Lecture 6: A step back
Last time

We introduced a “scripting language” known as Python -- We started with the barest of motivation in terms of historical context and then jumped right in, focusing mainly on your recipe task

We dealt mainly in Python’s ability to manipulate text, reserving a more detailed overview of its built-in data types for today -- We also examined its module framework for sharing code, and auditioned the built-in module for regular expressions
Today

We will finish the introduction we started last time, talking briefly about BeautifulSoup and about a networking module that lets us fetch data from various web services

Next, we are going to be a bit more formal about some of the basic data types in Python (numbers, strings, lists and dictionaries) -- We’ll go over things pretty completely, but leave it to your own study if it starts to feel tedious

We will also have a glimpse at control structures like if-statements, for- and while-loops and so on -- At very least we will see a small Python program and break it into pieces...
Python recap

Setting our expectations about Python, we have comments like the following from its creator

“In the future, we envision that computer programming will be taught in elementary school, just like reading, writing and arithmetic. We really mean computer programming -- not just computer use (which is already being taught).”

And we’ve read that the language is meant to emphasize simplicity, practicality and a certain degree of “obviousness” in its constructions
Python recap

In our discussion last time, we identified a few basic facts about the language that we’ll build on in today’s lecture

**Objects**: Everything in Python is an object, and we were introduced to some basic tools for examining them -- The built-in functions `type()`, `help()`, `dir()` and `str()` helped us identify the kind of object we have and its attributes

**Operator overloading**: Basic operators like “+” and “*” take on different meanings depending on the type of the operands

**Dynamic Typing**: Python does not force you to declare variables ahead of time or set their size; we’ll see that Python takes care of low-level memory allocation, sparing you the joy

**Modules**: The basic building blocks of larger programs in Python -- They also formalize the procedure by which we make use of other people’s code in Python (or share our own)
Objects in Python

Formally, all objects in Python have three things: An **identity**, a **type**, and a **value**

An object’s identity cannot change once it has been created and you can think of it as the object’s address in memory -- The `type()` function returns an integer representing an object’s identity

We have already discussed an object’s type, it also cannot change once the object is created -- The function `type()` is used to determine what kind of object we have

The value of an object might change depending on whether the object is mutable or immutable (Lists are mutable, strings are not, for example -- But we will have more to say about this later)

More on this as our lecture progresses...
Use cases

Python supports a number of different use cases (for lack of a better term) --
We have primarily used the interactive shell, typing commands that are then
interpreted and executed (this matched our model of interaction with the Unix
shell)

Rather than type commands on their own, we can also store them in a file --
There are a couple ways we can think about files of Python code (and the
distinction is pretty loose):

As a **script or single program**, a chunk of self-contained code that
executes (more or less) a single task; or

As a **module**, a collection of statements and definitions that are intended to
be used by other programs or other people

(Next time we will look at a simple programming mechanism that will let you
incorporate both behaviors in one file -- To be understandable, though, we need
to have introduced functions)
Use cases

On the next few slides, we show how a simple (simple minded) program to count words in the “text” portions of the recipes (as opposed to the markup or the headline, say) -- It is a standalone executable, able to be mixed with other Unix commands
cocteau@homework:~$ cp /data/text_examples/word_count.py .

cocteau@homework:~$ ls -al word_count.py
-rwlr-xr-x 1 cocteau cocteau 742 2010-10-13 16:49 word_count.py

cocteau@homework:~$ ./word_count.py /data/text_examples/1985/01/02/186946.sgml \  
   | sort -rn | head -20

 32 and
 21 of
 16 a
 14 in
 12 to
 11 the
 10 or
  9 chicken
  8 with
  8 finely
  7 potatoes
  7 chopped
  6 pieces
  6 parsley
  6 hot
  5 wine
  5 milk
  5 mashed
  5 food
  5 dish
cocteau@homework:~$ ./word_count.py /data/text_examples/1985/01/02/* \
    | sort -rn | head -20

243 the
157 and
130 of
98 a
95 to
89 in
63 with
46 is
39 or
31 as
30 oil
28 are
26 water
26 beans
25 for
25 add
21 until
21 minutes
20 that
20 it
cocteau@homework:~$ find /data/text_examples/1985 -type f | xargs ./word_count.py \
    | sort -rn | head -50

12028 the
8924 and
5629 a
5351 to
4837 of
4477 in
3057 with
2266 is
2162 or
2110 1
1804 for
1689 it
1589 2
1466 add
1306 are
1245 on
1242 about
1189 as
1133 be
1132 that
1121 until
1080 cup
991 into
977 minutes
957 12
948 pepper
914 from
907 salt
875 4
862 3
827 cook
810 at
802 tablespoons
cocteau@homework:~$ find /data/text_examples/198* -type f | xargs ./word_count.py \
  | sort -rn | head -50

