28.7. abc — Abstract Base Classes¶

New in version 2.6.

This module provides the infrastructure for defining an *abstract base class* (ABCs) in Python, as outlined in **PEP 3119**; see the PEP for why this was added to Python. (See also **PEP 3141** and the `numbers` module regarding a type hierarchy for numbers based on ABCs.)

The `collections` module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition the `collections` module has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, is it hashable or a mapping.

This module provides the following class:

class `abc.ABCMeta`¶

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as “virtual subclasses” – these and their descendants will be considered subclasses of the registering ABC by the built-in `issubclass()` function, but the registering ABC won’t show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via `super()`). [1]

Classes created with a metaclass of `ABCMeta` have the following method:

`register`(*subclass*)¶

Register *subclass* as a “virtual subclass” of this ABC. For example:

```
from abc import ABCMeta

class MyABC:
   __metaclass__ = ABCMeta

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

You can also override this method in an abstract base class:

`__subclasshook__`(*subclass*)¶

(Must be defined as a class method.)

Check whether *subclass* is considered a subclass of this ABC. This means that you can customize the behavior of `issubclass` further without the need to call `register()` on every class you want to consider a subclass of the ABC. (This class method is called from the `__subclasscheck__()` method of the ABC.)

This method should return `True`, `False` or `NotImplemented`. If it returns `True`, the *subclass* is considered a subclass of this ABC. If it returns `False`, the *subclass* is not considered a subclass of this ABC, even if it would normally be one. If it returns `NotImplemented`, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:

```
class Foo(object):
   def __getitem__(self, index):
       ...
   def __len__(self):
       ...
   def get_iterator(self):
       return iter(self)
```

```
class MyIterable:
   __metaclass__ = ABCMeta

   @abstractmethod
   def __iter__(self):
       while False:
           yield None

   def get_iterator(self):
       return self.__iter__()

   @classmethod
   def __subclasshook__(cls, C):
       if cls is MyIterable:
           if any("__iter__" in B.__dict__ for B in C.__mro__):
               return True
       return NotImplemented

MyIterable.register(Foo)
```

The ABC `MyIterable` defines the standard iterable method, `__iter__()`, as an abstract method. The implementation given here can still be called from subclasses. The `get_iterator()` method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The `__subclasshook__()` class method defined here says that any class that has an `__iter__()` method in its `__dict__` (or in that of one of its base classes, accessed via the `__mro__` list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`, even though it does not define an `__iter__()` method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make `get_iterator` available as a method of `Foo`, so it is provided separately.

It also provides the following decorators:

`abc.abstractmethod`(*function*)¶

A decorator indicating abstract methods.

Using this decorator requires that the class's metaclass is `ABCMeta` or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The `abstractmethod()` only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's `register()` method are not affected.

Usage:

```
class C:
   __metaclass__ = ABCMeta
   @abstractmethod
   def my_abstract_method(self, ...):
       ...
```

Note

Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the `super()` mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

`abc.abstractproperty`([*fget*[, *fset*[, *fdel*[, *doc*]]]])¶

A subclass of the built-in `property()`, indicating an abstract property.

Using this function requires that the class's metaclass is `ABCMeta` or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

```
class C:
   __metaclass__ = ABCMeta
   @abstractproperty
   def my_abstract_property(self):
       ...
```

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

```
class C:
   __metaclass__ = ABCMeta
   def getx(self): ...
   def setx(self, value): ...
   x = abstractproperty(getx, setx)
```

Footnotes

[1] C++ programmers should note that Python's virtual base class concept is not the same as C++'s.

Previous topic

28.6. `contextlib` — Utilities for `with`-statement contexts

Next topic

28.8. `atexit` — Exit handlers

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 28. Python Runtime Services »