Mixins All Versions

Examples

Mixin

A **Mixin** is a set of properties and methods that can be used in different classes, which *don't* come from a base class. In Object Oriented Programming languages, you typically use *inheritance* to give objects of different classes the same functionality; if a set of objects have some ability, you put that ability in a base class that both objects *inherit* from.

For instance, say you have the classes `Car` , `Boat` , and `Plane` . Objects from all of these classes have the ability to travel, so they get the function `travel` . In this scenario, they all travel the same basic way, too; by getting a route, and moving along it. To implement this function, you could derive all of the classes from `Vehicle` , and put the function in that shared class:

```
class Vehicle(object):
   """A generic vehicle class."""

   def __init__(self, position):
       self.position = position

   def travel(self, destination):
       route = calculate_route(from=self.position, to=destination)
       self.move_along(route)

class Car(Vehicle):
   ...

class Boat(Vehicle):
   ...

class Plane(Vehicle):
   ...
```

With this code, you can call `travel` on a car (`car.travel("Montana")`), boat (`boat.travel("Hawaii")`), and plane (`plane.travel("France")`)

However, what if you have functionality that's not available to a base class? Say, for instance, you want to give `Car` a radio and the ability to use it to play a song on a radio station, with `play_song_on_station` , but you also have a `Clock` that can use a radio too. `Car` and `Clock` could share a base class (`Machine`). However, not all machines can play songs; `Boat` and `Plane` can't (at least in this example). So how do you accomplish without duplicating code? You can use a mixin. In Python, giving a class a mixin is as simple as adding it to the list of subclasses, like this

```
class Foo(main_super, mixin): ...
```

`Foo` will inherit all of the properties and methods of `main_super` , but also those of `mixin` as well.

So, to give the classes `Car` and clock the ability to use a radio, you could override `Car` from the last example and write this:

```
class RadioUserMixin(object):
   def __init__(self):
       self.radio = Radio()

   def play_song_on_station(self, station):
       self.radio.set_station(station)
       self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
   ...

class Clock(Vehicle, RadioUserMixin):
   ...
```

Now you can call `car.play_song_on_station(98.7)` and `clock.play_song_on_station(101.3)` , but not something like `boat.play_song_on_station(100.5)`

The important thing with mixins is that they allow you to add functionality to much different objects, that don't share a "main" subclass with this functionality but still share the code for it nonetheless. Without mixins, doing something like the above example would be much harder, and/or might require some repetition.

Overriding Methods in Mixins

Mixins are a sort of class that is used to "mix in" extra properties and methods into a class. This is usually fine because many times the mixin classes don't override each other's, or the base class' methods. But if you do override methods or properties in your mixins this can lead to unexpected results because in Python the class hierarchy is defined right to left.

For instance, take the following classes

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

In this case the Mixin2 class is the base class, extended by Mixin1 and finally by BaseClass. Thus, if we execute the following code snippet:

```
>>> x = MyClass()
>>> x.test()
Base
```

We see the result returned is from the Base class. This can lead to unexpected errors in the logic of your code and needs to be accounted for and kept in mind

Syntax

```
class ClassName ( MainClass , Mixin1 , Mixin2 , ...): # Used to declare a class with the name ClassName , main (first) class MainClass , and mixins Mixin1 , Mixin2 , etc.
```

```
class ClassName ( Mixin1 , MainClass , Mixin2 , ...): # The 'main' class doesn't have to be the first class; there's really no difference between it and the mixin
```

Parameters

Remarks

Adding a mixin to a class looks a lot like adding a superclass, because it pretty much is just that. An object of a class with the mixin *Foo* will also be an instance of *Foo* , and isinstance(instance, Foo) will return true