101205 the
73090 and
44374 a
41396 to
36158 of
34912 in
27023 with
20428 1
19053 or
16716 is
15881 for
14636 2
12817 it
11956 add
10280 are
9893 on
9848 cup
9684 until
9506 minutes
9326 12
9157 be
8854 about
8755 as
8585 salt
8516 that
8474 pepper
8067 4
7798 3
7725 tablespoons
7707 into
7278 oil
7025 from
6906 butter
Aside: Interpreter directives

Note that the file `word_count.py` starts with `#!` -- This construction is known as an **interpreter directive** and is used in Unix (-like) systems as a way to specify an interpreter (the computer language, say) for the following lines in the file (here, the interpreter is Python and you can find its location on your system by typing the following)

```
% which python
```

We have not talked about shell scripting in this class, but if we assemble Unix commands into a file, **this directive will let us tell the operating system which flavor of shell to use** (Bourne Shell, C-shell, etc.)

In the case of Python (or Perl, say), this construction is used to **create stand-alone executable files that can be called like any other Unix command** -- the reference to Python follows the “magic number” 0x23 0x21 (hexadecimal for the ASCII characters `#!`)

The final flourish in making `word_count.py` was to ask that it be considered an executable file -- We do with with the following Unix command

```
% chmod +x word_count.py
```
Back to our recipes

Last time, we were using a single recipe file as a case study in Python’s basic structures -- We started by reading data in from a file, storing each recipe as a list of strings (each element in the list was another line in the file), and toward the end, started dividing the lines into words.

This kind of approach, however, completely ignores the fact that there is a certain degree of structure in these files -- Sure, it’s not structure that relates to the data, to the aspects of the recipes, but it does have a lot to do with the organization of the recipes as New York Times articles.

There are headline and byline “tags” as well as paragraph “tags” -- This markup would help a browser or some other presentation device exhibit our recipe as a document, introducing breaks between paragraphs and so on.
Creating a higher-level object

Rather than represent a recipe as a list of lines, of text strings, we could instead think about it as a document and think about it as a headline, a byline, and a main story broken up into paragraphs.

BeautifulSoup is a Python module by Leonard Richardson and is available at http://www.crummy.com/software/BeautifulSoup/ -- The module allows us to read a recipe (an SGML document), but maintains the document structure (the paragraph tags and so on).
BeautifulSoup

Here is how we work with BeautifulSoup -- The module supports two kinds of objects, one for HTML (BeautifulSoup) and one for XML (BeautifulStoneSoup), each buildable from a file object.

```python
>>> f = open("/data/text_examples/1987/01/04/898087.sgml")
>>> bs = BeautifulStoneSoup(f)
>>> type(bs)
<class 'BeautifulSoup.BeautifulStoneSoup'>
>>> # use the find() method to grab individual tags (takes the first if... # there are multiples)
... >>> byl = bs.find("byl")
... print byl
<byl>
By FLORENCE FABRICANT
</byl>
>>> hdl = bs.find("hdl")
>>> print hdl
<hdl>
FOOD;
REVISING A RECIPE FOR VARIED INGREDIENTS
</hdl>
>>> hdl.getText()
u'FOOD;
REVISING A RECIPE FOR VARIED INGREDIENTS'
>>> hdl.getText().replace("\n"," ")
u'FOOD; REVISING A RECIPE FOR VARIED INGREDIENTS'
>>> hdl.getText().replace("\n"," ").strip()
u'FOOD; REVISING A RECIPE FOR VARIED INGREDIENTS'
```
BeautifulSoup

Once we have a BeautifulSoup object, in this case, we can now access the different tags that constitute our document -- Here we look for the headline and byline; and extract the text (string) associated with the tag

```python
>>> f = open("/data/text_examples/1987/01/04/898087.sgml")
>>> bs = BeautifulSoup(f)
>>> type(bs)
<class 'BeautifulSoup.BeautifulStoneSoup'>
>>> # use the find() method to grab individual tags (takes the first if ...
# there are multiples)
...
>>> byl = bs.find("byl")
>>> print byl
<byl>
By FLORENCE FABRICANT
</byl>
>>> hdl = bs.find("hdl")
>>> print hdl
<hdl>
FOOD;
REVISING A RECIPE FOR VARIED INGREDIENTS
</hdl>
>>> hdl.getText()
u'FOOD;\nREVISING A RECIPE FOR VARIED INGREDIENTS'
>>> hdl.getText().replace("\n"," 
")
u'FOOD; REVISING A RECIPE FOR VARIED INGREDIENTS'
>>> hdl.getText().replace("\n"," ").strip()
u'FOOD; REVISING A RECIPE FOR VARIED INGREDIENTS'
```
BeautifulSoup

