interface Documentation

Release 1.4.0

Scott Sanderson

Apr 21, 2021

Contents:

`interface` is a library for declaring interfaces and for statically asserting that classes implement those interfaces. It provides stricter semantics than Python's built-in `abc` module, and it aims to produce *exceptionally useful error messages* when interfaces aren't satisfied.

`interface` supports Python 2.7 and Python 3.4+.

CHAPTER 1

Quick Start

```
from interface import implements, Interface

class MyInterface(Interface):

    def method1(self, x):
        pass

    def method2(self, x, y):
        pass


class MyClass(implements(MyInterface)):

    def method1(self, x):
        return x * 2

    def method2(self, x, y):
        return x + y
```

1.1 Why Interfaces?

1.1.1 What is an Interface?

In software generally, an **interface** is a description of the **capabilities** provided by a unit of code. In object-oriented languages like Python, interfaces are often defined by a collection of **method signatures** which must be provided by a class.

In `interface`, an interface is a subclass of `interface.Interface` that defines one or more methods with empty bodies. For example, the interface definition for a simple Key-Value Store might look like this:

```
class KeyValueStore(interface.Interface):

    def get(self, key):
        """Get the value for ``key``.
        """

    def set(self, key, value):
        """Set the value for ``key`` to ``value``.
        """

    def delete(self, key):
        """Delete the value for ``key``.
        """
```

1.1.2 Why Are Interfaces Useful?

Interfaces are useful for specifying the **contract** between two units of code. By marking that a type **implements** an interface, we give a programatically-verifiable guarantee that the implementation provides the methods specified by the interface signature. That guarantee makes it easier to write code that can work with **any** implementation of an interface, and it also serves as a form of documentation.

1.2 Using `interface`

1.2.1 Declaring Interfaces

An interface describes a collection of methods and properties that should be provided by implementors.

We create an interface by subclassing from `interface.Interface` and defining stubs for methods that should be part of the interface:

```
class MyInterface(interface.Interface):

    def method1(self, x, y, z):
        pass

    def method2(self):
        pass
```

1.2.2 Implementing Interfaces

To declare that a class implements an interface `I`, that class should subclass from `implements(I)`:

```
class MyClass(interface.implements(MyInterface)):

    def method1(self, x, y, z):
        return x + y + z

    def method2(self):
        return "foo"
```

A class can implement more than one interface:

```
class MathStudent(Interface):

    def argue(self, topic):
        pass

    def calculate(self, x, y):
        pass


class PhilosophyStudent(Interface):

    def argue(self, topic):
        pass

    def pontificate(self):
        pass


class LiberalArtsStudent(implements(MathStudent, PhilosophyStudent)):

    def argue(self, topic):
        print(topic, "is", ["good", "bad"][random.choice([0, 1])])

    def calculate(self, x, y):
        return x + y

    def pontificate(self):
        print("I think what Wittgenstein was **really** saying is...")
```

Notice that interfaces can have intersecting methods as long as their signatures match.

1.2.3 Properties

Interfaces can declare non-method attributes that should be provided by implementations using `property`:

```
class MyInterface(interface.Interface):

    @property
    def my_property(self):
        pass
```

Implementations are required to provide a `property` with the same name.

```
class MyClass(interface.implements(MyInterface)):

    @property
    def my_property(self):
        return 3
```

1.2.4 Default Implementations

Sometimes we have a method that should be part of an interface, but which can be implemented in terms of other interface methods. When this happens, you can use `interface.default` to provide a default implementation of a method.

```
class ReadOnlyMapping(interface.Interface):

    def get(self, key):
        pass

    def keys(self):
        pass

    @interface.default
    def get_all(self):
        out = {}
        for k in self.keys():
            out[k] = self.get(k)
        return out
```

Implementors are not required to implement methods with defaults:

```
class MyReadOnlyMapping(interface.implements(ReadOnlyMapping)):
    def __init__(self, data):
        self._data = data

    def get(self, key):
        return self._data[key]

    def keys(self):
        return self._data.keys()

    # get_all(self) will automatically be copied from the interface default.
```

Default implementations should always be implemented in terms of other interface methods.

