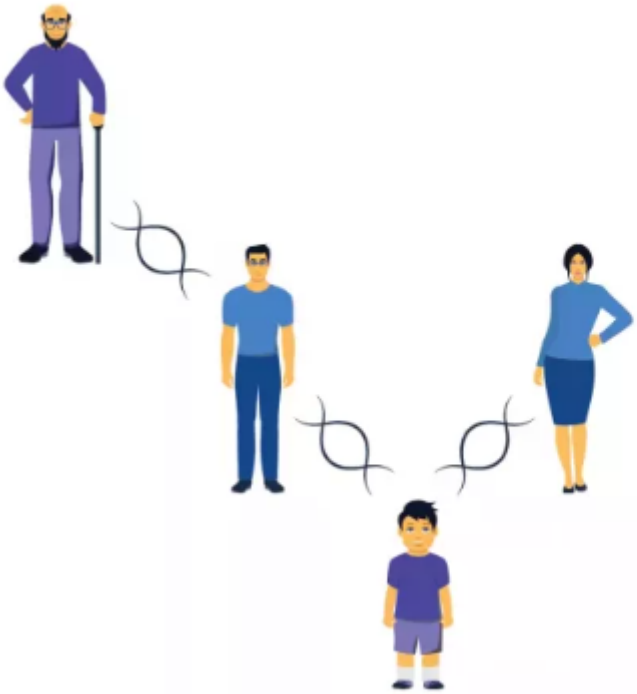


In [26]:
```
# python 3 syntax
# hierarchical inheritance example

class parent:                       # parent class
    def func1(self):
        print("Hello Parent")

class child1(parent):               # first child class
    def func2(self):
        print("Hello Child1")


class child2(parent):               # second child class
    def func3(self):
        print("Hello Child2")


# Driver Code
test1 = child1()                     # objects created
test2 = child2()

test1.func1()                       # child1 calling parent method
test1.func2()                       # child1 calling its own method

test2.func1()                       # child2 calling parent method
test2.func3()                       # child2 calling its own method
print(child1.__mro__)
print(child2.__mro__)
```

```
Hello Parent
Hello Child1
Hello Parent
Hello Child2
(<class '__main__.child1'>, <class '__main__.parent'>, <class 'object'>)
(<class '__main__.child2'>, <class '__main__.parent'>, <class 'object'>)
```

Hybrid Inheritance in Python

Hybrid Inheritance is the mixture of two or more different types of inheritance. Here we can have many relationships between parent and child classes with multiple levels.



Child inherits from grandparents and parents

In [27]:
```
# python 3 syntax
# hybrid inheritance example

class parent1:                              # first parent class
    def func1(self):
        print("Hello Parent")


class parent2:                              # second parent class
    def func2(self):
        print("Hello Parent")

class child1(parent1):                      # first child class
    def func3(self):
        print("Hello Child1")


class child2(child1, parent2):              # second child class
    def func4(self):
        print("Hello Child2")


# Driver Code
test1 = child1()                            # object created
test2 = child2()

test1.func1()                           # child1 calling parent1 method
test1.func3()                           # child1 calling its own method

test2.func1()                           # child2 calling parent1 method
test2.func2()                           # child2 calling parent2 method
test2.func3()                           # child2 calling child1 method
test2.func4()                           # child2 calling its own method
print(child1.__mro__)
print(child2.__mro__)
```

```
Hello Parent
Hello Child1
Hello Parent
Hello Parent
Hello Child1
Hello Child2
(<class '__main__.child1'>, <class '__main__.parent1'>, <class 'object'>)
(<class '__main__.child2'>, <class '__main__.child1'>, <class '__main__.parent1'>, <
class '__main__.parent2'>, <class 'object'>)
```