Our recipe file has a single <hdl> and <byl> tag, but has multiple <p> paragraph tags -- We can use another method, findAll() to create a list of tag objects.

```python
>>> # the findAll method can be used to find multiple instances of a tag
>>> # like, say, <p> for paragraph
... ps = bs.findAll("p")
>>> type(ps)
<type 'list'>
>>> len(ps)
28
>>> print ps[0]
<p>
KNOWING how to vary a recipe, to take off on one's own from what is printed on the page, makes for inventive cooking. Some recipes are especially suited to revising. Here's a case in point.
</p>
>>> print ps[1]
<p>
Start with chicken cutlets, the boned and skinned breast halves that are sold in supermarkets. (Or you can start with whole breasts and bone and skin your own.) The recipe that calls for stuffing them, then cooking them in sauce can be varied almost infinitely.
</p>
>>> print ps[-1]
<p>
Yield: 4 servings.
</p>
```
Networking

Finally, we will demonstrate the wealth of data resources that are available to you via a language like Python -- As we mentioned earlier, scripting languages are often “glue” for larger systems and they typically need high-level abstractions for networking

Here we'll use the module urllib2 to fetch some data from the web...
Recipes on the web

As you might anticipate, recipes are a big deal, and there are plenty of services on the web that help you find a recipe based on the contents of your refrigerator, based on your skill level or perhaps based on the recommendation of some celebrity.

This weekend, in fact, the entire New York Times Magazine was devoted to food, with an article about http://www.food52.com, a community-based recipe site -- To keep things simple, we'll look at a lower-profile site, but one that makes its data more easily available...
Recipe Puppy is an ingredient based recipe search engine.
Enter your ingredients above or try an example search:

Potato Pancake Recipes with onions and eggs but without green onions

Already know what you want to make?

Use CookThing Now

Have a Cooking Question?

Ask Recipe Labs
Search by Ingredients (comma separated):
white wine, butter,

---

**Butter Sauce Recipes** [LandOLakesFoodservice.com/Recipes](http://www.landolakesfoodservice.com/recipes)
Quality ingredients to make butter sauces. Great flavor in every dish.

**Seafood Pasta Recipes** [deliciousrecipeideas.com](http://www.deliciousrecipeideas.com)
Over 100 Seafood Pasta Recipes Quick & Easy Seafood Pasta Recipes

**Making Shrimp & Pasta?** [healthyha.newworldpasta.com](http://www.healthyha.newworldpasta.com)
Get Recipe for Shrimp & Pasta. Plus $1OFF Ronzoni Healthy Harvest®.

**Free Granola Bars** [www.Facebook.com/NaturesPath](http://www.facebook.com/naturespath)
Visit Nature's Path and receive a Free Granola Bar Sample!

---

**Stuffed Mussels With White Wine Recipe**
Tasty recipe to go with a nice dry white wine. I love mussels and was tired or just making them in a wine sauce. Something different. Prep:15m
white wine, butter, +garlic, +eggs, +mussels, +bread crumbs, +salt, +rosemary, +parsley, +saffron, +paprika, +tomato
[www.grouprecipes.com](http://www.grouprecipes.com) - Similar recipes

---

**Scallops with White Wine Sauce II**
"White wine, butter, and shallots make a great sauce for scallops. This is easy and non-creamy for those that don't like cream sauces."
butter, white wine, +chicken broth, +garlic, +lemon, +olive oil, +salt, +sea scallops,
Recipe Puppy API

Recipe Puppy has a very simple API. This api lets you search through recipe puppy database of over a million recipes by keyword and/or by search query. We only ask that you link back to Recipe Puppy and let me know if you are going to perform more than 1,000 requests a day.

The api is accessible at http://www.recipepuppy.com/api/.

For example:
http://www.recipepuppy.com/api/?i=onions,garlic&q=omelet&p=3

Optional Parameters:
i : comma delimited ingredients
q : normal search query
p : page
format=xml : if you want xml instead of json

No parameters are required. Let me know if you have any questions or if you want to share the project built on top of our api.
In our first lecture, we mentioned the proliferation of APIs (application programming interfaces) making data available as a kind of web service -- The Recipe Puppy API is just one of many you will find, but it was somehow the simplest and hence a reasonable choice to play with on pedagogical grounds.

Aside from complications with authentication (often organizations want to know who is accessing their data) the broad principle is largely the same -- A specialized URL represents data, the results of a search, say, and not necessarily an HTML page.

