# PEP 3119 -- Introducing Abstract Base Classes

| | |
|---|---|
| **PEP:** | 3119 |
| **Title:** | Introducing Abstract Base Classes |
| **Author:** | Guido van Rossum <guido at python.org>, Talin <viridia at gmail.com> |
| **Status:** | Final |
| **Type:** | Standards Track |
| **Created:** | 18-Apr-2007 |
| **Post-History:** | 26-Apr-2007, 11-May-2007 |

Contents

## Abstract

This is a proposal to add Abstract Base Class (ABC) support to Python 3000. It proposes:

- A way to overload `isinstance()` and `issubclass()` .
- A new module `abc` which serves as an "ABC support framework". It defines a metaclass for use with ABCs and a decorator that can be used to define abstract methods.
- Specific ABCs for containers and iterators, to be added to the collections module.

Much of the thinking that went into the proposal is not about the specific mechanism of ABCs, as contrasted with Interfaces or Generic Functions (GFs), but about clarifying philosophical issues like "what makes a set", "what makes a mapping" and "what makes a sequence".

There's also a companion PEP 3141 , which defines ABCs for numeric types.

## Rationale

In the domain of object-oriented programming, the usage patterns for interacting with an object can be divided into two basic categories, which are 'invocation' and 'inspection'.

Invocation means interacting with an object by invoking its methods. Usually this is combined with polymorphism, so that invoking a given method may run different code depending on the type of an object.

Inspection means the ability for external code (outside of the object's methods) to examine the type or properties of that object, and make decisions on how to treat that object based on that information.

Both usage patterns serve the same general end, which is to be able to support the processing of diverse and potentially novel objects in a uniform way, but at the same time allowing processing decisions to be customized for each different type of object.

In classical OOP theory, invocation is the preferred usage pattern, and inspection is actively discouraged, being considered a relic of an earlier, procedural programming style. However, in practice this view is simply too dogmatic and inflexible, and leads to a kind of design rigidity that is very much at odds with the dynamic nature of a language like Python.

In particular, there is often a need to process objects in a way that wasn't anticipated by the creator of the object class. It is not always the best solution to build in to every object methods that satisfy the needs of every possible user of that object. Moreover, there are many powerful dispatch philosophies that are in direct contrast to the classic OOP requirement of behavior being strictly encapsulated within an object, examples being rule or pattern-match driven logic.

On the other hand, one of the criticisms of inspection by classic OOP theorists is the lack of formalisms and the ad hoc nature of what is being inspected. In a language such as Python, in which almost any aspect of an object can be reflected and directly accessed by external code, there are many different ways to test whether an object conforms to a particular protocol or not. For example, if asking 'is this object a mutable sequence container?', one can look for a base class of 'list', or one can look for a method named '__getitem__'. But note that although these tests may seem obvious, neither of them are correct, as one generates false negatives, and the other false positives.

The generally agreed-upon remedy is to standardize the tests, and group them into a formal arrangement. This is most easily done by associating with each class a set of standard testable properties, either via the inheritance mechanism or some other means. Each test carries with it a set of promises: it contains a promise about the general behavior of the class, and a promise as to what other class methods will be available.

This PEP proposes a particular strategy for organizing these tests known as Abstract Base Classes, or ABC. ABCs are simply Python classes that are added into an object's inheritance tree to signal certain features of that object to an external inspector. Tests are done using `isinstance()` , and the presence of a particular ABC means that the test has passed.

In addition, the ABCs define a minimal set of methods that establish the characteristic behavior of the type. Code that discriminates objects based on their ABC type can trust that those methods will always be present. Each of these methods are accompanied by an generalized abstract semantic definition that is described in the documentation for the ABC. These standard semantic definitions are not enforced, but are strongly recommended.

Like all other things in Python, these promises are in the nature of a gentlemen's agreement, which in this case means that while the language does enforce some of the promises made in the ABC, it is up to the implementer of the concrete class to insure that the remaining ones are kept.

- A way to overload `isinstance()` and `issubclass()` .