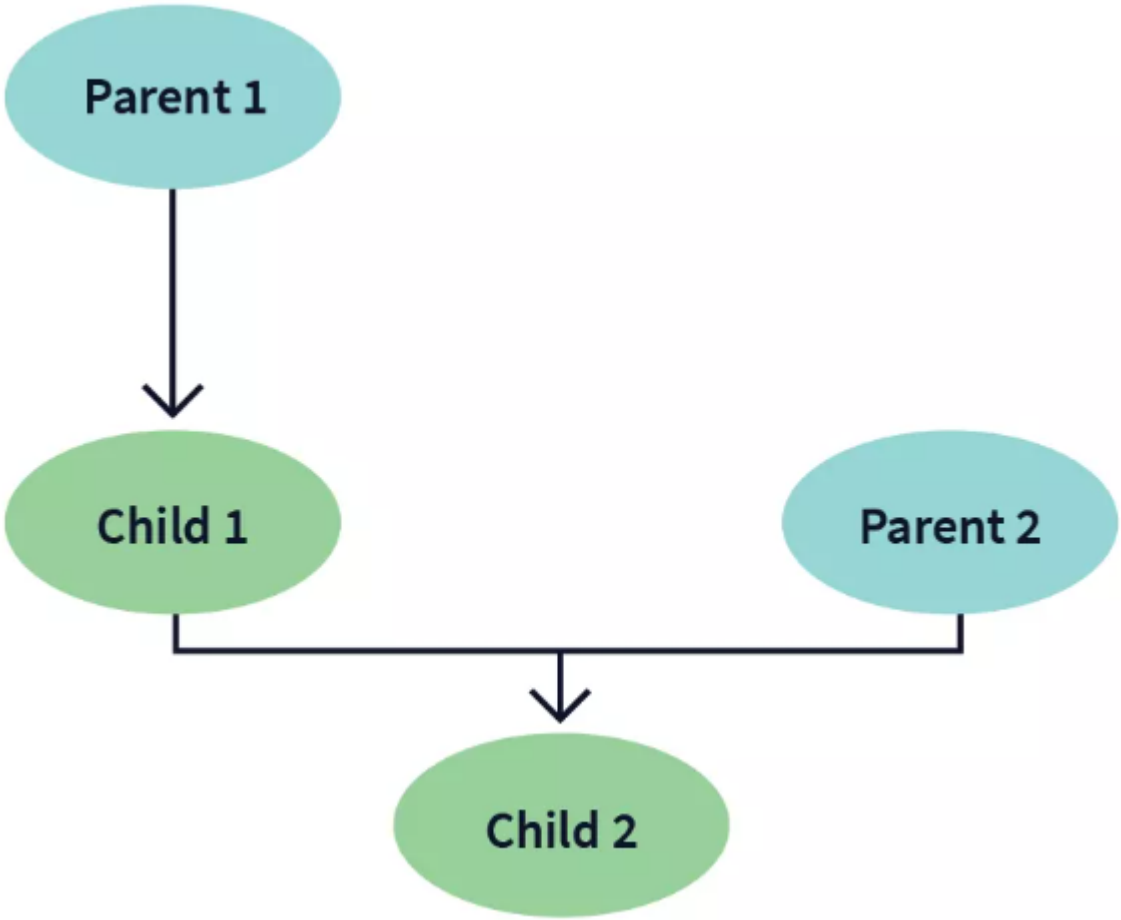
This example shows a combination of three types of python inheritance.

Parent1 -> Child1 : Single Inheritance

Parent1 -> Child1 -> Child2 : Multi – Level Inheritance

Parent1 -> Child2 <- Parent2 : Multiple Inheritance

The diagrammatic explanation of this hybrid inheritance is:



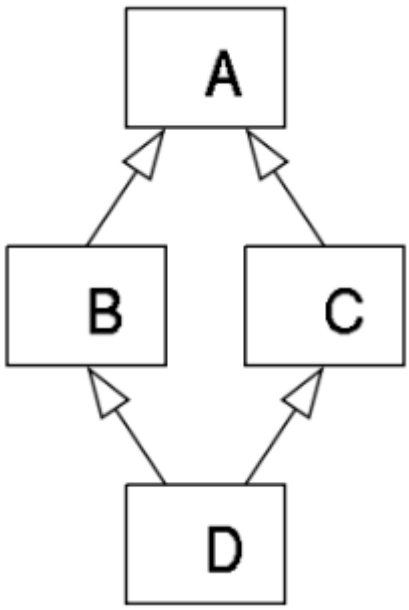
MRO -Method Resolution Order

Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass. In python, method resolution order defines the order in which the base classes are searched when executing a method. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. This order is also called Linearization of a class and set of rules are called MRO(Method Resolution Order). While inheriting from another class, the interpreter needs a way to resolve the methods that are being called via an instance. Thus we need the method resolution order. For Example

