Plugin and Extension Classes

All Versions

Examples

Mixins

In Object oriented programming language, a mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language.

It provides a mechanism for multiple inheritance by allowing multiple classes to use the common functionality, but without the complex semantics of multiple inheritance. Mixins are useful when a programmer wants to share functionality between different classes. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then inherited into each class that requires it.

When we use more than one mixins, Order of mixins are important. here is a simple example:

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

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

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

In this example we call `MyClass` and `test` method,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Result must be Mixin1 because Order is left to right. This could be show unexpected results when super classes add with it. So reverse order is more good just like this:

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

Result will be:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Mixins can be used to define custom plugins.

Python 3.x ≥ 3.0

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

 PluginSystemB().test()
# Base.
# Plugin B.
```

Plugins with Customized Classes

In Python 3.6, PEP 487 added the `__init_subclass__` special method, which simplifies and extends class customization without using metaclasses . Consequently, this feature allows for creating simple plugins . Here we demonstrate this feature by modifying a prior example :

Python 3.x ≥ 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
        super().test()
        print("Plugin A.")


class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

Results:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Syntax

Parameters

Remarks