
Implements Documentation

Release 0.1.3

Kamil Sindi

May 31, 2017

Contents

1	Implements	1
1.1	Install	1
1.2	Advantages	1
1.3	Usage	1
1.4	Justification	2
1.5	Credit	4
1.6	Test	4
1.7	License	4
2	Indices and tables	5

CHAPTER 1

Implements

Pythonic interfaces using decorators

Install

Implements is available on PyPI and can be installed with `pip`:

```
pip install implements
```

Advantages

1. Favor composition over inheritance.
2. Inheriting from multiple classes can be problematic, especially when the superclasses have the same method name but different signatures. Implements will throw a descriptive error if that happens to ensure integrity of contracts.
3. The decorators are evaluated at import time. Any errors will be raised then and not when an object is instantiated or a method is called.
4. It's cleaner. Using decorators makes it clear we want share behavior. Also, arguments are not allowed to be renamed.
5. Codebase is tiny: you can just copy the file over. This repo exists more for test coverage.

Usage

```
from implements import Interface, implements
```

```
class Duck:
    def __init__(self, age):
        self.age = age

class Flyable(Interface):
    @staticmethod
    def migrate(direction):
        pass

    def fly(self) -> str:
        pass

class Quackable(Interface):
    def fly(self) -> bool:
        pass

    def quack(self):
        pass

@implements(Flyable)
@implements(Quackable)
class MallardDuck(Duck):
    def __init__(self, age):
        super(MallardDuck, self).__init__(age)

    def migrate(self, dir):
        return True

    def fly(self):
        pass
```

The above would throw the following errors:

```
NotImplementedError: 'MallardDuck' must implement method 'fly((self) -> bool)'.
↳ defined in interface 'Quackable'
NotImplementedError: 'MallardDuck' must implement method 'quack((self))' defined in
↳ interface 'Quackable'
NotImplementedError: 'MallardDuck' must implement method 'migrate((direction))'.
↳ defined in interface 'Flyable'
```

You can find a more detailed example in `example.py` and by looking at `tests.py`.

Justification

There are currently two idiomatic ways to rewrite the above example.

The first way is to write base classes with mixins raising `NotImplementedError` in each method.

```
class Duck:
    def __init__(self, age):
        self.age = age
```

```

class Flyable:
    @staticmethod
    def migrate(direction):
        raise NotImplementedError("Flyable is an abstract class")

    def fly(self) -> str:
        raise NotImplementedError("Flyable is an abstract class")

class Quackable:
    def fly(self) -> bool:
        raise NotImplementedError("Quackable is an abstract class")

    def quack(self):
        raise NotImplementedError("Quackable is an abstract class")

class MallardDuck(Duck, Quackable, Flyable):

    def __init__(self, age):
        super(MallardDuck, self).__init__(age)

    def migrate(self, dir):
        return True

    def fly(self):
        pass

```

But there are a couple drawbacks implementing it this way:

1. We would only get a `NotImplementedError` when calling `quack` which can happen much later during runtime. Also, raising `NotImplementedError` everywhere looks clunky.
2. It's unclear without checking each parent class where `super` is being called.
3. Similarly the return types of `fly` in `Flyable` and `Quackable` are different. Someone unfamiliar with Python would have to read up on [Method Resolution Order](#).
4. The writer of `MallardDuck` made method `migrate` an instance method and renamed the argument to `dir` which is confusing.
5. We really want to be differentiating between behavior and inheritance.

The advantage of using `implements` is it looks cleaner and you would get errors at import time instead of when the method is actually called.

Another way is to use abstract base classes from the built-in `abc` module:

```

from abc import ABCMeta, abstractmethod, abstractstaticmethod

class Duck(metaclass=ABCMeta):
    def __init__(self, age):
        self.age = age

class Flyable(metaclass=ABCMeta):
    @abstractstaticmethod
    def migrate(direction):
        pass

```

```
@abstractmethod
def fly(self) -> str:
    pass

class Quackable(metaclass=ABCMeta):
    @abstractmethod
    def fly(self) -> bool:
        pass

    @abstractmethod
    def quack(self):
        pass

class MallardDuck(Duck, Quackable, Flyable):
    def __init__(self, age):
        super(MallardDuck, self).__init__(age)

    def migrate(self, dir):
        return True

    def fly(self):
        pass
```

Using abstract base classes has the advantage of throwing an error earlier on instantiation if a method is not implemented; also, there are static analysis tools that warn if two methods have different signatures. But it doesn't solve issues 2-4 and implements will throw an error even earlier in import. It also in my opinion doesn't look pythonic.

Credit

Implementation was inspired by a [PR](#) of @elifiner.

Test

Running unit tests:

```
make test
```

Running linter:

```
make lint
```

Running tox:

```
make test-all
```

License

MIT

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`