Abstract Base Classes (abc) Python 2.x 2.6–2.7 , Python 3.x 3.0–3.6

Examples

Setting the ABCMeta metaclass

Abstract classes are classes that are meant to be inherited but avoid implementing specific methods, leaving behind only method signatures that subclasses must implement.

Abstract classes are useful for defining and enforcing class abstractions at a high level, similar to the concept of interfaces in typed languages, without the need for method implementation.

One conceptual approach to defining an abstract class is to stub out the class methods, and then raise a NotImplementedError if accessed. This prevents children classes from accessing parent methods without overriding them first. Like so:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")


class Apple(Fruit):
    pass


a = Apple()
a.check_ripeness() # raises NotImplementedError
```

Creating an abstract class in this way prevents improper usage of methods that are not overriden, and certainly encourages methods to be defined in child classes, but it does not enforce their definition. With the `abc` module we can prevent child classes from being instantiated when they fail to override abstract class methods of their parents and ancestors:

```
from abc import ABCMeta

class AbstractClass(object):
    # the metaclass attribute must always be set as a class variable
    __metaclass__ = ABCMeta

   # the abstractmethod decorator registers this method as undefined
   @abstractmethod
   def virtual_method_subclasses_must_define(self):
       # Can be left completely blank, or a base implementation can be provided
       # Note that ordinarily a blank interpretation implicitly returns `None`,
       # but by registering, this behaviour is no longer enforced.
```

It is now possible to simply subclass and override:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

Why/How to use ABCMeta and @abstractmethod

Abstract base classes (ABCs) enforce what derived classes implement particular methods from the base class.

To understand how this works and why we should use it, let's take a look at an example that Van Rossum would enjoy. Let's say we have a Base class "MontyPython" with two methods (joke & punchline) that must be implemented by all derived classes.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

When we instantiate an object and call it's two methods, we'll get an error (as expected) with the `punchline()` method.

```
 >>> sketch = ArgumentClinic()
 >>> sketch = ArgumentClinic()
 >>> sketch.punchline()
NotImplementedError
```

However, this still allows us to instantiate an object of the ArgumentClinic class without getting an error. In fact we don't get an error until we look for the punchline().

This is avoided by using the Abstract Base Class (ABC) module. Let's see how this works with the same example:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

@abstractmethod
def punchline(self):
    pass

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

This time when we try to instantiate an object from the incomplete class, we immediately get a TypeError!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

In this case, it's easy to complete the class to avoid any TypeErrors:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

This time when you instantiate an object it works!

Syntax

Parameters

Remarks