For example, enter these two into a Firefox browser -- What do you get?

http://www.recipepuppy.com/api/?i=chicken,spinach
http://www.recipepuppy.com/api/?i=chicken,spinach&format=xml
Accessing data in Python

As we mentioned last time, Python is often the glue that binds pieces of a system together -- It has high-level constructions for accessing resources on the web, say

On the next slide, we use the urllib2 module to create and issue an HTTP request for a URL from Recipe Puppy -- This construction would allow us to, for example, change the UserAgent field, manage cookies and so on
```python
>>> import urllib2

>>> url = "http://www.recipepuppy.com/api/?i=chicken,spinach&format=xml&page=2"
>>> req = urllib2.Request(url)  # we can add headers here, set the useragent, etc

>>> response = urllib2.urlopen(req).read()
>>> bs = BeautifulSoup(response)
>>> print bs.prettify()
<recipes>
  <recipe>
    <title>
      Chicken With Spinach &amp;amp; Cheese Recipe
    </title>
    <href>
      http://cookeatshare.com/recipes/chicken-with-spinach-cheese-34127
    </href>
    <ingredients>
      chicken, spinach, mozzarella cheese
    </ingredients>
  </recipe>
  <recipe>
    <title>
      Spinach Burgers
    </title>
    <href>
      http://www.recipezaar.com/Spinach-Burgers-263874
    </href>
    <ingredients>
      bread crumbs, spinach, garlic, chicken
    </ingredients>
  </recipe>
  <recipe>
    <title>
      Simply Cordon Bleu Recipe
    </title>
    <href>
      http://www.grouprecipes.com/22/simply-cordon-bleu.html
    </href>
</recipes>
```
Two great tastes...

And once we have an XML document as a response, we can use BeautifulStoneSoup again to build a document object for us, that will let us do things like, extract all the titles or search the ingredient lists for specific items...
```python
>>> titles = bs.findAll("title")

>>> for t in titles: print t.getText()

... 
Chicken With Spinach &amp; Cheese Recipe
Spinach Burgers
Simply Cordon Bleu Recipe
Chicken Spinach Casserole Recipe
chicken rolls
VELVEETA Chicken Florentine
Chicken Florentine
Italian Chicken &amp; Spinach Salad
Rustic Spinach Salad
Broiled Chipotle Chicken With Creamy Spinach

>>> ingredients = bs.findAll("ingredients")

>>> for ing in ingredients: print ing.getText()

... 
chicken, spinach, mozzarella cheese
bread crumbs, spinach, garlic, chicken
brie cheese, chicken, olive oil, spinach
chicken, spinach, cheddar cheese, bacon
chicken, spinach, cream cheese, salt
italian dressing, chicken, spinach, velveeta cheese, milk
italian dressing, spinach, chicken, smoked ham, swiss cheese
spinach, chicken, broccoli, cherry tomato, italian dressing
spinach, apple, chicken, mozzarella cheese, italian dressing
chicken, chicken broth, chilies, salt, spinach
```
To sum

We’ve had a first look at Python -- We’ve seen it to be an **object oriented language**, a powerful data model that lets us easily move from lines of text to structured documents with very little effort

Python is built on a module-based system of code sharing, and a large community of developers has been steadily adding new functionality to the system -- Again, consider the power of BeautifulSoup

We have also seen **the ease with which we can access data** on our local filesystem as well as via web services **in “the cloud”**

Now we’re going to take a step back and deal with some of the formalities of the language, examining the built-in data types, walking through a more complete treatment of modules and precisely what happens when you execute a Python program... and so on
The basics

Next we'll cover a few of the **basic data types** (built-in object types) in Python; these tend to be pretty efficient and are the building blocks from which we extend the language.

We will start with the following

1. Numbers
2. Strings
3. Files
4. Lists and dictionaries
Numbers

Python knows about several different numeric types; integers, floating-point numbers, complex numbers -- The different types all imply different operations for addition, multiplication, and so on

When mixing types, Python will convert the separate operands up to the type of the most complicated operand and then performs its math on the common type; integers are simpler than floats, for example, and floats are simpler than complex numbers
Numbers

Operator precedence is settled as you would expect from any other language you have seen previously: simple operators like * and + are evaluated first, from left to right, with logical expressions coming last (see Lutz for a complete list).

Where there are questions of precedence, use parentheses to force processing in the order you want; Python evaluates expressions in parentheses first and supplies the result to enclosing expressions.
Numbers

First, let's have a look at how the different numeric types are written in Python -- with this in hand, we can use the interactive session like a big calculator; you can write expressions using basic operators, being mindful of the numeric types involved (mixed types covert “up”)

```python
>>> x = 3       # an integer
>>> type(x)
<type 'int'>

>>> x = 3.5     # a floating point number
>>> type(x)
<type 'float'>

>>> x = 3+5j    # a complex number
>>> type(x)
<type 'complex'>

>>> 3.0/10.0    # ordinary division
0.3

>>> 3.0/10      # the integer 10 is converted to a float first
0.3

>>> 20/3        # since 20 and 3 are integers, integer division is performed
6

>>> (10+12)/3.0
7.333333333333333
```
Numbers

