What's next?

- Classes for this session and probably the next
- Back to UFO Data
  - Use what we learned with Classes to store our UFO Data in a DB
  - To do that we need to learn: **DJANGO**
- Or any project that anyone wants to work on

Why IPython?

The python command shell is very good, but does have a few shortcomings that are solved by IPython

- Command History -- Go back to previous commands
- Easy to use for presentations -- No longer need to watch me type
- Save support -- Can save you session history, review later, & pickup where you last left off
- Fully interactive -- Change previous code and re-execute
- Tab Completion -- Automatically complete commands using the 'tab' key

IPython

Instructions to Install and Run IPython

- pip install ipython
- pip install tornado
- easy_install pyzmq

You are welcome to install PyZMQ from source using PIP on Windows. Good luck!!!

Running IPython Notebook

C:\Python27\Scripts\Ipython notebook notebook_name.ipynb

An Introduction to Classes

What is a Class?

A class defines the abstract characteristics of a thing (object), including its characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something.

For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors).

Classes provide modularity and structure in an object-oriented computer program.

Three major benefits of classes:

- Easy to use collection of code with similar or related functionality
- Code reuse through inheritance
- Polymorphism

Trust me you need to know about classes

They are in every major programming language: Java, Ruby, C++, etc.

Example of an Easy to Use Collection

At iN DEMAND we handle lots of files. I've created a class that has a bunch of useful functions for files, e.g. get_checksum(), is_file_locked(), read_file_to_string(), encrypt_file()

It's very helpful to have all of these functions in one logical place.

In [ ]:
```
#I need to tell Python the path to my HelperUtils library
import sys
sys.path.append(r"C:\Users\mraposa.VCNYC\Documents\HelperUtils")
```

In [ ]:
```
from file_utils import File
example_file = File(r"C:\Users\mraposa.VCNYC\Documents\ufo_data.zip")
```

In [ ]:
```
example_file.is_file_locked()
```

In [ ]:
```
example_file.get_checksum()
```

In [ ]:
```
example_file.get_checksum(checksum="CRC32")
```

In [ ]:
```
example_file.touch()
```

The key takeaway is that all of these useful functions are all in one easy to use and easy to find location.

Example of Class Inheritance and Code Reuse

iN DEMAND also handles tons of XMLs. There are generic XMLs and two XMLs of a fixed structure: ADI.XML and CHANGE.XML.

The ADI.XML and CHANGE.XML are both XMLs and any function that applies to an XML will apply to them as well. An ADI.XML Class inherits functions from the XML Class.

An example of one of these functions is saveXML(). When we save the XML it shouldn't matter if we are saving an ADI.XML, CHANGE.XML, or any other XML. The same code should apply.

In [ ]:
```
from Xml2x import XML, CHANGE_XML, ADI_XML
xml = XML(r"C:\Users\mraposa.VCNYC\Documents\Session_6\ADI.XML")
adi_xml = ADI_XML(r"C:\Users\mraposa.VCNYC\Documents\Session_6\ADI.XML")
change_xml = CHANGE_XML(r"C:\Users\mraposa.VCNYC\Documents\Session_6\CHANG
```

Neither ADI_XML or CHANGE_XML have save_XML() defined. However, since they are both XMLs and inherit from the parent XML class, they get to use all the functions in the XML class.

In [ ]:
```
adi_xml.saveXML()
change_xml.saveXML()
```

But ADI_XML and CHANGE_XML are different XML classes and hence have their own independent functions which they do *not* share

In [ ]:
```
adi_xml.getAssetName()
```

In [ ]:
```
adi_xml.get_HDContent_value()
```

In [ ]:
```
change_xml.get_mso_ids()
```

The key take away is that we wrote the saveXML() function only once in the parent XML Class and can easily reuse that code elsewhere in child classes that inherit from the parent.

Another Example

My models.py for a new application I'm building to migrate our content distribution system. File delivery Class stores information in a DB.

But where's all the missing pieces:

- How is input data validated?
- How is data queried?
- How is new data saved to the table?
- How are database sessions handled? Setup? Authentication? Maintentance? Tear-down?
- Where are the SQL statements?