- A new module `abc` which serves as an "ABC support framework". It defines a metaclass for use with ABCs and a decorator that can be used to define abstract methods.
- Specific ABCs for containers and iterators, to be added to the collections module.

## Overloading `isinstance()` and `issubclass()`

During the development of this PEP and of its companion, PEP 3141, we repeatedly faced the choice between standardizing more, fine-grained ABCs or fewer, course-grained ones. For example, at one stage, PEP 3141 introduced the following stack of base classes used for complex numbers: MonoidUnderPlus, AdditiveGroup, Ring, Field, Complex (each derived from the previous). And the discussion mentioned several other algebraic categorizations that were left out: Algebraic, Transcendental, and IntegralDomain, and PrincipalIdealDomain. In earlier versions of the current PEP, we considered the use cases for separate classes like Set, ComposableSet, MutableSet, HashableSet, MutableComposableSet, HashableComposableSet.

The dilemma here is that we'd rather have fewer ABCs, but then what should a user do who needs a less refined ABC? Consider e.g. the plight of a mathematician who wants to define his own kind of Transcendental numbers, but also wants float and int to be considered Transcendental. PEP 3141 originally proposed to patch float.__bases__ for that purpose, but there are some good reasons to keep the built-in types immutable (for one, they are shared between all Python interpreters running in the same address space, as is used by mod_python [16] ).

Another example would be someone who wants to define a generic function ( PEP 3124 ) for any sequence that has an `append()` method. The `Sequence` ABC (see below) doesn't promise the `append()` method, while `MutableSequence` requires not only `append()` but also various other mutating methods.

To solve these and similar dilemmas, the next section will propose a metaclass for use with ABCs that will allow us to add an ABC as a "virtual base class" (not the same concept as in C++) to any class, including to another ABC. This allows the standard library to define ABCs `Sequence` and `MutableSequence` and register these as virtual base classes for built-in types like `basestring` , `tuple` and `list` , so that for example the following conditions are all true:

```
isinstance([], Sequence)
issubclass(list, Sequence)
issubclass(list, MutableSequence)
isinstance((), Sequence)
not issubclass(tuple, MutableSequence)
isinstance("", Sequence)
issubclass(bytearray, MutableSequence)
```

The primary mechanism proposed here is to allow overloading the built-in functions `isinstance()` and `issubclass()` . The overloading works as follows: The call `isinstance(x, C)` first checks whether `C.__instancecheck__` exists, and if so, calls `C.__instancecheck__(x)` instead of its normal implementation. Similarly, the call `issubclass(D, C)` first checks whether `C.__subclasscheck__` exists, and if so, calls

`C.__subclasscheck__(D)` instead of its normal implementation.

Note that the magic names are not `__isinstance__` and `__issubclass__` ; this is because the reversal of the arguments could cause confusion, especially for the `issubclass()` overloader.

A prototype implementation of this is given in [12] .

Here is an example with (naively simple) implementations of `__instancecheck__` and `__subclasscheck__` :

```
class ABCMeta(type):

    def __instancecheck__(cls, inst):
        """Implement isinstance(inst, cls)."""
        return any(cls.__subclasscheck__(c)
                   for c in {type(inst), inst.__class__})

    def __subclasscheck__(cls, sub):
        """Implement issubclass(sub, cls)."""
        candidates = cls.__dict__.get("__subclass__", set()) | {cls}
        return any(c in candidates for c in sub.mro())

class Sequence(metaclass=ABCMeta):
    __subclass__ = {list, tuple}

assert issubclass(list, Sequence)
assert issubclass(tuple, Sequence)

class AppendableSequence(Sequence):
    __subclass__ = {list}

assert issubclass(list, AppendableSequence)
assert isinstance([], AppendableSequence)

assert not issubclass(tuple, AppendableSequence)
assert not isinstance((), AppendableSequence)
```

The next section proposes a full-fledged implementation.

