Methods And Inheritance

Yipping Dogs

Our lap-dog hierarchy:

```python
class Pet:
    sound = ""
    def __init__(self, name):
        self.name = name
    def describe(self):
        kind_of_pet = self.__class__.__name__.lower()
        return "The {} says: {}!".format(kind_of_pet, self.sound)
class Dog(Pet):
    sound = "Woof"
class LapDog(Dog):
    sound = "Yip"
class LoudLapDog(LapDog):
    sound = "YIP"
```

Two problems:

- `LapDog.sound` and `LoudLapDog.sound` are repetitive.
- Suppose we decide a loud lap dog should say "YIP!YIP!YIP!". How?

Step One: LoudLapDog.speak()

```
class Pet:
    sound = ""
    def __init__(self, name):
        self.name = name
    def speak(self):
        return self.sound + "!"
    def describe(self):
        kind_of_pet = self.__class__.__name__.lower()
        return "The {} says: {}".format(kind_of_pet, self.speak())
class Dog(Pet):
    sound = "Woof"
class LapDog(Dog):
    sound = "Yip"
class LoudLapDog(LapDog):
    sound = "YIP"
```

```
>>> nacho = LoudLapDog("Nacho")
>>> nacho.describe()
'The loudlapdog says: YIP!'
```

Not there yet...

Redefining speak()

Subclasses can define their own versions of a method.

This will **mask** the version inherited from the superclass.

```
class LoudLapDog(LapDog):
    # No need to define self.sound
    def speak(self):
        return "YIP!YIP!YIP!"
```

```
>>> nacho = LoudLapDog("Nacho")
>>> nacho.describe()
'The loudlapdog says: YIP!YIP!YIP!'
```

This is the behavior we want. BUT:

- Our code repeats itself
- It won’t update if we change `LapDog.sound`

super()

The built-in function `super()` is used in methods of a subclass.

It lets you call methods on `self`, from the perspective of the superclass.

```
class LoudLapDog(LapDog):
    def speak(self):
        # super().speak() refers to LapDog.speak()
        return super().speak().upper() * 3
```

```
>>> nacho = LoudLapDog("Nacho")
>>> nacho.describe()
'The loudlapdog says: YIP!YIP!YIP!'
```

Best of all worlds!

Design your class hierarchies so subclasses can inject their own custom behavior.

Full Hierarchy

```python
class Pet:
    sound = ""
    def __init__(self, name):
        self.name = name
    def speak(self):
        return self.sound + "!"
    def describe(self):
        kind_of_pet = self.__class__.__name__.lower()
        return "The {} says: {}".format(
            kind_of_pet, self.speak())
class Dog(Pet):
    sound = "Woof"
class LapDog(Dog):
    sound = "Yip"
class LoudLapDog(LapDog):
    def speak(self):
       return super().speak().upper() * 3
```

Stock HTML View

Let’s create an HTML view of the Stock model, with an icon indicating the recommendation:

```python
class StockHTMLView(StockView):
    def __init__(self, template):
        self.template = template

    def params(self, model):
        params = super().params(model)
        if model.is_bullish():
            icon = 'buy.jpg'
        else:
            icon = 'hold.jpg'
        params['icon'] = icon
        return params

    def render(self, model):
        params = self.params(model)
        return self.template.format_map(params)
```

Rendering HTML

```
STOCK_HTML_TEMPLATE = '''
<html>
  <title>Stock Report for {name}</title>
  <body>
     <dl><dt>Name:</dt><dd>{name}</dd>
         <dt>Closing price:</dt><dd>{price}</dd>
         <dt>Recommendation:</dt><img src="{icon}"/><dd></dd>
     </dl></body</html>
'''.strip()
```

```
>>> model = StockModel('FB', 172.06, 183.37, 76670183, 25219450)
>>> view = StockHTMLView(STOCK_HTML_TEMPLATE)
>>> print(view.render(model))
<html>
  <title>Stock Report for FB</title>
  <body>
     <dl><dt>Name:</dt><dd>FB</dd>
         <dt>Closing price:</dt><dd>183.37</dd>
         <dt>Recommendation:</dt><img src="buy.jpg"/><dd></dd>
     </dl></body</html>
```