Inheritance and Multiple Inheritance

Let's say there two classes, namely class A and class B. If you have to perform simple inheritance, it can be done as follows:

```
Class A:
    pass
Class B(A):
    pass
```

If class A is the parent class and class B is inheriting from it, then class A is passed inside class B as a parameter. This will allow class B to directly access the attributes and methods inside class A.

Multiple inheritance

You have learned about single inheritance so far, but Python also gives us the ability to perform multiple inheritance between classes.

Here is a simple example of how that can be done.

```
# Example 1
class A:
    a = 1

class B:
    b = 2

class C(A, B):
    pass

c = C()
print(c.a, c.b)
```

The output is:

```
1 2
```

First, two classes called A and B are created and then variables a and b respectively are initialized with values. A new class C is then defined and classes A and B are passed to it. This is how multiple inheritance is done in Python. The order of classes is important, but not in this specific example. I then instantiate an object 'c' of class C. The values of the a and b variables are printed over the object c of class C even though a and b are not present inside class C.

The code above is an example of multiple inheritance. There are also other types of inheritance that fall under the umbrella of multiple inheritance. Let's examine an example.

Multi-level inheritance

```
class A:
    a = 1

class B(A):
    a = 2

class C(B):
    pass

c = C()
print(c.a)
```

The output is 2 because C derives from the immediate super class of C, and that's B.

The case above is an example of multi-level inheritance where the derived class C inherits from base class B. The class B is in turn a derived class of base class C. Class B here is an intermediary derived class. There are three levels of inheritance in this case, but it could be extended as long as I want, though it may become impractical after a while.

Built-in functions

There are two built-in functions that can come in handy when trying to find the relationship between different classes and objects: **issubclass()** and **isinstance()**.
The first one, **issubclass ()** is demonstrated below.

```
issubclass(class A, class B)
```

Two classes are passed as arguments to this function and a Boolean result is returned. The above example can be extended as follows.

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

The output is:

```
False
True
```

This illustrates how the child class is passed as the first argument. To avoid confusion, this can be read as: “Is B subclass of A?“ You can see the result is "True" in the second case where child B is the subclass.
Another built-in function similar to this one is **isinstance()** thatdetermines if some object is an instance of some class. So if I write:

```
Class A:
    pass
Class B(A):
    pass

b = B()
print(isinstance(b,B))
print(isinstance(b,B))
```

The output that I will get is "True".

Now that you know how classes can be extended from other classes, let's look at another useful built-in function called the **super() function**.
The super() function is a built-in function that can be called inside the derived class and gives access to the methods and variables of the parent classes or sibling classes. Sibling classes are the classes that share the same parent class. When you call the super() function, you get an object that represents the parent class in return.
The super() function plays an important role in multiple inheritance and helps drive the flow of the code execution. It helps in managing or determining the control of from where I can draw the values of my desired functions and variables.
If you change anything inside the parent class, there is a direct retrieval of changes inside the derived class. This is mainly used in places where you need to initialize the functionalities present inside the parent class in the child class as well. You can then add additional code in the child class.
Here is an example.

```
class Fruit():
    def __init__(self, fruit):
        print('Fruit type: ', fruit)


class FruitFlavour(Fruit):
    def __init__(self):
        super().__init__('Apple')
        print('Apple is sweet')

```

```
11    apple = FruitFlavour()
```

The output is:

```
1    Fruit type:  Apple
2    Apple is sweet
```

In the code above, if I had commented out the line for super() function, the output is:

```
1    Apple is sweet
```

This happened because when you initialize the child class, you don’t initialize the base class with it. super() function helps you to achieve this and add the initialization of base class with the derived class.