In [34]:
```
# Python program showing
# how MRO works

class A:
        def test(self):
                print(" In class A")
class B(A):
        def test(self):
                print(" In class B")
class C(A):
        def test(self):
                print("In class C")

# classes ordering
class D(B, C):
        pass

obj = D()
obj.test()
print(D.__mro__)
```

```
 In class B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.
A'>, <class 'object'>)
```

In the above example we use multiple inheritances and it is also called Diamond inheritance and it looks as follows:



Python follows a depth-first lookup order and hence ends up calling the method from class A. By following the method resolution order, the lookup order as follows. Class D -> Class B -> Class C -> Class A Python follows depth-first order to resolve the methods and attributes. So in the above example, it executes the method in class B.

Special Functions in Python Inheritance

super() function

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method already provided by one of its super-classes or parent classes. This discrepancy is caused due to similar naming convention of the methods. Commonly we can see this situation when the parent's init() is overridden by the child's init(), and hence the child class cannot inherit attributes from the parent class.

Similarly, we can face this same problem with methods other than init but having the same naming convention across parent and child classes.

One solution is to call the parent method inside the child method.

In [28]:
```
# python 3 syntax
# solution to method overriding - 1

class parent:                       # parent class

    def __init__(self):             # __init__() of parent
        self.attr1 = 50
        self.attr2 = 66

class child(parent):                # child class

    def __init__(self):             # __init__() of child
        parent.__init__(self)       # calling parent's __init__()
        self.attr3 = 45

test = child()                      # object initiated

print (test.attr1)                  # parent attribute called via child
```

```
50
```

Another way to solve this problem without explicitly typing out the parent name is to use super(). It automatically references the parent/base class from which the child class is derived. It is extremely helpful to call overridden methods in classes with many methods.

In [29]:
```python
# python 3 syntax
# solution to method overriding - 2

class parent:                       # parent class

    def display(self):              # display() of parent
        print("Hello Parent")

class child(parent):                # child class

    def display(self):              # display() of child
        super().display()           # referencing parent via super()
        print("Hello Child")

test = child()                      # object initiated

test.display()                      # display of both activated
```

```
Hello Parent
Hello Child
```

issubclass() and isinstance()

issubclass(): The issubclass() function is a convenient way to check whether a class is the child of the parent class. In other words, it checks if the first class is derived from the second class. If the classes share a parent-child relationship, it returns a boolean value of True. Otherwise, False.

isinstance(): isinstance() is another inbuilt function of python which allows us to check whether an object is an instance of a particular class or any of the classes it has been derived from. It takes two parameters, i.e. the object and the class we need to check it against. It returns a boolean value of True if the object is an instance and otherwise, False.

In [31]:
```
# python 3 syntax
# issubclass() and isinstance() example

class parent:                    # parent class
    def func1():
        print("Hello Parent")

class child(parent):             # child class
    def func2():
        print("Hello Child")


# Driver Code

print(issubclass(child,parent))          # checks if child is subclass of parent

print(issubclass(parent,child))          # checks if parent is subclass of child

A = child()                       # objects initialized
B = parent()

print(isinstance(A,child))                # checks if A is instance of child
print(isinstance(A,parent))               # checks if A is instance of parent
print(isinstance(B,child))                # checks if B is instance of child
print(isinstance(B,parent))             # checks if B is instance of parent
```

```
True
False
True
True
False
True
```

Advantages of Inheritance in Python

Modular Codebase: Increases modularity, i.e. breaking down codebase into modules, making it easier to understand. Here, each class we define becomes a separate module that can be inherited separately by one or many classes.

