abc — Abstract Base Classes

Purpose: Define and use abstract base classes for interface verification.

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, a common API can be established for a set of subclasses. This capability is especially useful in situations where someone less familiar with the source for an application is going to provide plug-in extensions, but can also help when working on a large team or with a large code-base where keeping track of all of the classes 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 an application or library requires a particular API, `issubclass()` or `isinstance()` can be used to check an object against the abstract class.

To start, define an abstract base class to represent the API of a set of plug-ins for saving and loading data. Set the metaclass for the new base class to `ABCMeta`, and use decorators to establish the public API for the class. The following examples use `abc_base.py`.

```
# abc_base.py

import abc


class PluginBase(metaclass=abc.ABCMeta):

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

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

Registering a Concrete Class

There are two ways to indicate that a concrete class implements an abstract API: either explicitly register the class or create a new subclass directly from the abstract base. Use the `register()` class method as a decorator on a concrete class to add it explicitly when the class provides the required API, but is not part of the inheritance tree of the abstract base class.

```
# abc_register.py

import abc
from abc_base import PluginBase


class LocalBaseClass:
    pass


@PluginBase.register
class RegisteredImplementation(LocalBaseClass):

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

    def save(self, output, data):
```

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


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

In this example the `RegisteredImplementation` is derived from `LocalBaseClass`, but is registered as implementing the `PluginBase` API so `issubclass()` and `isinstance()` treat it as though it is derived from `PluginBase`.

```
$ python3 abc_register.py

Subclass: True
Instance: True
```

Implementation Through Subclassing

Subclassing directly from the base avoids the need to register the class explicitly.

```
# abc_subclass.py

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, normal Python class management features are used to recognize `SubclassImplementation` as implementing the abstract `PluginBase`.

```
$ python3 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 a plug-in 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).

```
# abc_find_subclasses.py

import abc
from abc_base import PluginBase
import abc_subclass
import abc_register

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

Even though abc_`register()` is imported, `RegisteredImplementation` is not among the list of subclasses because it is not actually derived from the base.

```
$ python3 abc_find_subclasses.py
```

```
SubclassImplementation
```

Helper Base Class

Forgetting to set the metaclass properly means the concrete implementations do not have their APIs enforced. To make it easier to set up the abstract class properly, a base class is provided that sets the metaclass automatically.

```
# abc_abc_base.py

import abc


class PluginBase(abc.ABC):

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

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


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))
```

To create a new abstract class, simply inherit from ABC.

```
$ python3 abc_abc_base.py

Subclass: True
Instance: True
```

Incomplete Implementations

Another benefit of subclassing directly from the abstract base class is that the subclass cannot be instantiated unless it fully implements the abstract portion of the API.

```
# abc_incomplete.py

import abc
from abc_base import PluginBase


@PluginBase.register
class IncompleteImplementation(PluginBase):

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


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

This keeps incomplete implementations from triggering unexpected errors at runtime.

```
$ python3 abc_incomplete.py

Subclass: True
Traceback (most recent call last):
  File "abc_incomplete.py", line 24, in <module>
    print('Instance:', isinstance(IncompleteImplementation(),
TypeError: Can't instantiate abstract class
IncompleteImplementation with abstract methods load
```

Concrete Methods in ABCs

Although a concrete class must provide implementations of all abstract methods, the abstract base class can also provide implementations that can be invoked via `super()`. This allows common logic to be reused by placing it in the base class, but forces subclasses to provide an overriding method with (potentially) custom logic.

```
# abc_concrete_method.py

import abc
import io


class ABCWithConcreteImplementation(abc.ABC):

    @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 = io.StringIO("""line one
line two
line three
""")

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

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

```
$ python3 abc_concrete_method.py

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

Abstract Properties

If an API specification includes attributes in addition to methods, it can require the attributes in concrete classes by combining `abstractmethod()` with `property()`.

