:mod:`abc` --- Abstract Base Classes

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 4)

Unknown directive type "module".

```
.. module:: abc
   :synopsis: Abstract base classes according to :pep:`3119`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 7)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Guido van Rossum
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 8)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Georg Brandl
```

Source code: :source:`Lib/abc.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 11); *backlink*

Unknown interpreted text role "source".

This module provides the infrastructure for defining :term:`abstract base classes <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 :mod:`numbers` module regarding a type hierarchy for numbers based on ABCs.)

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 15); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 15); *backlink*

Unknown interpreted text role "mod".

The :mod:`collections` module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the :mod:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 20); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 20); *backlink*

Unknown interpreted text role "mod".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 27); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 27); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 32); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 32); *backlink*

Unknown interpreted text role "class".

```
from abc import ABC

class MyABC(ABC):
    pass
```

Note that the type of :class:`ABC` is still :class:`ABCMeta`, therefore inheriting from :class:`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 :class:`ABCMeta` directly, for example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 41); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 41); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 41); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 41); *backlink*

Unknown interpreted text role "class".

```
from abc import ABCMeta

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 52)

Unknown directive type "versionadded".

```
.. versionadded:: 3.4
```

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 :func:`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 :func:`super`). [1]

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 59); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 59); *backlink*

Unknown interpreted text role "func".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 68); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 70)

Unknown directive type "method".

```
.. 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)

   .. versionchanged:: 3.3
      Returns the registered subclass, to allow usage as a class decorator.

   .. versionchanged:: 3.4
      To detect calls to :meth:`register`, you can use the
      :func:`get_cache_token` function.
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 94)

Unknown directive type "method".

```
.. method:: __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 :meth:`register` on every class you want to consider a
   subclass of the ABC.  (This class method is called from the
   :meth:`__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.

        .. XXX explain 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, :meth:`~iterator.__iter__`, as an abstract method. The implementation given here can still be called from subclasses. The :meth:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 143); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 143); *backlink*

Unknown interpreted text role "meth".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 149); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 149); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 149); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 149); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 154); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 154); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 154); *backlink*

Unknown interpreted text role "meth".

The :mod:`abc` module also provides the following decorator:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 163); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 165)

Unknown directive type "decorator".

```
.. decorator:: abstractmethod

   A decorator indicating abstract methods.

   Using this decorator requires that the class's metaclass is :class:`ABCMeta`
   or is derived from it.  A class that has a metaclass derived from
   :class:`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.  :func:`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 :func:`update_abstractmethods` function.  The
   :func:`abstractmethod` only affects subclasses derived using regular
   inheritance; "virtual subclasses" registered with the ABC's :meth:`register`
   method are not affected.

   When :func:`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
        :attr:`__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 :class:`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 :func:`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 :mod:`abc` module also supports the following legacy decorators:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 239); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 241)

Unknown directive type "decorator".

```
.. decorator:: abstractclassmethod

   .. versionadded:: 3.2
   .. deprecated:: 3.3
       It is now possible to use :class:`classmethod` with
       :func:`abstractmethod`, making this decorator redundant.

   A subclass of the built-in :func:`classmethod`, indicating an abstract
   classmethod. Otherwise it is similar to :func:`abstractmethod`.

   This special case is deprecated, as the :func:`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):
              ...
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 262)

Unknown directive type "decorator".

```
.. decorator:: abstractstaticmethod
```

```
   .. versionadded:: 3.2
   .. deprecated:: 3.3
       It is now possible to use :class:`staticmethod` with
       :func:`abstractmethod`, making this decorator redundant.

   A subclass of the built-in :func:`staticmethod`, indicating an abstract
   staticmethod. Otherwise it is similar to :func:`abstractmethod`.

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

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 283)

Unknown directive type "decorator".

```
.. decorator:: abstractproperty

   .. deprecated:: 3.3
       It is now possible to use :class:`property`, :meth:`property.getter`,
       :meth:`property.setter` and :meth:`property.deleter` with
       :func:`abstractmethod`, making this decorator redundant.

   A subclass of the built-in :func:`property`, indicating an abstract
   property.

   This special case is deprecated, as the :func:`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
          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 :mod:`abc` module also provides the following functions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 326); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

main\Doc\library\[cpython-main][Doc][library]abc.rst, line 328)

Unknown directive type "function".

```
.. function:: 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 :meth:`ABCMeta.register` on any ABC.

   .. versionadded:: 3.4
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]abc.rst, line 338)

Unknown directive type "function".

```
.. function:: 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 :class:`ABCMeta`, does nothing.

   .. note::

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

   .. versionadded:: 3.10
```

Footnotes

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