abc – Abstract Base Classes - Python Module of the Week

Purpose:	Define and use abstract base classes for API checks in your code.
Available In:	2.6

Why use Abstract Base Classes?

Abstract base classes are a form of interface checking more strict than individual `hasattr()` checks for particular methods. By defining an abstract base class, you can define a common API for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins to an application, but can also aid you when working on a large team or with a large code-base where keeping all classes in your head at the same time is difficult or not possible.

How ABCs Work

`abc` works by marking methods of the base class as abstract, and then registering concrete classes as implementations of the abstract base. If your code requires a particular API, you can use `issubclass()` or `isinstance()` to check an object against the abstract class.

Let's start by defining an abstract base class to represent the API of a set of plugins for saving and loading data.

```
import abc

class PluginBase(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source and return an object."""
        return

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
        return
```

Registering a Concrete Class

There are two ways to indicate that a concrete class implements an abstract: register the class with the abc or subclass directly from the abc.

```
import abc
from abc_base import PluginBase

class RegisteredImplementation(object):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

PluginBase.register(RegisteredImplementation)

if __name__ == '__main__':
    print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

In this example the `PluginImplementation` is not derived from `PluginBase`, but is registered as implementing the `PluginBase` API.

```
$ python abc_register.py

Subclass: True
Instance: True
```

Implementation Through Subclassing

By subclassing directly from the base, we can avoid the need to register the class explicitly.

```
import abc
from abc_base import PluginBase

class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)

if __name__ == '__main__':
    print 'Subclass:', issubclass(SubclassImplementation, PluginBase)
    print 'Instance:', isinstance(SubclassImplementation(), PluginBase)
```

In this case the normal Python class management is used to recognize `PluginImplementation` as implementing the abstract `PluginBase`.

```
$ python abc_subclass.py

Subclass: True
Instance: True
```

A side-effect of using direct subclassing is it is possible to find all of the implementations of your plugin by asking the base class for the list of known classes derived from it (this is not an abc feature, all classes can do this).

```
import abc
from abc_base import PluginBase
import abc_subclass
import abc_register

for sc in PluginBase.__subclasses__():
    print sc.__name__
```

Notice that even though `abc_register` is imported, `RegisteredImplementation` is not among the list of subclasses because it is not actually derived from the base.

```
$ python abc_find_subclasses.py

SubclassImplementation
```

Dr. André Roberge has described using this capability to discover plugins by importing all of the modules in a directory dynamically and then looking at the subclass list to find the implementation classes.

Incomplete Implementations

Another benefit of subclassing directly from your abstract base class is that the subclass cannot be instantiated unless it fully implements the abstract portion of the API. This can keep half-baked implementations from triggering unexpected errors at runtime.

```
import abc
from abc_base import PluginBase

class IncompleteImplementation(PluginBase):

    def save(self, output, data):
        return output.write(data)
```

```
PluginBase.register(IncompleteImplementation)

if __name__ == '__main__':
    print 'Subclass:', issubclass(IncompleteImplementation, PluginBase)
    print 'Instance:', isinstance(IncompleteImplementation(), PluginBase)
```

```
$ python abc_incomplete.py

Subclass: True
Instance:
Traceback (most recent call last):
  File "abc_incomplete.py", line 22, in <module>
    print 'Instance:', isinstance(IncompleteImplementation(), PluginBase)
TypeError: Can't instantiate abstract class IncompleteImplementation with abstract
methods load
```

Concrete Methods in ABCs

Although a concrete class must provide an implementation of an abstract methods, the abstract base class can also provide an implementation that can be invoked via `super()`. This lets you re-use common logic by placing it in the base class, but force subclasses to provide an overriding method with (potentially) custom logic.

