29.8 abc — Abstract Base Classes

Source code: Lib/abc.py

This module provides the infrastructure for defining *abstract base classes* (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.abc` submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is *hashable* or if it is a mapping.

This module provides the metaclass `ABCMeta` for defining ABCs and a helper class `ABC` to alternatively define ABCs through inheritance:

class abc.ABC

A helper class that has `ABCMeta` as its metaclass. With this class, an abstract base class can be created by simply deriving from `ABC` avoiding sometimes confusing metaclass usage, for example:

```
from abc import ABC

class MyABC(ABC):
    pass
```

Note that the type of `ABC` is still `ABCMeta`, therefore inheriting from `ABC` requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also define an abstract base class by passing the metaclass keyword and using `ABCMeta` directly, for example:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

New in version 3.4.

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 ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

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

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

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

Changed in version 3.4: To detect calls to *register()*, you can use the *get_cache_token()* function.

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:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @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.

The *abc* module also provides the following decorator:

@abc.**abstractmethod**

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. *abstractmethod()* may be used to declare abstract methods for properties and descriptors.

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

When *abstractmethod()* is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using `__isabstractmethod__`. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python's built-in *property* does the equivalent of:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

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.

The *abc* module also supports the following legacy decorators:

`@abc.`**`abstractclassmethod`**

New in version 3.2.

Deprecated since version 3.3: It is now possible to use *classmethod* with *abstractmethod()*, making this decorator redundant.

A subclass of the built-in *classmethod()*, indicating an abstract classmethod. Otherwise it is similar to *abstractmethod()*.

This special case is deprecated, as the *classmethod()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

`@abc.`**`abstractstaticmethod`**

New in version 3.2.

Deprecated since version 3.3: It is now possible to use *staticmethod* with *abstractmethod()*, making this decorator redundant.

A subclass of the built-in *staticmethod()*, indicating an abstract staticmethod. Otherwise it is similar to *abstractmethod()*.

This special case is deprecated, as the *staticmethod()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

`@abc.`**`abstractproperty`**

Deprecated since version 3.3: It is now possible to use *property*, `property.getter()`, `property.setter()` and `property.deleter()` with *abstractmethod()*, making this decorator redundant.

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

This special case is deprecated, as the *property()* decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

The above example defines a read-only property; you can also define a read-write abstract property by appropriately marking one or more of the underlying methods as abstract:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def x(self, val):
        ...
```

If only some components are abstract, only those components need to be updated to create a concrete property in a subclass:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

The `abc` module also provides the following functions:

`abc.get_cache_token()`

Returns the current abstract base class cache token.

The token is an opaque object (that supports equality testing) identifying the current version of the abstract base class cache for virtual subclasses. The token changes with every call to `ABCMeta.register()` on any ABC.

New in version 3.4.

`abc.update_abstractmethods(cls)`

A function to recalculate an abstract class's abstraction status. This function should be called if a class's abstract methods have been implemented or changed after it was created. Usually, this function should be called from within a class decorator.

Returns *cls*, to allow usage as a class decorator.

If *cls* is not an instance of `ABCMeta`, does nothing.

Note: This function assumes that *cls*'s superclasses are already updated. It does not update any subclasses.

New in version 3.10.

29.9 `atexit` — Exit handlers

The `atexit` module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. `atexit` runs these functions in the *reverse* order in which they were registered; if you register `A`, `B`, and `C`, at interpreter termination time they will be run in the order `C`, `B`, `A`.

Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when `os._exit()` is called.

Note: The effect of registering or unregistering functions from within a cleanup function is undefined.

Changed in version 3.7: When used with C-API subinterpreters, registered functions are local to the interpreter they were registered in.

`atexit.register(func, *args, **kwargs)`

Register *func* as a function to be executed at termination. Any optional arguments that are to be passed to *func* must be passed as arguments to `register()`. It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if `sys.exit()` is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.