Python has some built-ins like `pow` and `abs`, and there is a module that contains a number of common mathematical functions -- there is also a module for generating observations from a few of the more common probability distributions.

```python
>>> abs(-203.5)  # a built-in function
203.5

>>> import math  # a module with a number of other math functions
>>> dir(math)

>>> math.asin(math.sqrt(0.4))
0.684719203002283

>>> help(math)
```
Help on module math:

NAME
   math

FILE
   /usr/local/lib/python2.7/lib-dynload/math.so

MODULE DOCS
   http://docs.python.org/library/math

DESCRIPTION
   This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS
   acos(...)
      acos(x)

      Return the arc cosine (measured in radians) of x.

   acosh(...)
      acosh(x)
Numbers

Python has some built-ins like `pow` and `abs`, and there is a module that contains a number of common mathematical functions -- there is also a module for generating observations from a few of the more common probability distributions.

```python
>>> import random  # a module with random number generators

>>> x = random.expovariate(3)  # a single exponential observation
>>> type(x)
<type 'float'>

>>> y = [random.expovariate(3) for i in range(100)]  # a teaser!
>>> type(y)
<type 'list'>

>>> len(y)
100

>>> y[0]
0.8412092191136957

>>> y[1]
0.2853034767434156
```
Assignments

Notice on the last slide that $x$ was an integer, then a float and in our last session it was a string; when it comes to assigning variables, Python employs something called **dynamic typing**

A variable is created when it is first assigned a value by your code; for example, consider what happens when we type $x = 3$

1. First, Python **creates an “object”** to represent the value 3 (the type of this object is `int`)

2. Then Python creates a variable $x$ if it does not exist; a variable is just an entry in a search table, **a name**

3. Finally, Python associates the name $x$, the variable, with the object that it refers to; **it adds a link to the object in the search table**
>>> 3  # an object representing the number 3
3

>>> type(3)  # its type
<type 'int'>

>>> id(3)  # ... and identity
36771720

>>> x = 3  # take the name x and associate it with the object 3

>>> type(x)  # its type
<type 'int'>

>>> id(x)  # ... and identity
36771720

>>> y = x  # the name y is now associated with the same object as x

>>> y
3

>>> type(y)  # its type
<type 'int'>

>>> id(y)  # ... and identity
36771720

>>> y = 5  # now create a new object 5 and assign the name y to it

>>> type(y)  # its type
<type 'int'>

>>> id(y)  # ... and identity
36771672

>>> x  # this leaves x unchanged
3

>>> type(x)  # its type
<type 'int'>

>>> id(x)  # ... and identity
36771720
Assignments

We’ll say a bit more about assignments in a moment -- These mechanics seem pretty clear, but you need to be a little careful when you work with mutable objects like lists

More on that shortly -- For now, let’s review valid variable names...
Variable names

This is as good a place as any to comment on what names are open to you; there are a few rules

1. Variable names can start with an underscore or a letter and can be followed by any number of letters, digits or underscores

2. Python is case-sensitive, meaning x and X are two different things

3. Finally, you can’t choose one of the so-called reserved words, names that mean other things in the language

    and     del     from     not     while
    as      elif    global   or      with
assert    else    if      pass     yield
break    except  import   print
class    exec    in       raise
continue finally is     return
def      for     lambda  try
Other tools

Python actually has a fair bit of support for numerical calculations.

There is also a separate module called NumPy that provides access to the basic computations from linear algebra, advanced random number capabilities, and even the fast Fourier transform.

It will be interesting to compare Python+(sundry numerical extensions) with R, a language specifically...
Heavy lifting

NumPy is an attempt to bring tools from what we classically refer to as “scientific computing” to Python

There are several such projects, some falling under the general title of SciPy

As an upcoming homework, you might get a chance to play with one or more of these...
Heavy lifting

NumPy is an attempt to bring tools from what we classically refer to as “scientific computing” to Python.

There are several such projects, some falling under the general title of SciPy.

As an upcoming homework, you might get a chance to play with one or more of these...
String literals

Strings are defined with single-, double- or triple-quoted sequences of characters.

Within quotations of any variety, we can embed escape sequences that encode special byte sequences representing, for example, new lines (‘one \ntwo’); in Python, we can turn off the escape with another backslash (‘one\\ntwo’) or with a prefix r for “raw” (r’one\ntwo’).

Operators like + and * have well-defined meanings for string objects; the interpreter is smart enough to apply them where appropriate (note that we get an error if we try to mix types here).

