1. What is the concept of an abstract superclass?

An abstract superclass is one way to provide re-usable code.

* You can extend the abstract class and inherit the code. This is sometimes more convenient than using static methods or object composition to share code.
* The abstract class can "fix" parts of the code (by making it final). This is called the "template method" pattern (and this is not possible with an interface, which cannot provide final methods).

Of course, you can achieve both with a non-abstract superclass as well.

An abstract class has the additional benefit that it does not have to provide a complete implementation (that would make sense to instantiate on its own), some parts can be left specified, but unimplemented (the abstract methods).

1. What happens when a class statement top level contains a basic assignment statement?

1. Why does a class need to manually call a superclass \_\_init\_\_ method?

The main reason for always calling base class \_init\_\_ is that **base class may typically create member variable and initialize them to defaults**. So if you don't call base class init, none of that code would be executed and you would end up with base class that has no member variables. Run this code.

The main reason for always calling base class \_init\_\_ is that base class may typically create member variable and initialize them to defaults. So if you don't call base class init, none of that code would be executed and you would end up with base class that has no member variables.

**Example**:

class Base:

def \_\_init\_\_(self):

print('base init')

class Derived1(Base):

def \_\_init\_\_(self):

print('derived1 init')

class Derived2(Base):

def \_\_init\_\_(self):

super(Derived2, self).\_\_init\_\_()

print('derived2 init')

print('Creating Derived1...')

d1 = Derived1()

1. How can you augment, instead of completely replacing, an inherited method?

A more sophisticated way to augment an inherited method involves forwarding. **Message forwarding** allows you to augment an inherited method in such a way that it can perform its inherited action and some new action.

**Augmenting Inherited Methods**

Each method that is inherited from a superclass can be augmented to perform some different action in the new class. If there is no augmentation then the inherited methods will perform as defined in the superclass.

An inherited method is augmented simply by creating a new definition for the method in your class definition. The following example demonstrates augmenting an inherited method:

Class cMessageButton1 is a Button

    Procedure DoShowSomething

        Send Info\_Box "First Message"

    End\_Procedure

    Procedure OnClick

        Send DoShowSomething

    End\_Procedure

End\_Class

Class cMessageButton2  is a cMessageButton1

    Procedure DoShowSomething

        Send Info\_Box "Second Message"

    End\_Procedure

End\_Class

In this example, the first class cMessageButton1 defines a new method DoShowSomething that displays the message "First Message". Thus object of this class, when clicked, would show the message "First Message". The second class cMessageButton2 inherits DoShowSomething and OnClick from cMessageButton1, but it augments DoShowSomething to display a different message. An object of this class, when clicked would show the message "Second Message."

**Message Forwarding**

A more sophisticated way to augment an inherited method involves *forwarding*. Message forwarding allows you to augment an inherited method in such a way that it can perform its inherited action *and* some new action.

All three method types can be forwarded (Procedure, Procedure Set and Function). Messages are forwarded by executing a *Forward* statement. The syntax for forwarding a message is different for each type of method as outlined below:

**Forwarding Procedure Methods**

Forward Send {method-name}  {Param1 … ParamN}

**Forwarding Procedure Set Methods**

Forward Set {method-name} {Param1 … ParamN}  To {value1 … valueN}

**Forwarding Function Methods**

Forward Get {method-name} {Param1 … ParamN} To {receiving-variable}

You can see that the syntax for forwarding a message is the same as the syntax for executing a method except that the Forward command is appended to the front of each statement. There is also no object reference when forwarding a method. You are *always* forwarding the message to the superclass of the *current object*.

An example of method augmentation with message forwarding follows:

Class cMessageButton  is a Button

    Procedure Construct\_Object

        Forward Send Construct\_Object

        Property String psMessage Public "First Message"

    End\_Procedure

    Procedure OnClick

        String sMessage

        Get psMessage To sMessage

        Send Info\_Box sMessage

    End\_Procedure

End\_Class

Class cShowLabelButton  is a cMessageButton

    Procedure Set Label  String sLabel

        Forward Set Label To sLabel

        Set psMessage     To sLabel

    End\_Procedure

End\_Class

In this example, the first class augments the Construct\_Object method cMessageButton to define a new string property. Note that Construct\_Object is forwarded to ensure that all the superclass's attributes are inherited. The OnClick event method is augmented to show the value of the new string property in a message box. The second class cShowLabelButton inherits the behavior of cMessageButton, but it also augments the inherited Procedure Set Label method so that the message that is shown when the button is clicked is the same as the button's label. The Forward Set Label statement is important because, without it, the Set Label method would no longer be able to set the button's label.

Fortunately, Python supports a feature called <i>inheritance.</i> By using inheritance, **you can obtain the features you want from a parent class when creating a child class**.\r\n\r\nOverriding the features that you don't need and adding new features lets you create new classes relatively fast and with a lot less effort.

5. How is the local scope of a class different from that of a function?

**Local scope is the area between an { and it's closing }.** **Function scope is the area between the opening { of a function and its closing }, which may contain more "local" scopes**. A label is visible in the entirety of the function within which it is defined, e.g. int c is not visible outside its enclosing block.

Local Scope **occurs when you create a variable inside a function**. By doing that, the visibility and accessibility of the variable is only allowed within that function. Any variable created inside the yellow box is a local variable, just like any variable inside blue box is a global one.

Scope” is just a technical term for the parts of your code that have access to a variable. In the picture below, the scope of the local variable is highlighted blue – it's **the function where that var was defined**. Any code inside that function can access (read and change) this variable.

Class scope **defines the accessibility or visibility of class variables or functions**. The term scope is defined as a place where in variable name, function name and typedef are used inside a program.