Inheritance and Abstract Base Classes

What is inheritance?

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

The syntax for inheritance is as follows:

```
class BaseClass:
    Body of base class
class DerivedClass(BaseClass):
    Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Here is an example:

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:

   def __init__(self, no_of_sides):
       self.n = no_of_sides
       self.sides = [0 for i in range(no_of_sides)]

    def input_sides(self):
       for i in range(self.n):
            side = float(input("Enter side {}: ".format(i+1)))
            self.sides[i] = side

    def disp_sides(self):
        for i in range(self.n):
            print("Side", i+1, "is", self.sides[i])
```

This class has data attributes to store the number of sides, n and magnitude of each side as a list, sides.

Method `input_sides()` takes in magnitude of each side and similarly, `disp_sides()` will display these properly.

A triangle is a polygon with 3 sides. So, we can create a class called `Triangle` which inherits from `Polygon`. This makes all the attributes available in class `Polygon` readily available in `Triangle`. We don't need to define them again (code re-usability). `Triangle` is defined as follows:

```
class Triangle(Polygon):
    def __init__(self):
        super().__init__(self,3)  # we initialize the parent class here

    def find_area(self):
        a, b, c = self.sides  # this is called unpacking a list
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `find_area()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()
>>> t.input_sides()
Enter side 1: 3
Enter side 2: 5
Enter side 3: 4

>>> t.disp_sides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.find_area()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like `input_sides()` or `disp_sides()` for class `Triangle`, we were able to use them. If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Multiple Inheritance

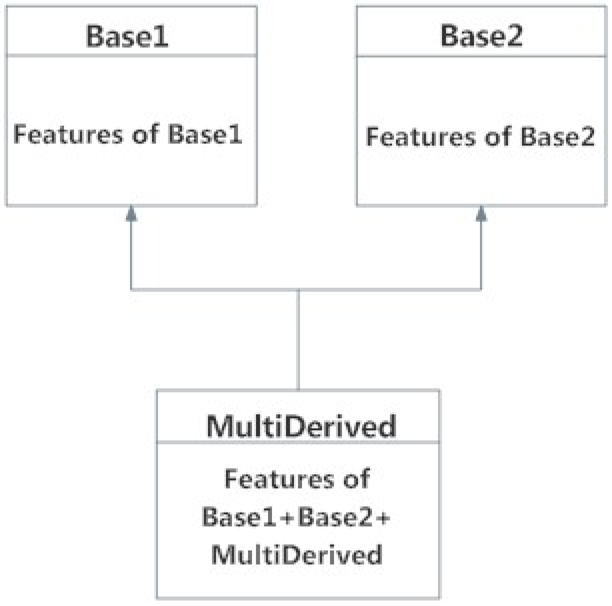
A class can be derived from more than one base classes in Python. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

The syntax for multiple inheritence looks like this:

```
class Base1:
    pass
class Base2:
    pass
class MultiDerived(Base1, Base2):
    pass
```

Here, `MultiDerived` is derived from classes `Base1` and `Base2`.



Multilevel Inheritance

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

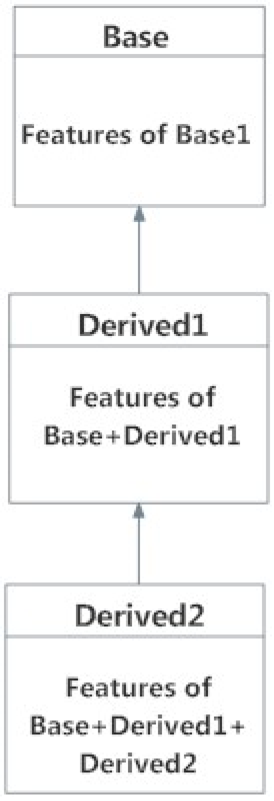
An example with corresponding visualization is given below.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    Pass
```

Here, `Derived1` is derived from `Base`, and `Derived2` is derived from `Derived1`.



Abstract Base Classes

Abstract classes are classes that contain one or more abstract methods (recall, method is just another name for a function). An abstract method is a method that is declared, but contains no implementation (similar to declaring the prototype of a function in C, before implementing it later). Abstract classes can not be instantiated (instantiation: creating an object from a class), and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are **not required** to implement abstract methods of the parent class.

```
class AbstractClass:
    def do_something(self):
        pass
class B(AbstractClass):
    pass
a = AbstractClass()
b = B()
```

If we run this program, we see that this is not an abstract class, because:

- we can instantiate an instance from `AbstractClass`
- we are not required to implement do_something in the class defintition of B

Our example implemented a case of simple inheritance which has nothing to do with an abstract class. In fact, Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - for obvious reasons - `abc`.

The following Python code uses the `abc` module and defines an abstract base class:

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

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

We will define now a subclass using the previously defined abstract class. You will notice that we haven't implemented the `do_something` method, even though we are required to implement it, because this method is decorated as an abstract method with the decorator "`abstractmethod`". We get an exception that DoAdd42 can't be instantiated:

```
class DoAdd42(AbstractClassExample):
    pass
x = DoAdd42(4)
```

The previous Python code returned the following output:

```
TypeError: Can't instantiate abstract class DoAdd42 with abstract methods
do_something
```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```
class DoAdd42(AbstractClassExample):
    def do_something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):
    def do_something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)
print(x.do_something())
print(y.do_something())
```

Output of the above program:

```
52
420
```

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

You may think that abstract methods can't be implemented in the abstract base class. This impression is wrong: An abstract method can have an implementation in the abstract class! Even if they are implemented, designers of subclasses will be forced to override the implementation. Like in other cases of "normal" inheritance, the abstract method can be invoked with `super()` call mechanism. This makes it possible to provide some basic functionality in the abstract method, which can be enriched by the subclass implementation.

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
    def do_something(self):
        super().do_something()
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

Output:

```
Some implementation!
The enrichment from AnotherSubclass
```