Code Reusability: the child class copies all the attributes and methods of the parent class into its class and use. It saves time and coding effort by not rewriting them, thus following modularity paradigms.

Less Development and Maintenance Costs: changes need to be made in the base class, all derived classes will automatically follow.

Disadvantages of Inheritance in Python

Decreases the Execution Speed: loading multiple classes because they are interdependent

Tightly Coupled Classes: this means that even though parent classes can be executed independently, child classes cannot be executed without defining their parent classes.

Abstract Class

Abstraction is the concept in object-oriented programming that is used to hide the internal functionality of the classes from the users. Abstraction is implemented using the abstract classes. An abstract class in Python is typically created to declare a set of methods that must be created in any child class built on top of this abstract class. Similarly, an abstract method is one that doesn't have any implementation.

In abstraction, the users are familiar with the purpose of the class's methods, but they don't know how they solve the purpose, which means that they know the inputs and expected outputs, but the inner working is hidden.

There are plenty of real-life examples, like when we press a button on a TV remote to change the channel; we don’t know how it does that. We are just interested in the fact that when we press a particular button, it changes the channel.

Smartphones are another example of abstraction; we are unaware of any of the internal functionalities of the phone. We are just concerned with what we need to do to do a task.

For example, we don’t know how the phone records a video by pressing the record button, we just touch/press a button, and it does that. There are numerous other examples of abstraction for us to observe in the real world.

We can deduce from the above examples that abstraction helps us make everything more user-friendly and less complex. In the context of programming, Abstraction is used to make the life of other developers easy. For example, in Python we do not know how the sort() function of the list class works, we just use this function, and it sorts a list for us.

Although Python is not a fully object-oriented programming language, as we can write code in Python without creating any classes, it supports all the features of object-oriented programming including abstraction and abstract classes.

We cannot create an abstract class in Python directly. However, Python does provide a module that allows us to define abstract classes. The module we can use to create an abstract class in Python is abc(abstract base class) module.

Abstract methods force the child classes to give the implementation of these methods in them and thus help us achieve abstraction as each subclass can give its own implementation. A class containing one or more than one abstract method is called an abstract class.

We can use the following syntax to create an abstract class in Python:

In [36]:
```
from abc import ABC
class DemoAbstractClass(ABC):
        pass
```

To define an abstract method we use the @abstractmethod decorator of the abc module. It tells Python that the declared method is abstract and should be overridden in the child classes.

We can use the following syntax to create an abstract method in Python:

In [38]:
```
from abc import ABC, abstractmethod
class DemoAbstractClass(ABC):
    @abstractmethod
    def abstract_method_name(self):
        pass
```

Now, we have seen how to create an abstract class in Python. Let’s take a very common example to illustrate the use of abstract classes and methods.

Let’s say we want to draw different shapes. To do this, we create a basic abstract class Shape which would contain the blueprint for specific shapes to follow. This class will provide the methods that are needed to be implemented by its child classes for specific shapes.

Let’s create the Shape class first:

In [39]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractmethod
    def draw(self):
        pass
```

As you may say that we have declared an undefined method draw(). This method will be implemented by the other classes that inherit this class.

For example, If we want to draw a circle shape. We create a class Circle that will inherit the Shape class and implement the draw() method. Let's see the implementation of the Circle class:

In [41]:
```
class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw(self):
        print("Drawing a Circle")
```

Now, let's say we want to draw another shape, say a Triangle. We create a class Triangle similar to the Circle class that will inherit the Shape class and implements the draw() method. Let’s see the implementation of the Triangle class:

In [42]:
```
class Triangle(Shape):

    def __init__(self):
        super().__init__("triangle")

    def draw(self):
        print("Drawing a Triangle")
```

Now, let's create an object of both shapes and call their draw() methods.

In [44]:
```
#create a circle object
circle = Circle()
circle.draw()