```
# abc_abstractproperty.py

import abc


class Base(abc.ABC):

    @property
    @abc.abstractmethod
    def value(self):
        return 'Should never reach here'

    @property
    @abc.abstractmethod
    def constant(self):
        return 'Should never reach here'


class Implementation(Base):

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

    constant = 'set by a class attribute'


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

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

The Base class in the example cannot be instantiated because it has only an abstract version of the property getter methods for value and constant. The value property is given a concrete getter in `Implementation` and `constant` is defined using a class attribute.

```
$ python3 abc_abstractproperty.py

ERROR: Can't instantiate abstract class Base with abstract
methods constant, value
Implementation.value   : concrete property
Implementation.constant: set by a class attribute
```

Abstract read-write properties can also be defined.

```
# abc_abstractproperty_rw.py

import abc


class Base(abc.ABC):

    @property
    @abc.abstractmethod
    def value(self):
        return 'Should never reach here'

    @value.setter
    @abc.abstractmethod
    def value(self, new_value):
        return


class PartialImplementation(Base):
    @property
    def value(self):
        return 'Read-only'


class Implementation(Base):

    _value = 'Default value'

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

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


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

p = PartialImplementation()
print('PartialImplementation.value:', p.value)

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

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

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

The concrete property must be defined the same way as the abstract property, as either read-write or read-only. Overriding a read-write property in `PartialImplementation` with one that is read-only leaves the property read-only – the property's setter method from the base class is not reused.

```
$ python3 abc_abstractproperty_rw.py

ERROR: Can't instantiate abstract class Base with abstract
methods value
PartialImplementation.value: Read-only
ERROR: can't set attribute
Implementation.value: Default value
Changed value: New value
```

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

Abstract Class and Static Methods

Class and static methods can also be marked as abstract.

```
# abc_class_static.py

import abc


class Base(abc.ABC):

    @classmethod
    @abc.abstractmethod
    def factory(cls, *args):
        return cls()

    @staticmethod
    @abc.abstractmethod
    def const_behavior():
        return 'Should never reach here'


class Implementation(Base):

    def do_something(self):
        pass

    @classmethod
    def factory(cls, *args):
        obj = cls(*args)
        obj.do_something()
        return obj

    @staticmethod
    def const_behavior():
        return 'Static behavior differs'


try:
    o = Base.factory()
    print('Base.value:', o.const_behavior())
except Exception as err:
    print('ERROR:', str(err))

i = Implementation.factory()
print('Implementation.const_behavior :', i.const_behavior())
```

Although the class method is invoked on the class rather than an instance, it still prevents the class from being instantiated if it is not defined.

```
$ python3 abc_class_static.py

ERROR: Can't instantiate abstract class Base with abstract
methods const_behavior, factory
Implementation.const_behavior : Static behavior differs
```

See also

- Standard library documentation for abc
- **PEP 3119** – Introducing Abstract Base Classes
- collections – The collections module includes abstract base classes for several collection types.
- **PEP 3141** – A Type Hierarchy for Numbers
- Wikipedia: Strategy Pattern – Description and examples of the strategy pattern, a common plug-in implementation pattern.
- Dynamic Code Patterns: Extending Your Applications With Plugins – PyCon 2013 presentation by Doug Hellmann
- Python 2 to 3 porting notes for abc

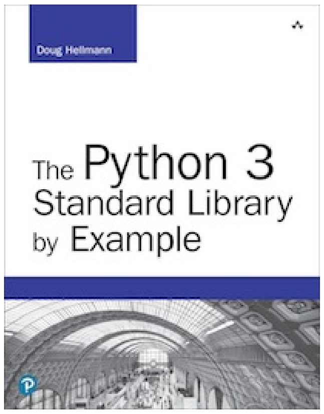
warnings — Non-fatal Alerts

dis — Python Bytecode Disassembler

This page was last updated 2016-12-31.

Navigation

warnings — Non-fatal Alerts
dis — Python Bytecode Disassembler



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?