## The `abc` Module: an ABC Support Framework

The new standard library module `abc` , written in pure Python, serves as an ABC support framework. It defines a metaclass `ABCMeta` and decorators `@abstractmethod` and `@abstractproperty` . A sample implementation is given by [13] .

The `ABCMeta` class overrides `__instancecheck__` and `__subclasscheck__` and defines a `register` method. The `register` method takes one argument, which must be a class; after the call `B.register(C)`, the call `issubclass(C, B)` will return True, by virtue of `B.__subclasscheck__(C)` returning True. Also, `isinstance(x, B)` is equivalent to `issubclass(x.__class__, B) or issubclass(type(x), B)`. (It is possible `type(x)` and `x.__class__` are not the same object, e.g. when x is a proxy object.)

These methods are intended to be called on classes whose metaclass is (derived from) `ABCMeta`; for example:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

MyABC.register(tuple)

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

The last two asserts are equivalent to the following two:

```
assert MyABC.__subclasscheck__(tuple)
assert MyABC.__instancecheck__(())
```

Of course, you can also directly subclass MyABC:

```
class MyClass(MyABC):
    pass

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

Also, of course, a tuple is not a `MyClass`:

```
assert not issubclass(tuple, MyClass)
assert not isinstance((), MyClass)
```

You can register another class as a subclass of `MyClass`:

```
MyClass.register(list)

assert issubclass(list, MyClass)
assert issubclass(list, MyABC)
```

You can also register another ABC:

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

AnotherClass.register(basestring)

MyClass.register(AnotherClass)

assert isinstance(str, MyABC)
```

That last assert requires tracing the following superclass-subclass relationships:

```
MyABC -> MyClass (using regular subclassing)
MyClass -> AnotherClass (using registration)
AnotherClass -> basestring (using registration)
basestring -> str (using regular subclassing)
```

The `abc` module also defines a new decorator, `@abstractmethod` , to be used to declare abstract methods. A class containing at least one method declared with this decorator that hasn't been overridden yet cannot be instantiated. Such methods may be called from the overriding method in the subclass (using `super` or direct invocation). For example:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @abstractmethod
    def foo(self): pass

A()  # raises TypeError

class B(A):
    pass

B()  # raises TypeError

class C(A):
    def foo(self): print(42)

C()  # works
```

**Note:** The `@abstractmethod` decorator should only be used inside a class body, and only for classes whose metaclass is (derived from) `ABCMeta` . 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 `register()` method are not affected.

**Implementation:** The `@abstractmethod` decorator sets the function attribute `__isabstractmethod__` to the value `True` . The `ABCMeta.__new__` method computes the type attribute `__abstractmethods__` as the set of all method names that have an `__isabstractmethod__` attribute whose value is true. It does this by combining the `__abstractmethods__` attributes of the base classes, adding the names of all methods in the new class dict that have a true `__isabstractmethod__` attribute, and removing the names of all methods in the new class dict that don't have a true `__isabstractmethod__` attribute. If the resulting `__abstractmethods__` set is non-empty, the class is considered abstract, and attempts to instantiate it will raise `TypeError` . (If this were implemented in CPython, an internal flag `Py_TPFLAGS_ABSTRACT` could be used to speed up this check [6] .)

**Discussion:** Unlike Java's abstract methods or C++'s pure abstract methods, abstract methods as defined here 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 framework using cooperative multiple-inheritance [7] , [8] .

A second decorator, `@abstractproperty` , is defined in order to define abstract data attributes. Its implementation is a subclass of the built-in `property` class that adds an `__isabstractmethod__` attribute:

```
class abstractproperty(property):
    __isabstractmethod__ = True
```

It can be used in two ways:

```
class C(metaclass=ABCMeta):