In Python 3, `default` will show a warning if a default implementation uses non-interface members of an object:

```
class ReadOnlyMapping(interface.Interface):

    def get(self, key):
        pass

    def keys(self):
        pass

    @interface.default
    def get_all(self):
        # This is supposed to be a default implementation for **any**
        # ReadOnlyMapping, but this implementation assumes that 'self' has
        # an _data attribute that isn't part of the interface!
        return self._data.keys()
```

Running the above example displays a warning about the default implementation of `get_all`:

```
$ python example.py
example.py:4: UnsafeDefault: Default for ReadOnlyMapping.get_all uses non-interface␣
↪attributes.

The following attributes are used but are not part of the interface:
  - _data
```

(continues on next page)

(continued from previous page)

```
Consider changing ReadOnlyMapping.get_all or making these attributes part of␣
↪ReadOnlyMapping.
   class ReadOnlyMapping(interface.Interface):
```

1.2.5 Interface Subclassing

Interfaces can inherit requirements from other interfaces via subclassing. For example, if we want to create interfaces for read-write and read-only mappings, we could do so as follows:

```
class ReadOnlyMapping(interface.Interface):
    def get(self, key):
        pass

    def keys(self):
        pass


class ReadWriteMapping(ReadOnlyMapping):

    def set(self, key, value):
        pass

    def delete(self, key):
        pass
```

An interface that subclasses from another interface inherits all the function signature requirements from its parent interface. In the example above, a class implementing `ReadWriteMapping` would have to implement `get`, `keys`, `set`, and `delete`.

Warning: Subclassing from an interface is not the same as implementing an interface. Subclassing from an interface **creates a new interface** that adds additional methods to the parent interface. Implementing an interface creates a new class whose method signatures must be compatible with the interface being implemented.

1.3 Error Detection

`interface` aims to provide clear, detailed, and complete error messages when an interface definition isn't satisfied.

An implementation can fail to implement an interface in a variety of ways:

1.3.1 Missing Methods

Implementations must define all the methods declared in an interface.

```
from interface import implements, Interface

class MathStudent(Interface):

    def argue(self, topic):
        pass
```

(continues on next page)

(continued from previous page)

```
    def prove(self, theorem):
        pass

    def calculate(self, x, y, z):
        pass


class Freshman(implements(MathStudent)):

    def argue(self, topic):
        print(topic, "is terrible!")
```

The above example produces the following error message:

```
Traceback (most recent call last):
  ...
interface.interface.InvalidImplementation:
class Freshman failed to implement interface MathStudent:

The following methods of MathStudent were not implemented:
  - calculate(self, x, y, z)
  - prove(self, theorem)
```

1.3.2 Methods with Incompatible Signatures

Implementations must define interface methods with compatible signatures:

```
from interface import implements, Interface

class MathStudent(Interface):

    def argue(self, topic):
        pass

    def prove(self, theorem):
        pass

    def calculate(self, x, y, z):
        pass


class SloppyMathStudent(implements(MathStudent)):

    def argue(self, topic):
        print(topic, "is terrible!")

    def prove(self, lemma):
        print("That's almost a theorem, right?")

    def calculate(self, x, y):
        return x + y
```

```
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
interface.interface.InvalidImplementation:
class SloppyMathStudent failed to implement interface MathStudent:

The following methods of MathStudent were implemented with invalid signatures:
  - calculate(self, x, y) != calculate(self, x, y, z)
  - prove(self, lemma) != prove(self, theorem)
```

1.3.3 Method/Property Mismatches

If an interface defines an attribute as a `property`, the corresponding implementation attribute must also be a `property`:

```
class Philosopher(Interface):
    @property
    def favorite_philosopher(self):
        pass

class AnalyticPhilosopher(implements(Philosopher)):

    def favorite_philosopher(self):  # oops, should have been a property!
        return "Ludwig Wittgenstein"
```

```
Traceback (most recent call last):
  ...
interface.interface.InvalidImplementation:
class AnalyticPhilosopher failed to implement interface Philosopher:

The following methods of Philosopher were implemented with incorrect types:
  - favorite_philosopher: 'function' is not a subtype of expected type 'property'
```

1.4 `interface` vs. `abc`

The Python standard library `abc` (**A**bstract **B**ase **C**lass) module is often used to define and verify interfaces.