Strings are just an ordered collection of characters, and you can access portions of them via indexing and slicing.
**Strings**

Recall that in the shell, single quotes were a kind of just-the-facts string (no funny expansions); in Python, we can turn off the escape (escape the escape?) by preceding things with an `r` for “raw”

```python
>>> filename = "C:\new\data.txt"  # a problem lurking
>>> print filename
C:
ew\data.txt

>>> filename = r"C:\new\data.txt"  # turn off the escape mechanism
>>> print filename
C:\new\data.txt
```
Strings

Finally, we can have strings spill over to multiple lines either by using single- or double-quotes and ending each line with a “\” or by using triple- (yes, triple) quotes

```python
>>> x = "one\
... two\
... three\
... four"

>>> x
'onetwothreefour'

>>> x = """one
... two
... three
... four"

>>> x
'onenettwonethree\nfour'

>>> print x
one
two
three
four
Strings

Operators like + and * have well-defined meanings for string objects; the interpreter is smart enough to apply them where appropriate (note that we get an error if we try to mix types here)

```python
>>> x = "one"
>>> y = "two"

>>> x+y
"onetwo"

>>> x*10
"oneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneoneone
Operator overloading

You can “overload” Python’s operators to deal with different types of objects; operators like “+” and “*”, for example, have been extended to handle many of the built-in types.
Strings

Strings are just an ordered collection of characters, and you can access portions of them via indexing and “slicing”

```python
>>> brown = "I've been in the kitchen, I've taken the heat."
    >>> brown[0]
    'I'
    >>> brown[1]
    ':'
    >>> brown[1:3]
    '''
    >>> brown[:6]
    "I've b"
    >>> brown[-7:]
    'e heat.'
```
Strings

Python has also been good enough to provide you with several functions to make dealing with strings a little easier -- These are all attributes of the string type and allow you to replace portions of the string, translate characters to upper case and so on

As we saw last time, Python also offers incredible regular expression support through the re module -- Collectively, all of these capabilities will server you well in your hunt for ingredients and instructions
Lists

Lists are built-in object types that represent collections of objects; much of the syntax for working with lists is the same as that for strings (but lists hold more general objects than characters)

1. Ordered collections of arbitrary objects

2. Indexed by position

3. Have variable length and can be nested

4. Stores references to objects and not actual objects
Lists

On the next slide, we review how to make lists manually (bracketing a comma-separated list of objects with [ ]) and how operator overloading works for multiplication and addition of lists.

We will also illustrate some convenience functions that allow us to add and delete elements of a list, sort the list and so on.
# manually create a list, bracketing with []'s

```python
>>> x = [1, 2, "eggs", "butter", 5+3j]
```

```python
>>> type(x)
<type 'list'>
```

```python
>>> x
[1, 2, 'eggs', 'butter', (5+3j)]
```

# lists can be arbitrarily nested...

```python
>>> x = [1, 2, "eggs", "butter", 5+3j, [3, 4, 5]]
```

# operator overloading again... first multiplication

```python
>>> 2*x
[1, 2, 'eggs', 'butter', (5+3j), 1, 2, 'eggs', 'butter', (5+3j)]
```

# ... and then addition

```python
>>> y = ["salt", "pepper"]
>>> x+y
[1, 2, 'eggs', 'butter', (5+3j), 'salt', 'pepper']
```
# make an empty list

```python
>>> x = []
```

```python
>>> type(x)
<type 'list'>
```

# examine the attributes associated with it...

```python
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

```python
>>> help(x)
```

```python
>>> x.append(1)
```

# append a value to the end of the list

```python
>>> x.append("Brown")
>>> x.append(5)
>>> x.append("Whitman")
```

```python
>>> x
[1, 'Brown', 5, 'Whitman']
```
# make an empty list
>>> x = []
>>> type(x)
<type 'list'>

# examine the attributes associated with it...
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'

>>> help(x)

# append a value to the end of the list
>>> x.append(1)

# append a value
>>> x.append("Brown")

# append a value
>>> x.append(5)

# append a value
>>> x.append("Whitman")

>>> x
[1, 'Brown', 5, 'Whitman']
# make an empty list

```python
tax = []
type(x)
```

# examine the attributes associated with it...

