Access Control

Access Control

Many languages have access control; variables and methods designated as public, private, or protected.

```
/* Java code */
class SignalParser {
    private String state;
    public SignalParser() { this.state = "waiting"; }
    protected void startReceiving() {
        this.state = "receiving";
    }
    public void receive(String data) {
        this.startReceiving();
        System.out.println("RECEIVING: " + data);
    }
    public String getState() { return this.state; }
}
```

Python handles this concept very differently.

"Protected" Attributes

Convention: prefixing with a single underscore means "don't rely on this being available outside the class." Equivalent to "protected".

```
class SignalParser:
    def __init__(self):
        self._state = 'waiting'
    def _start_receiving(self):
        self._state = 'receiving'
    def receive(self, data):
        self._start_receiving()
        print('RECEIVING: ' + data)
    def get_state(self):
        return self._state
```

"Protected" Attributes

Strictly speaking, EVERYTHING on a Python object is public. There's no way to change that.

```
>>> sp = SignalParser()
>>> sp.receive("Heads up!")
RECEIVING: Heads up!
>>> print("Mwahaha, I can see your hidden state is " + sp._state)
Mwahaha, I can see your hidden state is receiving
```

Class-Private

Hide from subclasses: prefix with **two** underscores.

```
class SignalParser:
    __state: str
    def __init__(self):
        self.__state = 'waiting'
    def _start_receiving(self):
        self.__state = 'receiving'
    def receive(self, data):
        self._start_receiving()
        print('RECEIVING: ' + data)
    def get_state(self):
        return self.__state
```

Class-Private

This mangles the name, so it's not visible in subclasses. Like "private".

```
>>> sp = SignalParser()
>>> print("Mwahaha, I can see your hidden state is " + sp.__state)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SignalParser' object has no attribute '__state'
```

But even this can be circumvented:

```
>>> print("Clever, but not clever enough! " + sp._SignalParser__state)
Clever, but not clever enough! waiting
```

Example: CappedList class

A *capped list* is like a list, except it has a maximum length. As you keep adding new elements, old ones are forgotten.

```
class CappedList:
    def __init__(self, size):
        self.size = size
        self.position = 0
        self.data = []
    def append(self, value):
        if len(self.data) == self.position:
            self.data.append(value)
        else:
            self.data[self.position] = value
        self.position += 1
        if self.position == self.size:
            self.position = 0
    def all_values(self):
        pivot = self.position
        return self.data[pivot:] + self.data[0:pivot]
```

Using the CappedList

```
>>> # Closing stock prices for GOOG, 11/15-11/29, 2017
... all_prices = [
...     1020.91, 1032.5, 1019.09, 1018.38, 1034.49,
...     1035.96, 1040.61, 1054.21, 1047.41, 1021.66]
>>> prices = CappedList(5)
>>> for price in all_prices:
...     prices.append(price)
...
>>> prices.all_values()
[1035.96, 1040.61, 1054.21, 1047.41, 1021.66]
```

Exposing State

Look at the state set up in the constructor: `position`, `size`, and `data`. What needs to be exposed publically? What doesn't?

```
class CappedList:
    def __init__(self, size):
        self.size = size
        self.position = 0
        self.data = []
    def append(self, value):
        if len(self.data) == self.position:
            self.data.append(value)
        else:
            self.data[self.position] = value
        self.position += 1
        if self.position == self.size:
            self.position = 0
    def all_values(self):
        pivot = self.position
        return self.data[pivot:] + self.data[0:pivot]
```

Hiding State

```
class CappedList:
    def __init__(self, size):
        self.size = size
        self._data = []
        self.__position = 0
    def _increment(self):
        self.__position += 1
        if self.__position == self.size:
            self.__position = 0
    def append(self, value):
        if len(self._data) == self.__position:
            self._data.append(value)
        else:
            self._data[self.__position] = value
        self._increment()
    def all_values(self):
        pivot = self.__position
        return self._data[pivot:] + self._data[0:pivot]
```

Richer Stock Model

```
BULLISH = 1
NEUTRAL = 0
BEARISH = -1

class StockModel:
    def __init__(self, symbol, open_price, close_price,
                 volume, average_volume):
        self.symbol = symbol
        self.open_price = open_price
        self.close_price = close_price
        self.volume = volume
        self.average_volume = average_volume
```

Richer Stock Model

```
    def _price_ratio(self):
        return self.close_price / self.open_price

    def is_bullish(self):
        volume_ratio = self.volume / self.average_volume
        return self._price_ratio() > 1.02 and volume_ratio > 1.1

    def is_bearish(self):
        return self.price_ratio() < 0.97

    def sentiment(self):
        if self.is_bullish():
            return BULLISH
        elif self.is_bearish():
            return BEARISH
        else:
            return NEUTRAL
```

Base Stock View

```
class StockView:
    SENTIMENTS = {
        BULLISH: 'Bullish',
        NEUTRAL: 'Neutral',
        BEARISH: 'Bearish',
        }
    def params(self, model):
        sentiment = self.SENTIMENTS[model.sentiment()]
        return {
            'name': model.symbol,
            'price': model.close_price,
            'sentiment': sentiment,
            }

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

HTML View

```
class StockHTMLView(StockView):
    ICONS = {
        BULLISH: 'buy.jpg',
        NEUTRAL: 'hold.jpg',
        BEARISH: 'sell.jpg',
        }
    def __init__(self, template):
        self.template = template

    def params(self, model):
        params = super().params(model)
        params['icon'] = self.ICONS[model.sentiment()]
        return params

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

Lab: Simple Inheritance

Lab file: `integration.py`

- In `labs` folder
- When you are done, give a thumbs up...
- ... and then work on any remaining labs from this class