Python's Syntax For Classes

Data And Behavior

Here are some functions to create and work with a dictionary:

```
def new_person(first, last):
    return { "first": first, "last":  last }
def full_name(person):
    return person["first"] + " " + person["last"]
def formal_name(person, title):
    return title + " " + full_name(person)
```

In action:

```
>>> person = new_person("John", "Smith")
>>> full_name(person)
'John Smith'
>>> formal_name(person, "Mr.")
'Mr. John Smith'
```

This works. But what are its shortcomings?

Shortcomings

Everything depends on what keys are in the dict. And that dependency is spread over several unrelated components (functions).

1) It's easy to accidentally insert the wrong key, mispell one, etc. How can we prevent this?

1) What if the keys change? All code using them needs to be found, updated, and tested.

2) What if people start relying on the dict just having "first" and "last", and we want to add "middle"?

3) How do we organize the code, so that people know what functions they can and cannot use?

Also...

There may be many other dicts in our code that don't represent people. We have to spend mental energy deciding which dicts are for a person, and which are for something else.

Better: create our own new data type, *just* to represent a person.

Best: Tie the various functions to that data type, so everything's bundled together.

Classes

In Python, you can create a **class**. This is your own, custom type.

```
# A coin worth 25 cents.
class Quarter:
    value = 25
```

Notice that:

- `value` is indented inside the `class` block, and
- `Quarter` is capitalized. (A convention.)

Create an **instance** of that class by calling it like a function:

```
>>> coin = Quarter()
>>> coin.value
25
```

The `coin` object has a `value` attribute. We call this a **member variable**.

Distinct Objects

Each distinct object has an ID. And you can create many different instances of the same class.

```
>>> another_coin = Quarter()
>>> id(coin)
4480623000
>>> id(another_coin)
4480622888
```

Notice the different numbers.

Methods

A class can have **methods**:

```
class Quarter:
    value = 25
    def in_nickels(self):
        return self.value // 5
```

```
>>> quarter = Quarter()
>>> quarter.in_nickels()
5
```

This is a function attached to the class itself. Notice:

- It's indented, and
- The first argument is `self`.

Methods Taking Arguments

```
import time
class FileLogger:
    filename = "log.txt"
    def write_message(self, message):
        # Append message to log file.
        line = "{} {}\n".format(int(time.time()), message)
        with open(self.filename, "a") as outfile:
            outfile.write(line)
```

```
# Running these lines:
logger = FileLogger()
logger.write_message("Hello!")

# ... will add this to log.txt:
1514095058 Hello!
```

The __init__ Constructor

You can also write `Quarter` this way:

```
class QuarterV2:
    def __init__(self):
        self.value = 25
    def to_pennies(self):
        return self.value
```

```
>>> quarter = QuarterV2()
>>> quarter.value
25
```

The `__init__` method is special. It's executed automatically, once, when the object is created.

In object-oriented programming, this is called a **constructor**.

Person Class

Going back to this function:

```
def new_person(first, last):
    return { "first": first, "last":  last }
```

This is like a constructor. You can instead write:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
```

```
>>> person = Person("John", "Smith")
>>> person.first
'John'
>>> person.last
'Smith'
```

New Methods

Remember, we also had these functions:

```
# For the dict version.
def full_name(person):
    return person["first"] + " " + person["last"]
def formal_name(person, title):
    return title + " " + full_name(person)
```

You can convert them to methods of `Person`:

1) Indent inside the `class Person:` block.

2) Replace the `person` argument with `self`.

3) Inside the method, reference `self.first` and `self.last`.

Full Person Class

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    def full_name(self):
        return self.first + " " + self.last

    def formal_name(self, title):
        return title + " " + self.full_name()
```

Notice in `formal_name()`, you call `self.full_name()`.

Practice

Create a file named `person.py`. Write in the following program:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    def full_name(self):
        return self.first + " " + self.last
    def formal_name(self, title):
        return title + " " + self.full_name()
person = Person("Your", "Name")
print(person.formal_name("Dr."))
```

Run it, and verify it prints out your name with "Dr.":

```
Dr. Aaron Maxwell
```

Extra Credit: Modify `Person` to also accept a middle name.

Lab: Classes and Methods

Lab file: `classesandmethods.py`

- In `labs` folder
- When you are done, give a thumbs up...
- ... and then do `classesandmethods_extra.py`

Instructions: `LABS.txt` in courseware.

Single-Responsibility Principle

A principle in two parts:

1. Each class covers one piece of functionality. And
2. Everything about that piece is encapsulated inside the class.

An important organizational principle.

Model & View

This `StockModel` represents a data abstraction:

```
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

    def is_bullish(self):
        price_ratio = self.close_price / self.open_price
        volume_ratio = self.volume / self.average_volume
        return price_ratio > 1.02 and volume_ratio > 1.1
```

Model & View

This `StockView` encapsulates how to render it for the user:

```
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)
```

Demo

This default view renders it as brief text.

```
>>> model = StockModel('AAPL', 159.29, 163.05, 44035531, 22509937)
>>> view = StockView()
>>> view.render(model)
'AAPL: $163.05 (Bullish)'
```