```python
dir(x)
help(x)
```
>>> x
[1, 'Brown', 5, 'Whitman']

>>> len(x)
4

# insert an item before index 2 -- we start counting from 0!
>>> x.insert(2, "Nicandra Diaz Santillan")
>>> x
[1, 'Brown', 'Nicandra Diaz Santillan', 5, 'Whitman']

# and remove an item explicitly...
>>> x.remove("Brown")
>>> x
[1, 'Nicandra Diaz Santillan', 5, 'Whitman']

# ... or by index; here we remove and return the item at index 3
>>> z = x.pop(3)
>>> z
'Whitman'

>>> x
[1, 'Nicandra Diaz Santillan', 5]

# sort the list alphabetically
>>> x.sort()
>>> x
[1, 5, 'Nicandra Diaz Santillan']
>>> comment = "Perhaps it is nothing more than unbridled enthusiasm for globe artichokes"
>>> words = comment.split(" ")
>>> words
['Perhaps', 'it', 'is', 'nothing', 'more', 'than', 'unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

# default sort is alphabetically...

>>> words.sort()

>>> words
['Perhaps', 'artichokes', 'enthusiasm', 'for', 'globe', 'is', 'it', 'more', 'nothing', 'than', 'unbridled']

# make a new function that can be used to sort a list -- here
# we give an example in which we sort by the length of the string

>>> def len_compare(x,y):
    if len(x)>len(y): return 1
    elif len(x)==len(y): return 0
    else: return -1  # this means x < y

# there's a lot here -- we defined a function called len_compare
# that takes two arguments; it's body is the indented code; and it
# returns 1 if string x is longer than string y, 0 if they have the
# same length, and -1 if y is longer than x

>>> words.sort(len_compare)

>>> words
['is', 'it', 'for', 'more', 'than', 'globe', 'Perhaps', 'nothing', 'unbridled', 'artichokes', 'enthusiasm']
Assignment (again)

A couple of concepts came up when we started to make assignments with lists; in particular we needed to distinguish between what you can do with a list (a mutable sequence) and what you can do with a string (an immutable sequence).

We can select pieces of both strings and lists (subsetting and slicing) but we can only change the value of the elements of a list; we cannot change the elements of a string “in place”
>>> comment
'Perhaps it is nothing more than unbridled enthusiasm for globe artichokes'

>>> comment[4]
a
>>> comment[4:10]
'aps it'

# try assignment "in place"
>>> comment[4] = "g"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

# now create a list...
>>> words = comment.split(" ")

>>> words
['Perhaps', 'it', 'is', 'nothing', 'more', 'than', 'unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

>>> words[4]
'more'

>>> words[4:10]
['more', 'than', 'unbridled', 'enthusiasm', 'for', 'globe']

# try making an assignment in place...
>>> words[7] = "disdain"

>>> words
['Perhaps', 'it', 'is', 'nothing', 'more', 'than', 'disdain', 'for', 'globe', 'artichokes']

# aside: what is this doing???

>>> " ".join(words)
'Perhaps it is nothing more than unbridled disdain for globe artichokes'
Immutable versus mutable objects

Mutable objects are those that can be changed once they are created (these include lists and dictionaries); and immutable objects cannot be changed after they’ve been created (these include strings and numbers) -- Ultimately, mutability is a function of an object’s type

Python offers several “container” classes that hold “references” to other objects; some of these containers are mutable (lists and dictionaries) and some are not (tuples) -- Note: The concept of mutability extends to the “identities” of the contained objects
# a tuple is an immutable container...

```python
>>> somewords = ('unbridled', 'enthusiasm', 'for', 'globe', 'artichokes')
```

```python
>>> somewords
('unbridled', 'enthusiasm', 'for', 'globe', 'artichokes')
```

```python
>>> type(somewords)
<type 'tuple'>
```

# we can subset or slice a tuple

```python
>>> somewords[3:5]
('globe', 'artichokes')
```

# but we can’t change it “in place”

```python
>>> somewords[1] = "disdain"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

# and sorting is not supported

```python
>>> somewords.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
```

# and the same goes for appending or deleting from the tuple

```python
>>> somewords.append("to")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```
Assignments and mutability

In Python, the mutability of an object determines how things like assignments take place -- **Immutable objects are copied** for you when you start to work with them, making their behavior similar to what you saw in R.

When dealing with mutable objects, on the other hand, we are really working with a reference to the underlying object -- This means that when we change the underlying object, **all the variables that point to the same object have had their values changed as well**.

Let's see what this means for the simple case of a list, a mutable object...
# recall the issues with assignment...

```python
>>> somewords1 = ['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
>>> somewords2 = somewords1
>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

# somewords1 and somewords2 now “reference” or point to the same object; therefore
# making a change affects both...

>>> id(somewords1)
140095094341856
>>> id(somewords2)
140095094341856

>>> somewords1[1] = "disdain"
>>> somewords1
['unbridled', 'disdain', 'for', 'globe', 'artichokes']
>>> somewords2
['unbridled', 'disdain', 'for', 'globe', 'artichokes']
```
# we can force (nearly) R-like copying using an “empty” slice

```python
>>> somewords1 = ['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
>>> somewords2 = somewords1[:]

>>> somewords1
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

# now when you make a change, it applies to just one...

>>> somewords1[1] = "not"

>>> somewords1
['unbridled', 'disdain', 'for', 'globe', 'artichokes']

>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
```
# now, make a list that contains a list

```python
>>> somewords = ['unbridled', 'enthusiasm']
>>> words1 = ['globe', 'artichokes', somewords]
```

# and copy words1 into words2 using an empty slice

```python
>>> words2 = words1[:]
```

# now, make some changes...