```
import abc
from cStringIO import StringIO

class ABCWithConcreteImplementation(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def retrieve_values(self, input):
        print 'base class reading data'
        return input.read()

class ConcreteOverride(ABCWithConcreteImplementation):

    def retrieve_values(self, input):
        base_data = super(ConcreteOverride, self).retrieve_values(input)
        print 'subclass sorting data'
        response = sorted(base_data.splitlines())
        return response

input = StringIO("""line one
```

```
line two
line three
""")

reader = ConcreteOverride()
print reader.retrieve_values(input)
print
```

Since `ABCWithConcreteImplementation` is an abstract base class, it isn't possible to instantiate it to use it directly. Subclasses *must* provide an override for `retrieve_values()`, and in this case the concrete class massages the data before returning it at all.

```
$ python abc_concrete_method.py

base class reading data
subclass sorting data
['line one', 'line three', 'line two']
```

Abstract Properties

If your API specification includes attributes in addition to methods, you can require the attributes in concrete classes by defining them with `@abstractproperty`.

```
import abc

class Base(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractproperty
    def value(self):
        return 'Should never get here'


class Implementation(Base):

    @property
    def value(self):
        return 'concrete property'


try:
    b = Base()
    print 'Base.value:', b.value
```

```
except Exception, err:
    print 'ERROR:', str(err)

i = Implementation()
print 'Implementation.value:', i.value
```

The `Base` class in the example cannot be instantiated because it has only an abstract version of the property getter method.

```
$ python abc_abstractproperty.py

ERROR: Can't instantiate abstract class Base with abstract methods value
Implementation.value: concrete property
```

You can also define abstract read/write properties.

```
import abc

class Base(object):
    __metaclass__ = abc.ABCMeta

    def value_getter(self):
        return 'Should never see this'

    def value_setter(self, newvalue):
        return

    value = abc.abstractproperty(value_getter, value_setter)


class PartialImplementation(Base):

    @abc.abstractproperty
    def value(self):
        return 'Read-only'


class Implementation(Base):

    _value = 'Default value'

    def value_getter(self):
        return self._value
```

```
    def value_setter(self, newvalue):
        self._value = newvalue

    value = property(value_getter, value_setter)


try:
    b = Base()
    print 'Base.value:', b.value
except Exception, err:
    print 'ERROR:', str(err)

try:
    p = PartialImplementation()
    print 'PartialImplementation.value:', p.value
except Exception, err:
    print 'ERROR:', str(err)

i = Implementation()
print 'Implementation.value:', i.value

i.value = 'New value'
print 'Changed value:', i.value
```

Notice that the concrete property must be defined the same way as the abstract property. Trying to override a read/write property in `PartialImplementation` with one that is read-only does not work.

```
$ python abc_abstractproperty_rw.py

ERROR: Can't instantiate abstract class Base with abstract methods value
ERROR: Can't instantiate abstract class PartialImplementation with abstract methods value
Implementation.value: Default value
Changed value: New value
```

To use the decorator syntax does with read/write abstract properties, the methods to get and set the value should be named the same.

```
import abc

class Base(object):
    __metaclass__ = abc.ABCMeta
```

```
    @abc.abstractproperty
    def value(self):
        return 'Should never see this'

    @value.setter
    def value(self, newvalue):
        return


class Implementation(Base):

    _value = 'Default value'

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, newvalue):
        self._value = newvalue


i = Implementation()
print 'Implementation.value:', i.value

i.value = 'New value'
print 'Changed value:', i.value
```

Notice that both methods in the `Base` and `Implementation` classes are named `value()`, although they have different signatures.

```
$ python abc_abstractproperty_rw_deco.py

Implementation.value: Default value
Changed value: New value
```

Collection Types

The `collections` module defines several abstract base classes related to container (and containable) types.

General container classes:

Iterator and Sequence classes:

Unique values:

Mappings:

Miscelaneous:

- Callable

In addition to serving as detailed real-world examples of abstract base classes, Python's built-in types are automatically registered to these classes when you import `collections`. This means you can safely use `isinstance()` to check parameters in your code to ensure that they support the API you need. The base classes can also be used to define your own collection types, since many of them provide concrete implementations of the internals and only need a few methods overridden. Refer to the standard library docs for collections for more details.

See also