`interface` differs from Python's `abc` module in two important ways:

1. Interface requirements are checked at class creation time, rather than at instance creation time. This means that `interface` can tell you if a class fails to implement an interface even if you never create any instances of that class.
2. `interface` requires that method signatures of implementations are compatible with signatures declared in interfaces. For example, the following code using `abc` does not produce an error:

```
>>> from abc import ABCMeta, abstractmethod
>>> class Base(metaclass=ABCMeta):
...
...     @abstractmethod
...     def method(self, a, b):
...         pass
...
>>> class Implementation(MyABC):
...
...     def method(self):  # Missing a and b parameters.
```

(continues on next page)

(continued from previous page)

```
...         return "This shouldn't work."
...
>>> impl = Implementation()
>>>
```

The equivalent code using `interface` produces an error telling us that the implementation method doesn't match the interface:

```
>>> from interface import implements, Interface
>>> class I(Interface):
...     def method(self, a, b):
...         pass
...
>>> class C(implements(I)):
...     def method(self):
...         return "This shouldn't work"
...
TypeError:
class C failed to implement interface I:

The following methods were implemented but had invalid signatures:
  - method(self) != method(self, a, b)
```

1.5 Examples

Suppose we're writing an application that needs to load and save user preferences.

We expect that we may want to manage preferences differently in different contexts, so we separate out our preference-management code into its own class, and our main application looks like this:

```
class MyApplication:

    def __init__(self, preferences):
        self.preferences = preferences

    def save_resolution(self, resolution):
        self.preferences.set('resolution', resolution)

    def get_resolution(self):
        return self.preferences.get('resolution')

    def save_volume(self, volume):
        self.preferences.set('volume', volume)

    def get_volume(self):
        return self.preferences.get('volume')

    ...
```

When we ship our application to users, we store preferences as a json file on the local hard drive:

```
class JSONFileStore:

    def __init__(self, path):
```

(continues on next page)

(continued from previous page)

```
        self.path = path

    def get(self, key):
        with open(self.path) as f:
            return json.load(f)[key]

    def set(self, key, value):
        with open(self.path) as f:
            data = json.load(f)

        data[key] = value

        with open(self.path, 'w') as f:
            json.dump(data, f)
```

In testing, however, we want to use a simpler key-value store that stores preferences in memory:

```
class InMemoryStore:

    def __init__(self):
        self.data = {}

    def get(self, key):
        return self.data[key]

    def set(self, key, value):
        self.data[key] = value
```

Later on, we add a cloud-sync feature to our application, so we add a third implementation that stores user preferences in a database:

```
class SQLStore:
    def __init__(self, user, connection):
        self.user = user
        self.connection = connection

    def get(self, key):
        self.connection.execute(
            "SELECT * FROM preferences where key=%s and user=%s",
            [self.key, self.user],
        )

    def set(self, key, value):
        self.connection.execute(
            "INSERT INTO preferences VALUES (%s, %s, %s)",
            [self.user, self.key, value],
)
```

As the number of `KeyValueStore` implementations grows, it becomes more and more difficult for us to make changes to our application. If we add a new method to any of the key-value stores, we can't use it in the application unless we add the same method to the other implementations, but in a large codebase we might not even know what other implementations exist!

By declaring `KeyValueStore` as an `Interface` we can get `interface` to help us keep our implementations in sync:

```
class KeyValueStore(interface.Interface):

    def get(self, key):
        pass

    def set(self, key, value):
        pass

class JSONFileStore(implements(KeyValueStore)):
    ...

class InMemoryStore(implements(KeyValueStore)):
    ...

class SQLStore(implements(KeyValueStore)):
    ...
```

Now, if we add a method to the interface without adding it to an implementation, we'll get a helpful error at class definition time.

For example, if we add a `get_default` method to the interface but forget to add it to `SQLStore`:

```
class KeyValueStore(interface.Interface):

    def get(self, key):
        pass

    def set(self, key, value):
        pass

    def get_default(self, key, default):
        pass


class SQLStore(interface.implements(KeyValueStore)):

    def get(self, key):
        pass

    def set(self, key, value):
        pass

    # We forgot to define get_default!
```

We get the following error at import time:

```
$ python example.py
Traceback (most recent call last):
  File "example.py", line 16, in <module>
    class SQLStore(interface.implements(KeyValueStore)):
  File "/home/ssanderson/projects/interface/interface/interface.py", line 394, in __
↪new__
    raise errors[0]
  File "/home/ssanderson/projects/interface/interface/interface.py", line 376, in __
↪new__
    defaults_from_iface = iface.verify(newtype)
  File "/home/ssanderson/projects/interface/interface/interface.py", line 191, in␣
↪verify
```

(continues on next page)

(continued from previous page)

```
    raise self._invalid_implementation(type_, missing, mistyped, mismatched)
interface.interface.InvalidImplementation:
class SQLStore failed to implement interface KeyValueStore:

The following methods of KeyValueStore were not implemented:
  - get_default(self, key, default)
```

1.6 API Reference

CHAPTER 2

Indices and tables

- genindex
- modindex
- search