This seems like hundreds if not a thousand of lines of code. Where is the missing code?

Note: models.Model in the code

Django Models

Key takeaway: Two words in your code gives you 1000+ lines of code in functionality through inheritance. Or you can't easily\cheaply write a feature rich application without inheritance

Example of Polymorphism

Remain Calm

For a moment try to ignore the syntax and just read through the code

In [ ]:
```
class Animal:
    def __init__(self, name):     # Constructor of the class
        self.name = name
    def talk(self):               # Abstract method, defined by convention
        raise NotImplementedError("Subclass must implement abstract method

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Missy'),
           Cat('Mr. Mistoffelees'),
           Dog('Lassie')]

for animal in animals:
    print animal.name + ': ' + animal.talk()
```

Key takeaway: We called talk() on each animal and python **automatically** "knew" to call the correct talk() function for each class.

Without polymorphism we would need to detect the type of each class, e.g. "Dog" or "Cat" and then call the appropriate talk() function. Would require at least 10 lines of code, i.e. 5x more.

Your First Class

In [ ]:
```
class Dog(object):
    """
    This is an example of a Dog class

    Use this class to store interesting info about a Dog

    Call its methods to make the dog perform actions
```

```
    """
    name = None
    breed = None
```

In [ ]:
```
dog1 = Dog()
print dog1.name
```

In [ ]:
```
help(dog1)
```

We've instantiated a new Dog class object and assigned it to the variable dog1.

Now, we will assign values to the attributes of that object

In [ ]:
```
dog1.name = "Rex"
dog1.breed = "Husky"
```

In [ ]:
```
print dog1.breed, type(dog1)
```

A Note on Nomenclature

- *instantiated*: an instance of a class usually assigned to a variable
- *attribute*: internal variables assigned to the class
- *method*: internal functions assigned to the class

We can define new attributes on the instantiated class. Although this is rarely done.

In [ ]:
```
dog1.age = 10
```

In [ ]:
```
print dog1.age
```

Now we are going to create a new instance of Dog() and assign values to it's attributes

In [ ]:
```
dog2 = Dog()
dog2.name = "Bella"
dog2.breed = "Bijon"
```

In [ ]:
```
print dog1.name, dog2.name
```

Let's add some functions to our class. We are adding functionality to our existing class. Note: that users of our class will have one place to go to for all "Dog" related functions.

This meets our first criteria for classes:

> Easy to use collection of code with similar or related functionality

In [ ]:
```
class Dog(object):
    name = None
    breed = None

    def bark(self):
        print("BOW WOW")
```

In [ ]:
```
dog1 = Dog()
dog1.bark()
```

"self" refers to the instantiated object you created and not the class itself.

In [ ]:
```
class Dog(object):
    name = None
    breed = None

    def bark(self):
        print("BOW WOW")

    def say_your_name(self):
        # Note the use of self.name below. Calling the self, i.e. the temp
        # placeholder for the object
        print("Woof - {}".format(self.name))
```

"self" is a temporary placeholder that refers to the instantiated object

In [ ]:
```
dog1 = Dog()
dog1.name = 'Bella'
dog2 = Dog()
dog2.name = 'Rex'
dog1.say_your_name()
dog2.say_your_name()
```

We want to initialize the object variables when the object is instantiated.

Assigning them later can be confusing and doesn't allow any additional input validation

In [ ]:
```
class Dog(object):
    name = None
    breed = None

    def __init__(self, name, breed):
```

```
        self.name = name
        self.breed = breed

    def bark(self):
        print("BOW WOW")

    def say_your_name(self):
```

In [ ]:
```
dog1 = Dog("Bella", "Bijon")
dog1.say_your_name()
```

The **init** forces the user to initialize all parameters.

Default parameters can be used, e.g. **init**(self, name, breed="Husky")

In [ ]:
```
dog2 = Dog("Ruby")
```

We want to publish our class and don't want people playing around with the internal attributes.