#create a triangle object
triangle = Triangle()
triangle.draw()
```

```
Drawing a Circle
Drawing a Triangle
```

Why Use Abstract Base Classes?

As we have discussed above, abstract classes are used to create a blueprint of our classes as they don't contain the method implementation. This is a very useful capability, especially in situations where child classes should provide their own separate implementation. Also, in complex projects involving large teams and a huge codebase, It is fairly difficult to remember all the class names.

The importance of using abstract classes in Python is that if our subclasses don’t follow that blueprint, Python will give an error. Thus we can make sure that our classes follow the structure and implement all the abstract methods defined in our abstract class.

Let’s take an example to understand this. Suppose in our example of the Circle class, we do not define the draw() method; instead, we define a draw_circle() method, which is doing the same thing as the draw().

In [48]:
```
class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw_circle(self):
        print("drawing circle")
```

In [49]:
```
c = Circle()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-902db13f64bb> in <module>
----> 1 c = Circle()

TypeError: Can't instantiate abstract class Circle with abstract methods draw
```

So, by using the abstract classes, we can ensure that our subclasses use the same structure and same method names for similar tasks.

Also, by using the abstract classes, we can hide unnecessary details from the user and reduce the programming complexity to a great extent.

Abstract Properties

Abstract class in Python also provides the functionality of abstract properties in addition to abstract methods. Properties are Pythonic ways of using getters and setters. The abc module has a @abstractproperty decorator to use abstract properties. Just like abstract methods, we need to define abstract properties in implementation classes. Otherwise, Python will raise the error.

As we have been told, properties are used in Python for getters and setters. Abstract property is provided by the abc module to force the child class to provide getters and setters for a variable in Python.

Let's define an abstract property in our Shape class:

In [51]:
```
from abc import ABC, abstractmethod, abstractproperty

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractproperty
    def name(self):
        pass

    @abstractmethod
    def draw(self):
        pass
```

In [55]:
```
class Circle(Shape):
  def __init__(self):
    super().__init__("circle")
  @property
  def name(self):
    return self.shape_name
  def draw(self):
    print("Drawing a Circle")
```

In [56]:
```
circle = Circle()
print(f"The shape name is: {circle.name}")
```

```
The shape name is: circle
```

Invoke Methods from Abstract Classes

Unlike some other programming languages, abstract Python methods don't need to be completely abstract; they can have some basic-level implementation that can be used by all the concrete classes. A Concrete class is a class that has a definition for all its methods and has no abstract method.

Concrete classes can use the super() to call the base abstract method and do some additional tasks into it. In our example, the abstract method draw() of the Shape class could be used to prepare a basic canvas, and then concrete classes could work on top of that to draw specific shapes. Let's see this in the code:

In [57]:
```
from abc import ABC, abstractmethod, abstractproperty

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractmethod
    def draw(self):
        print("Preparing the Canvas")
```

As you can see, instead of declaring an empty method, we have defined high-level implementation in the draw() method.

Let's see the Circle class now:

In [58]:
```
class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw(self):
        super().draw()
        print("Drawing a Circle")
```

Here, we have first called the draw() method of the Shape class through super().draw() and then provided the additional implementation required.

Let's try to call the draw() method of Circle class again:

In [59]:
```
#create a circle object
circle = Circle()
circle.draw()
```

```
Preparing the Canvas
Drawing a Circle
```

The abstract classes may also contain concrete methods that have the implementation of the method and can be used by all the concrete classes. To define a concrete method in an abstract class, we simply define a method with implementation and don't decorate it with the @abstractmethod decorator. If needed, we may also override this concrete method in the concrete class to provide any additional functionality as per user needs.

Let's have a look at the following code snippet to create a concrete method in the abstract class(Shape):

In [60]:
```
from abc import ABC, abstractmethod, abstractproperty

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    def print_greetings(self):
        print("Hello from Shape class")

    @abstractmethod
    def draw(self):
        pass
```

In [63]:
```
class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw(self):
        super().draw()
        print("Drawing a Circle")
```

In the above code, we have defined a concrete method in the Shape class called print_greetings() that will just print a simple message. We can now invoke this method using the object of our concrete class(Circle):

In [64]:
```
#create a circle object
circle = Circle()
circle.print_greetings()
```

```
Hello from Shape class
```