    # A read-only property:

    @abstractproperty
    def readonly(self):
        return self.__x

    # A read-write property (cannot use decorator syntax):

    def getx(self):
        return self.__x
    def setx(self, value):
        self.__x = value
    x = abstractproperty(getx, setx)
```

Similar to abstract methods, a subclass inheriting an abstract property (declared using either the decorator syntax or the longer form) cannot be instantiated unless it overrides that abstract property with a concrete property.

## ABCs for Containers and Iterators

The `collections` module will define ABCs necessary and sufficient to work with sets, mappings, sequences, and some helper types such as iterators and dictionary views. All ABCs have the above-mentioned `ABCMeta` as their metaclass.

The ABCs provide implementations of their abstract methods that are technically valid but fairly useless; e.g. `__hash__` returns 0, and `__iter__` returns an empty iterator. In general, the abstract methods represent the behavior of an empty container of the indicated type.

Some ABCs also provide concrete (i.e. non-abstract) methods; for example, the `Iterator` class has an `__iter__` method returning itself, fulfilling an important invariant of iterators (which in Python 2 has to be implemented anew by each iterator class). These ABCs can be considered "mix-in" classes.

No ABCs defined in the PEP override `__init__` , `__new__` , `__str__` or `__repr__` . Defining a standard constructor signature would unnecessarily constrain custom container types, for example Patricia trees or gdbm files. Defining a specific string representation for a collection is similarly left up to individual implementations.

**Note:** There are no ABCs for ordering operations ( `__lt__` , `__le__` , `__ge__` , `__gt__` ). Defining these in a base class (abstract or not) runs into problems with the accepted type for the second operand. For example, if class `Ordering` defined `__lt__` , one would assume that for any `Ordering` instances `x` and `y` , `x < y` would be defined (even if it just defines a partial ordering). But this cannot be the case: If both `list` and `str` derived from `Ordering` , this would imply that `[1, 2] < (1, 2)` should be defined (and presumably return False), while in fact (in Python 3000!) such "mixed-mode comparisons" operations are explicitly forbidden and raise `TypeError` . See PEP 3100 and [14] for more information. (This is a special case of a more general issue with operations that take another argument of the same type).

`Hashable`
The base class for classes defining `__hash__` . The `__hash__` method should return an integer. The abstract `__hash__` method always returns 0, which is a valid (albeit inefficient) implementation. **Invariant:** If classes `C1` and `C2` both derive from `Hashable` , the condition `o1 == o2` must imply `hash(o1) == hash(o2)` for all instances `o1` of `C1` and all instances `o2` of `C2` . In other words, two objects should never compare equal if they have different hash values.

Another constraint is that hashable objects, once created, should never change their value (as compared by `==` ) or their hash value. If a class cannot guarantee this, it should not derive from `Hashable` ; if it cannot guarantee this for certain instances, `__hash__` for those instances should raise a `TypeError` exception.

**Note:** being an instance of this class does not imply that an object is immutable; e.g. a tuple containing a list as a member is not immutable; its `__hash__` method raises `TypeError` . (This is because it recursively tries to compute the hash of each member; if a member is unhashable it raises `TypeError` .)

`Iterable`
The base class for classes defining `__iter__` . The `__iter__` method should always return an instance of `Iterator` (see below). The abstract `__iter__` method returns an empty iterator.

`Iterator`
The base class for classes defining `__next__` . This derives from `Iterable` . The abstract `__next__` method raises `StopIteration` . The concrete `__iter__` method returns `self` . Note the distinction between `Iterable` and `Iterator` : an `Iterable` can be iterated over, i.e. supports the `__iter__` methods; an `Iterator` is what the built-in function `iter()` returns, i.e. supports the `__next__` method.

`Sized`
The base class for classes defining `__len__` . The `__len__` method should return an `Integer` (see "Numbers" below) >= 0. The abstract `__len__` method returns 0. **Invariant:** If a class `C` derives from `Sized` as well as from `Iterable` , the invariant `sum(1 for x in c) == len(c)` should hold for any instance `c` of `C` .

`Container`
The base class for classes defining `__contains__` . The `__contains__` method should return a `bool` . The abstract `__contains__` method returns `False` . **Invariant:** If a class `C` derives from `Container` as well as from `Iterable` , then `(x in c for x in c)` should be a generator yielding only True values for any instance `c` of `C` .

**Open issues:** Conceivably, instead of using the ABCMeta metaclass, these classes could override `__instancecheck__` and `__subclasscheck__` to check for the presence of the applicable special method; for example:

