What's a class?

A different way of organizing a program

- Code
- Data

Instead of letting just *any* code read or change data, hide all the data for one "thing" behind functions that know how to work with it.

There's nothing big or scary about classes, but they *are* often helpful for:

- Making large programs easier to understand
- Making it easier to use code without understanding all the details

Word count without classes

In [5]:
```
def wc_initialize():
    return []

def wc_add_word(data, word):
    data.append(word)

def wc_get_count(data, word):
    count = 0
    for w in data:
        if w == word:
            count += 1

    return count

data = wc_initialize()
for word in 'the quick brown fox jumped over the lazy dog'.split(' '):
    wc_add_word(data, word)
print(wc_get_count(data, 'lazy'))
print(wc_get_count(data, 'the'))
```

```
1

2
```

Word count with classes

In [14]:
```
class WordCounter(object):
    def __init__(self):
        self.words = []

    def add(self, word):
        self.words.append(word)

    def get_count(self, word):
        count = 0
        for w in self.words:
            if w == word:
                count += 1

        return count

wc = WordCounter()
for word in 'the quick brown fox jumped over the lazy dog'.split(' '):
    wc.add(word)
print(wc.get_count('lazy'))
print(wc.get_count('the'))
```

```
1
2
```

Any questions?

How does this compare to the previous example?

- "wc" has both the data and functions.
- Less error prone

Python will still let you use "wc.words" directly, but you should avoid this.

Do you see anything undesirable about the way this class is written?

- Storing all words takes more memory
- Is slower to get the count

Hiding information helps when things change...

In [15]:
```
class WordCounter(object):
    def __init__(self):
        self.words = {}

    def add(self, word):
        if word not in self.words:
                self.words[word] = 0
            self.words[word] += 1

    def get_count(self, word):
        return self.words.get(word, 0)

wc = WordCounter()
for word in 'the quick brown fox jumped over the lazy dog'.split(' '):
    wc.add(word)
print(wc.get_count('lazy'))
print(wc.get_count('the'))
```

```
1
2
```

Notice:

- We're storing the information a completely different way.
- The code at the bottom doesn't change at all.

Which version is better?

Now we'll ignore case...

In [6]:

```
class WordCounter(object):
    def __init__(self):
        self.words = {}

    def add(self, word):
        word = word.lower()
        if word not in self.words:
            self.words[word] = 0
        self.words[word] += 1

wc = WordCount()
for word in 'The quick brown fox jumped over the lazy dog'.split(' '):
    wc.add(word)
print(wc.words)
```

```
{'brown': 1, 'lazy': 1, 'jumped': 1, 'over': 1, 'fox': 1, 'dog': 1,
'quick': 1, 'the': 2}
```

Aside: You've probably used some of Python's built-in classes already:

- string (str)
- list
- dictionary (dict)

- file

Built-in and app-defined classes work very similarly (in fact many of Python's library classes are written directly in Python).

Objects can contain other objects

In [12]:
```python
class Part(object):
    def __init__(self, description):
        self.description = description

class Tire(Part):
    def __init__(self, description, miles_warranty):
        Part.__init__(self, description)
        self.miles_warranty = miles_warranty

class Car(object):
    def __init__(self, vin):
        self.vin = vin
        self.steering_wheel = Part('steering wheel')
        self.tires = [Tire('tire #%d' % i, 30000) for i in range(4)]

car = Car('123')
print(car.steering_wheel.description)
for tire in car.tires:
    print('%s: %d' % (tire.description, tire.miles_warranty))
```

```
steering wheel
tire #0: 30000
tire #1: 30000
tire #2: 30000
tire #3: 30000
```

Discussion

Here are some phrases from "Python the Hard Way":

- class X(Y) : "Make a class named X that is-a Y."
- class X(object): def __init__(self, J) : "class X has-a __init__ that takes self and J parameters."
- class X(object): def M(self, J) : "class X has-a function named M that takes self and J parameters."
- foo = X() : "Set foo to an instance of class X."
- foo.M(J) : "From foo get the M function, and call it with parameters self, J."
- foo.K = Q : "From foo get the K attribute and set it to Q."

For each example phrase, change it so it talks about the Car or WordCounter example.

Discussion

For the following kinds of objects, what fields might you want? Should any of the fields be objects themselves?

- Elevator (running versus maintaining it)
- Person (address book versus hr)
- Phone call (phone company versus call center)
- Song (iTunes store versus Pandora)

Classes can be real-world things, but also computer-oriented things:

- File (on your machine versus Dropbox)
- User
- Photo

I showed two versions of WordCounter that looked the same on the outside but worked differently on the inside. This consistency makes it very easy for users if the program changes. The same thing is true in the real world:

- Cars: gas, electric, hybrid
- Phones: land line, cell

Discussion

Sometimes classes don't represent "things" at all. Maybe actions instead...

In [ ]:
```
class Plus(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def apply(self):
        return self.a + self.b

class Minus(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def apply(self):
        return self.a - self.b
```

This can be a very powerful, flexible way to organize a program with discrete "behaviors" you want to switch out. For example, in a calendar program when you choose to receive reminders by email or text message, it may use this same technique.

Coding Exercise

Complete the following example. "Running average" means that it accepts numbers one at a time and computes an updated average every time a number is added.

In [ ]:

```
class RunningAverage:
    def __init__(self):
        ...

    def insert(self, number):
       ...

    def get_average(self):
```