**Inheritance and Private Variables**

**Inheritance:**

Inheritance is a mechanism in OOP that allows a new class (derived or child class) to inherit attributes and behaviours from an existing class (base or parent class). The derived class can reuse and extend the functionality of the base class.

**Private Variables:**

Private variables are variables that are intended to be accessed only within the class that defines them. In many programming languages, including Python, a convention is used to indicate that a variable is private by prefixing its name with an underscore.

The syntax for a derived class definition looks like this:

class DerivedClassName(BaseClassName):

    <statement-1>

.

.

.

    <statement-N>

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

class DerivedClassName(modname.BaseClassName):

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There’s nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments). This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as BaseClassName in the global scope.)

Python has two built-in functions that work with inheritance:

Use isinstance() to check an instance’s type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is int or some class derived from int.

Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float, int) is False since float is not a subclass of int.

**Multiple inheritance**

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

class DerivedClassName(Base1, Base2, Base3):

    <statement-1>

.

.

.

    <statement-N>

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to super(). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. 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, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance.

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. \_spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.