Interfaces

Substitution Principle

```
>>> pets = [Cat("Momento"), LapDog("Harry"), Duck("Don"),
...         Owl("Jerry"), Dog("Fido"), Bird("Pascal")]
>>> for pet in pets:
...     print(pet.describe())
The cat says: Meow!
The lap dog says: Yip!
The duck says: Quack!
The owl says: Hoooo!
The dog says: Woof!
The bird says: Chirp!
```

What’s the implied interface of this code:

```
for pet in pets:
    print(pet.describe())
```

Describing Pets

```python
class Pet:
    def __init__(self, name):
        self.name = name
class Dog(Pet):
    def describe(self):
        return "The dog says: Woof!"
class Cat(Pet):
    def describe(self):
        return "The cat says: Meow!"
# And so on for the other animals.
```

`Pet` does not have `describe()`. But each subclass *must* have it.

```python
# No describe(). Creates a runtime error; may not be caught during dev.
class BadSheep(Pet):
    def about(self):
        return "The baaad sheep says: Baaaa!"
```

How do you make this requirement explicit?

Abstract Method

Python does not have interfaces.
But it does have abstract base classes.

```
import abc

class Pet(metaclass=abc.ABCMeta):
    def __init__(self, name):
        self.name = name
    @abc.abstractmethod
    def describe(self):
        pass
```

`@abc.abstractmethod` applies a *decorator*.

It ensures subclasses must implement `describe()` before they can be instantiated.

Bad Sheep

```python
import abc
class Pet(metaclass=abc.ABCMeta):
    def __init__(self, name):
        self.name = name
    @abc.abstractmethod
    def describe(self):
        pass
class BadSheep(Pet):
    def about(self):
        return "The baaad sheep says: Baaaa!"
gary = BadSheep("Gary")
print(gary.name + ": " + gary.describe())
```

```
Traceback (most recent call last):
  File "gary-the-bad-sheep.py", line 13, in <module>
    gary = BadSheep("Gary")
TypeError: Can't instantiate abstract class BadSheep with abstract methods
describe
```

Good Sheep

```
import abc
class Pet(metaclass=abc.ABCMeta):
    def __init__(self, name):
        self.name = name
    @abc.abstractmethod
    def describe(self):
        pass

class GoodSheep(Pet):
    def describe(self):
        return "The good sheep says: Baaaa! (Politely.)"

tom = GoodSheep("Tom")
print(tom.name + ": " + tom.describe())
```

```
Tom: The good sheep says: Baaaa! (Politely.)
```

What’s in an abstract method?

Typically, the body of the abstract method will just be "`pass`". In other words, empty.

```
    @abc.abstractmethod
    def describe(self):
        pass
```

There are rare exceptions to this.

We’ll have more to say about abstract base classes and methods later.

Substitution Principle

The **Substitution Principle**:

Anywhere in your code you can use an instance of a class, you can also use an instance of its subclasses.

So if you can use an instance of `Pet`, you can also use an instance of `Dog`.

This means methods in a subclass have *compatible signatures* with those in the superclass. There are some subtleties.

(Usually called the "Liskov Substitution Principle", after the computer scientist who discovered and developed it. Formally expressed using mathematical type theory.)

Need @abstractmethod?

People will sometimes not bother in practice.

Their Python code is written to assume the interface, and generate a runtime error if that fails.

This is called **duck typing**. Often it's perfectly fine. The larger the application, the more important abstract base classes become.

```
class Pet:
    def __init__(self, name):
        self.name = name
    # No describe() abstract method.

class Dog(Pet):
    def describe(self):
        return "The dog says: Woof!"
```

Stock View

Here's the `StockView` class:

```python
class StockView:
    def params(self, model):
        if model.is_bullish():
            sentiment = 'Bullish'
        else:
            sentiment = 'Bearish'
        return {
            'name': model.symbol,
            'price': model.close_price,
            'sentiment': sentiment,
            }
    def render(self, model):
        params = self.params(model)
        return '{name}: ${price:0.2f} ({sentiment})'.format_map(params)
```

It's used as a base class, as well as a default text implementation. Let's separate these two.

StockTextView

```python
class StockView(metaclass=abc.ABCMeta):
    def params(self, model):
        if model.is_bullish():
            sentiment = 'Bullish'
        else:
            sentiment = 'Bearish'
        return {
            'name': model.symbol,
            'price': model.close_price,
            'sentiment': sentiment,
            }
    @abc.abstractmethod
    def render(self, model):
        pass

class StockTextView(StockView):
    def render(self, model):
        params = self.params(model)
        return '{name}: ${price:0.2f} ({sentiment})'.format_map(params)
```