Python V

Modules and Classes

Modules

- These are simply Python files
- They can be imported into other Python files with the `import` keyword, can also use `as` for alias
- Can import contents with `from … import`

```
def apple():
    print "I AM APPLES!"

# this is just a variable
tangerine = "Living reflection of a dream"
```

Inside mystuff.py

```
import mystuff

mystuff.apple()
print mystuff.tangerine
```

Inside another Python file

```
mystuff['apple'] # get apple from dict
mystuff.apple() # get apple from the module
mystuff.tangerine # same thing, it's just a variable
```

Three ways of accessing stuff

Classes: creating one

`__init__` is a constructor

`self` is the `this` of Python

```
class MyStuff(object):

    def __init__(self):
        self.tangerine = "And now a thousand years between"

    def apple(self):
        print "I AM CLASSY APPLES!"
```

Creating and using classes

All instance methods of the class have to have the `self` keyword as the first argument

We don't pass in anything for `self` when calling the method

```
class Song(object):

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_a_song(self):
        for line in self.lyrics:
            print line

happy_bday = Song(["Happy birthday to you",
                   "I don't want to get sued",
                   "So I'll stop right there"])

bulls_on_parade = Song(["They rally around tha family",
                        "With pockets full of shells"])

happy_bday.sing_me_a_song()

bulls_on_parade.sing_me_a_song()
```

Class and instance attributes

Instance attributes are prefixed by self.

Class attributes are declared outside of a function

```
class Car(object):

    wheels = 4

    def __init__(self, make, model):
        self.make = make
        self.model = model

mustang = Car('Ford', 'Mustang')
print mustang.wheels
# 4
print Car.wheels
# 4
```

Static and class methods

- Static methods have no self argument: they belong to all instances
- Class methods are passed in the class
- “Decorators” (annotations) make things clearer

```
class Car(object):
    ...
    def make_car_sound():
        print 'VRoooooommmm!'
```

```
class Car(object):
    ...
    @staticmethod
    def make_car_sound():
        print 'VRoooooommmm!'
```

```
class Vehicle(object):
    ...
    @classmethod
    def is_motorcycle(cls):
        return cls.wheels == 2
```

Inheritance

Consider the following classes

(For the complete class, see classes.py link on class site)

```
class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership.

    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """
```

Inheritance Continued

These classes inherit from Vehicle, including all of its methods

```
class Car(Vehicle):

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Car object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 8000


class Truck(Vehicle):

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 10000
```

Abstract Base classes

Should there really be "vehicle" objects (as opposed to trucks and cars)?

Abstract Base Classes can't be instantiated

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership.


    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    __metaclass__ = ABCMeta

  @abstractmethod
  def vehicle_type():
      """Return a string representing the type of vehicle this is."""
      pass
```

See abstract_classes.py on website for whole code