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).

Abstract Class The "abstract" keyword can be applied to a class e.g. public abstract class Account { ... A class that has one or more abstract methods is abstract -- it cannot be instantiated. "New" may not be used to create instances of the Abstract class. A class is not abstract if all of the abstract methods of its superclasses have definitions. Abstract Super Class A common superclass for several subclasses. Factor up common behavior Define the methods they all respond to. Methods that subclasses should implement are declared abstract Instances of the subclasses are created, but no instances of the superclass

**2. What happens when a class statement's top level contains a basic assignment statement?**

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

## **assignment\_stmt** ::= ([target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list) "=")+ ([starred\_expression](https://docs.python.org/3/reference/expressions.html" \l "grammar-token-python-grammar-starred_expression) | [yield\_expression](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-yield_expression))

## **target\_list** ::= [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target) ("," [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target))\* [","]

## **target** ::= [identifier](https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-python-grammar-identifier)

## | "(" [[target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list)] ")"

## | "[" [[target\_list](https://docs.python.org/3/reference/simple_stmts.html" \l "grammar-token-python-grammar-target_list)] "]"

## | [attributeref](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-attributeref)

## | [subscription](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-subscription)

## | [slicing](https://docs.python.org/3/reference/expressions.html#grammar-token-python-grammar-slicing)

## | "\*" [target](https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-target)

Assignment of an object to a single target is recursively defined as follows.

* If the target is an identifier (name):
  + If the name does not occur in a [global](https://docs.python.org/3/reference/simple_stmts.html#global) or [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) statement in the current code block: the name is bound to the object in the current local namespace.
  + Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal), respectively.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

* If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily [AttributeError](https://docs.python.org/3/library/exceptions.html" \l "AttributeError" \o "AttributeError)).
* **3. Why does a class need to manually call a superclass's \_\_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.

Yes, you should always call base class \_\_init\_\_ explicitly as a good coding practice. Forgetting to do this can cause subtle issues or run time errors. This is true even if \_\_init\_\_ doesn't take any parameters. This is unlike other languages where compiler would implicitly call base class constructor for you. Python doesn't do that!

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()

print('Creating Derived2...')

d2 = Derived2()

**This prints..**

Creating Derived1...

derived1 init

Creating Derived2...

base init

derived2 init

**4. 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.

### 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

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

Declaring a variable in a class (outside of a function): all class functions can access it (basically a public variable)

Declaring a variable inside a function inside a class: only that function can access it (it's in that function's scope)

Declaring a variable with self.(variable name) inside a function inside a class: all class functions can access it (how is this different from global (variable name)?)

And since there is no private/protected, everything is public, so everything accessible from inside a class is accessible from outside the class.

**def** scope\_test():

**def** do\_local():

spam = "local spam"

**def** do\_nonlocal():

**nonlocal** spam

spam = "nonlocal spam"

## **def** do\_global():

## **global** spam

## spam = "global spam"

## spam = "test spam"

## do\_local()

## print("After local assignment:", spam)

## do\_nonlocal()

## print("After nonlocal assignment:", spam)

## do\_global()

## print("After global assignment:", spam)

## scope\_test()

## print("In global scope:", spam)

**The output of the example code is:**

## After local assignment: test spam

## After nonlocal assignment: nonlocal spam

## After global assignment: nonlocal spam

## In global scope: global spam