```
class Sized(metaclass=ABCMeta):
    @abstractmethod
    def __hash__(self):
        return 0
    @classmethod
    def __instancecheck__(cls, x):
        return hasattr(x, "__len__")
    @classmethod
    def __subclasscheck__(cls, C):
        return hasattr(C, "__bases__") and hasattr(C, "__len__")
```

This has the advantage of not requiring explicit registration. However, the semantics are hard to get exactly right given the confusing semantics of instance attributes vs. class attributes, and that a class is an instance of its metaclass; the check for `__bases__` is only an approximation of the desired semantics. **Strawman:** Let's do it, but let's arrange it in such a way that the registration API also works.

## Sets

These abstract classes represent read-only sets and mutable sets. The most fundamental set operation is the membership test, written as `x in s` and implemented by `s.__contains__(x)` . This operation is already defined by the `Container` class defined above. Therefore, we define a set as a sized, iterable container for which certain invariants from mathematical set theory hold.

The built-in type `set` derives from `MutableSet` . The built-in type `frozenset` derives from `Set` and `Hashable` .

`Set`

This is a sized, iterable container, i.e., a subclass of `Sized` , `Iterable` and `Container` . Not every subclass of those three classes is a set though! Sets have the additional invariant that each element occurs only once (as can be determined by iteration), and in addition sets define concrete operators that implement the inequality operations as subclass/superclass tests. In general, the invariants for finite sets in mathematics hold. [11]

Sets with different implementations can be compared safely, (usually) efficiently and correctly using the mathematical definitions of the subclass/superclass operations for finite sets. The ordering operations have concrete implementations; subclasses may override these for speed but should maintain the semantics. Because `Set` derives from `Sized` , `__eq__` may take a shortcut and return `False` immediately if two sets of unequal length are compared. Similarly, `__le__` may return `False` immediately if the first set has more members than the second set. Note that set inclusion implements only a partial ordering; e.g. `{1, 2}` and `{1, 3}` are not ordered (all three of `<` , `==` and `>` return `False` for these arguments). Sets cannot be ordered relative to mappings or sequences, but they can be compared to those for equality (and then they always compare unequal).

This class also defines concrete operators to compute union, intersection, symmetric and asymmetric difference, respectively `__or__` , `__and__` , `__xor__` and `__sub__` . These operators should return instances of `Set` . The default implementations call the overridable class method `_from_iterable()` with an iterable argument. This factory method's default implementation returns a `frozenset` instance; it may be overridden to return another appropriate `Set` subclass.

Finally, this class defines a concrete method `_hash` which computes the hash value from the elements. Hashable subclasses of `Set` can implement `__hash__` by calling `_hash` or they can reimplement the same algorithm more efficiently; but the algorithm implemented should be the same. Currently the algorithm is fully specified only by the source code [15] .

**Note:** the `issubset` and `issuperset` methods found on the set type in Python 2 are not supported, as these are mostly just aliases for `__le__` and `__ge__` .

`MutableSet`

This is a subclass of `Set` implementing additional operations to add and remove elements. The supported methods have the semantics known from the `set` type in Python 2 (except for `discard` , which is modeled after Java):

`.add(x)`

Abstract method returning a `bool` that adds the element `x` if it isn't already in the set. It should return `True` if `x` was added, `False` if it was already there. The abstract implementation raises `NotImplementedError` .

`.discard(x)`
Abstract method returning a `bool` that removes the element `x` if present. It should return `True` if the element was present and `False` if it wasn't. The abstract implementation raises `NotImplementedError` .

`.pop()`
Concrete method that removes and returns an arbitrary item. If the set is empty, it raises `KeyError` . The default implementation removes the first item returned by the set's iterator.

`.toggle(x)`
Concrete method returning a `bool` that adds x to the set if it wasn't there, but removes it if it was there. It should return `True` if `x` was added, `False` if it was removed.

`.clear()`
Concrete method that empties the set. The default implementation repeatedly calls `self.pop()` until `KeyError` is caught. ( **Note:** this is likely much slower than simply creating a new set, even if an implementation overrides it with a faster approach; but in some cases object identity is important.)

This also supports the in-place mutating operations `|=` , `&=` , `^=` , `-=` . These are concrete methods whose right operand can be an arbitrary `Iterable` , except for `&=` , whose right operand must be a `Container` . This ABC does not provide the named methods present on the built-in concrete `set` type that perform (almost) the same operations.

## Mappings

These abstract classes represent read-only mappings and mutable mappings. The `Mapping` class represents the most common read-only mapping API.

The built-in type `dict` derives from `MutableMapping` .

`Mapping`
A subclass of `Container` , `Iterable` and `Sized` . The keys of a mapping naturally form a set. The (key, value) pairs (which must be tuples) are also referred to as items. The items also form a set. Methods:

`.__getitem__(key)`
Abstract method that returns the value corresponding to `key` , or raises `KeyError` . The implementation always raises `KeyError` .

`.get(key, default=None)`
Concrete method returning `self[key]` if this does not raise `KeyError` , and the `default` value if it does.

`.__contains__(key)`
Concrete method returning `True` if `self[key]` does not raise `KeyError` , and `False` if it does.

`.__len__()`

Abstract method returning the number of distinct keys (i.e., the length of the key set).

`.__iter__()`
Abstract method returning each key in the key set exactly once.

`.keys()`
Concrete method returning the key set as a `Set` . The default concrete implementation returns a "view" on the key set (meaning if the underlying mapping is modified, the view's value changes correspondingly); subclasses are not required to return a view but they should return a `Set` .

`.items()`
Concrete method returning the items as a `Set` . The default concrete implementation returns a "view" on the item set; subclasses are not required to return a view but they should return a `Set` .

`.values()`
Concrete method returning the values as a sized, iterable container (not a set!). The default concrete implementation returns a "view" on the values of the mapping; subclasses are not required to return a view but they should return a sized, iterable container.

The following invariants should hold for any mapping `m` :