We don't people to assign 'name' and 'breed' directly. We want them to use the Initialization method

Adding a '_' prefix tells python developers that this is a private variable and should not be set directly

In [ ]:
```
class Dog(object):
    _name = None
    _breed = None

    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    def bark(self):
        print("BOW WOW")

    def say_your_name(self):
        print("Woof - {}".format(self._name))
```

In [ ]:
```
dog1 = Dog("Bella", "Bijon")
dog1.say_your_name()
```

Functions of classes can also be hidden with the prefix underscore

In [ ]:
```
import random
```

```
class Dog(object):
    _name = None
    _breed = None
    age = 0

    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    # Hidden function -- should only be called from within the class
    def _get_next_bark(self):
        return random.choice(["BOW WOW", "WOOF", "Yap Yap Yap", "MEOW"])

    def bark(self):
        print(self._get_next_bark())

    def say_your_name(self):
        print("Woof - {}".format(self._name))

dog1 = Dog("Bella", "Bijon")
```

In [ ]:
```
dog1.bark()
dog1.bark()
```

When in doubt use private attributes and functions. You can change your private attributes and functions at anytime. However public atributes and functions are expected to remain fixed/constant.

Changing Attributes

What if you wanted to change an attribute after the object is instantiated?

For public attributes this is easy

In [ ]:
```
import random

class Dog(object):
    _name = None
    _breed = None
    age = 0

    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    def _get_next_bark(self):
        return random.choice(["BOW WOW", "WOOF", "MEOW"])

    def bark(self):
        print(self._get_next_bark())
```

```
    def say_your_name(self):
        print("Woof - {}".format(self._name))

dog1 = Dog("Bella", "Bijon")
```

In [ ]:
```
dog1.age = 5
```

But what if someone sets an illogical value. Sometimes we want to validate attributes as they are set.

For example, the code below is accepted, but makes no sense

In [ ]:
```
dog1.age = -1
```

In [ ]:
```
dog1.age = "Hello"
```

In JAVA, we would use setters and getters to set and retrieve attributes.

In [ ]:
```
class Dog(object):
    _name = None
    _breed = None
    _age = None

    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    def bark(self):
        print("BOW WOW")

    def say_your_name(self):
        print("Woof - {}".format(self._name))

    def get_age(self):
        return self._age

    def set_age(self, age):
        if age < 0:
            raise ValueError("Age must be greater than 0: {}".format(age))
        else:
            self._age = age
```

But now we need to change all our code to use set_age() and get_age(). This isn't Pythonic

In [ ]:
```
dog1 = Dog("Bella", "Bijon")
dog1.set_age(5)
print dog1.get_age()
```

Instead we can use the @property decorator to define a function that is an attribute to a class

In addition, we use an @attribute.setter decorator to decorate a function used to set our attribute

It's OK if you don't fully understand decorators at this point.

In [ ]:
```
class Dog(object):
    _name = None
    _breed = None

    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    def bark(self):
        print("BOW WOW")

    def say_your_name(self):
        print("Woof - {}".format(self._name))

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        if age < 0:
            raise ValueError("Age must be greater than 0: {}".format(age))
        else:
            self._age = age

dog1 = Dog("Bella", "Bijon")
```

Before getting an attribute it needs to exist first. I could have made 'age' a parameter in the **init** and forced it to be initialized when the class was instantiated.

In [ ]:
```
print dog1.age
```

In [ ]:
```
dog1.age = -1
```

In [ ]:
```
dog1.age = 5
print(dog1.age)
```

Inheritance

First let's define a base or parent class that other classes will inherit from.

We define a generic "Employee" class that has basic functions and attributes shared by all employees, e.g. First Name, Last Name, Salary, etc.