```
len(m.values()) == len(m.keys()) == len(m.items()) == len(m)
[value for value in m.values()] == [m[key] for key in m.keys()]
[item for item in m.items()] == [(key, m[key]) for key in m.keys()]
```

i.e. iterating over the items, keys and values should return results in the same order.

`MutableMapping`
A subclass of `Mapping` that also implements some standard mutating methods. Abstract methods include `__setitem__` , `__delitem__` . Concrete methods include `pop` , `popitem` , `clear` , `update` . **Note:** `setdefault` is *not* included. **Open issues:** Write out the specs for the methods.

## Sequences

These abstract classes represent read-only sequences and mutable sequences.

The built-in `list` and `bytes` types derive from `MutableSequence` . The built-in `tuple` and `str` types derive from `Sequence` and `Hashable` .

`Sequence`
A subclass of `Iterable` , `Sized` , `Container` . It defines a new abstract method `__getitem__` that has a somewhat complicated signature: when called with an integer, it returns an element of the sequence or raises `IndexError` ; when called with a `slice` object, it returns another `Sequence` . The concrete `__iter__` method iterates over the elements using `__getitem__` with integer arguments 0, 1, and so on, until `IndexError` is raised. The length should be equal to the number of values returned by the iterator.

**Open issues:** Other candidate methods, which can all have default concrete implementations that only depend on `__len__` and `__getitem__` with an integer argument: `__reversed__` , `index` , `count` , `__add__` , `__mul__` .

`MutableSequence`
A subclass of `Sequence` adding some standard mutating methods. Abstract mutating methods: `__setitem__` (for integer indices as well as slices), `__delitem__` (ditto), `insert` . Concrete mutating methods: `append` , `reverse` , `extend` , `pop` , `remove` . Concrete mutating operators: `+=` , `*=` (these mutate the object in place). **Note:** this does not define `sort()` -- that is only required to exist on genuine `list` instances.