In [ ]:
```
class Employee(object):
    def __init__(self, first_name, last_name, salary):
        self._fname = first_name
        self._lname = last_name
        # We add some nice validations which we want to reuse elsewhere
        if salary < 0:
            raise ValueError("Salary must be a positive number: {}".format
        self._salary = salary

    def get_name(self):
        return "{} {}".format(self._fname, self._lname)

    def bi_weekly_wage(self):
        return self._salary / 26
```

Now we build a new class that inherits from the Employee class

In [ ]:
```
class Worker(Employee):
    def __init__(self, first_name, last_name, salary, manager):
        # Even though we are initializing a new worker class we still want
        # use the same initialization code that was used for Employee
        # i.e. we don't want to rewrite all the initialization code

        # this super function calls the __init__ method of the parent clas
        # We pass that __init__ the parameters necessary to complete the i
        super(Worker, self).__init__(first_name, last_name, salary)
        self._manager = manager

    def get_manager(self):
        return self._manager
```

Let's instantiate a new worker

In [ ]:
```
jblow = Worker("Joe", "Blow", 25000, "Susie BossLady")
```

We get to use all the functions of the Employee class, i.e. we get easy code reuse

In [ ]:
```
jblow.get_name()
```

In [ ]:
```
jblow.bi_weekly_wage()
```

We also get to define functions that are unique just to the Worker class

In [ ]:
```
jblow.get_manager()
```

Let's define a new class, Executive, that also inherits from Employee.

However, excutives will get a yearly_bonus and that bonus will be distributed as part of their normal bi-monthly check. Hence the Employee wage() has to be overriden for the Executive class

In [ ]:
```
class Executive(Employee):
    def __init__(self, first_name, last_name, salary, yearly_bonus):
        # Even though we are initializing a new worker class we still want
        # use the same initialization code that was used for Employee
        # i.e. we don't want to rewrite all the initialization code

        # this super function calls the __init__ method of the parent clas
        # We pass that __init__ the parameters necessary to complete the i
        super(Executive, self).__init__(first_name, last_name, salary)
        # This next step calls the setter below
        # We are using the validations there to initialize this attribute
        self.yearly_bonus = yearly_bonus

    @property
    def yearly_bonus(self):
        return self._yearly_bonus

    @yearly_bonus.setter
    def yearly_bonus(self, yearly_bonus):
        if yearly_bonus < 0:
            raise ValueError("Yearly Bonus must be greater than 0: {}".for
        elif yearly_bonus > 10000000:
            raise ValueError("WOAH!!! You don't work for Apple")
        else:
            self._yearly_bonus = yearly_bonus

    # Overriding the the wage function set in employee
    def bi_weekly_wage(self):
        # Our executives like their bonuses spread out over the year
        print("Calling from Executive Class")
        return (self._salary + self._yearly_bonus) / 26
```

In [ ]:
```
jbigshot = Executive("Joe", "BigShot", 100000, 50000000)
```

In [ ]:
```
jbigshot = Executive("Joe", "BigShot", 100000, 50000)
```

In [ ]:
```
jbigshot.get_name()
```

In [ ]:
```
jbigshot.yearly_bonus
```

In [ ]:
```
jbigshot.bi_weekly_wage()
```

Polymorphism

It's time to cut checks for our employees. We have four employees and want to get the wages for each employee

In [ ]:
```
jbigshot = Executive("Joe", "BigShot", 100000, 50000)
jblow = Worker("Joe", "Blow", 25000, "Susie BossLady")
sbosslady = Executive("Susie", "Bosslady", 75000, 25000)
jsmith = Worker("John", "Smith", 45000, "Joe Bigshot")

employees = [jbigshot, jblow, sbosslady, jsmith]
```

I've instantiated the four employees and added them to a list. I know want to loop through the list and get the wages for each employee.

In [ ]:
```
for employee in employees:
    print("Name: {}\t\tWage: {}".format(employee.get_name(), employee.bi_w
```

I've called the bi_weekly_wage() function and python has **automatically** called figured out the correct function to call for each employee object. This is polymorphism

Advanced Topics

We won't be covering these advanced topics but you are free to research them yourself.

- Static Methods: Use to define a function that you can execute *without* instantiating the class. Useful when you have a helper function but don't want to instantiate the entire class. Uses the @static_method decorator on a class function
- Multiple Inheritance: All our examples had classes inheriting from a single parent. However, classes can inherit from multiple parents and get the functions and attributes from all the parents. Watch for issues with attribute and function collisions
- Metaclasses: Classes used to describe other classes. Used to build a class dynamically.