```python
>>> words1[0] = 'small'
>>> words1
['small', 'artichokes', ['unbridled', 'enthusiasm']]
>>> words2
['globe', 'artichokes', ['unbridled', 'enthusiasm']]
```

# no difference. but if we change the list contained in words1...

```python
>>> words1[2][1]
'enthusiasm'
>>> words1[2][1] = 'disdain'
>>> words1
['small', 'artichokes', ['unbridled', 'disdain']]
>>> words2
['globe', 'artichokes', ['unbridled', 'disdain']]
```

```python
>>> somewords
['unbridled', 'disdain']
```

# the moral: an empty slice only makes a top-level copy; references to mutable objects are maintained
Copies

It's important to understand what's going on when you make assignments and copies; these are subtle issues that will come up (somewhat inevitably) in the form of hard-to-track-down bugs

If you want to make something more than just a top-level copy of an object, you can use the copy module...
Something new: Dictionaries

Dictionaries are like lists except that you index them by names instead of by numbers; in more technical parlance, **you access data with keys**

In the example that follows, we will be using strings as keys, but you can also use integers or any other immutable object
# create a new dictionary
```python
>>> x = {"Whitman":"It will be the same old same old.", "Brown":"I’ve taken the heat."}

>>> type(x)
<type 'dict'>
```

# see its attributes...
```python
>>> dir(x)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

```python
>>> help(x)
```

# extract the "keys" from the dictionary
```python
>>> x.keys()
['Whitman', 'Brown']
```

# or ask if the dictionary contains a key using a dictionary method...
```python
>>> x.has_key("Brown")
True
```
```python
>>> x.has_key("Browkaw")
False
```
```python
# ... or an operator
```
```python
>>> "Brown" in x
True
```
# create a new dictionary
>>> x = {"Whitman":"It will be the same old same old.","Brown":"I’ve taken the heat."

>>> type(x)
<type 'dict'>

# see its attributes...

>>> dir(x)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__method__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

>>> help(x)

# extract the "keys" from the dictionary

>>> x.keys()
['Whitman', 'Brown']

# or ask if the dictionary contains a key using a dictionary method...

>>> x.has_key("Brown")
True

>>> x.has_key("Browkaw")
False

# ... or an operator

>>> "Brown" in x
True
# create a new dictionary

```python
>>> x = {"Whitman":"It will be the same old same old.","Brown":"I’ve taken the heat."
```nn
```python
>>> type(x)
<type 'dict'>
```nn

# see its attributes...

```python
>>> dir(x)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```nn

```python
>>> help(x)
```nn

# extract the "keys" from the dictionary

```python
>>> x.keys()
['Whitman', 'Brown']
```nn

# or ask if the dictionary contains a key using a dictionary method...

```python
>>> x.has_key("Brown")
True
```nn
```python
>>> x.has_key("Browkaw")
False
```nn

# ... or an operator

```python
>>> "Brown" in x
True
```
# dictionaries are mutable so they can be changed “in place”

```python
>>> x["Brown"] = [1,2,3]
>>> x
{'Brown': [1, 2, 3], 'Whitman': 'It will be the same old same old.'}

# you can delete items...
>>> del x["Obama"]
>>> x
{'Whitman': 'It will be the same old same old.'}

# ... or add them
>>> x["Brokaw"] = "NBC"
>>> x
{'Brokaw': 'NBC', 'Whitman': 'It will be the same old same old.'}

# we can also extract just the values (returned in a list)
>>> x.values()
['NBC', 'It will be the same old same old.'](1)

# or the separate key-value pairs (again in a list... this time a list
# of tuples...
>>> x.items()
[('Brokaw', 'NBC'), ('Whitman', 'It will be the same old same old.')]

# a convenience function for extracting values
>>> x.get("Brokaw")
'NBC'

>>> x.get("Couric")  # returns None type if there is no such key
>>> x.get("Couric","not here")  # or some value you prescribe
'not here'
```
# finally, building up a dictionary in pieces...

```python
define x = {}
define x[1] = {}
define x
define x[1][2] = "Brown"
define x[1][5] = "Whitman"
define x
define x[10] = {}
define x
define x[10]['quote'] = 'It will be the same old same old.'
define x
define del x[10]
define x
define x[1][5] =
define x
```

```
{1: {2: 'Brown', 5: 'Whitman'}}
{1: {2: 'Brown', 5: 'Whitman'}, 10: {}}
{1: {2: 'Brown', 5: 'Whitman'}, 10: {'quote': 'It will be the same old same old.'}}
{1: {2: 'Brown', 5: 'Whitman'}}
{1: {2: 'Brown'}}
```
Now we’re in business...

We’re now in good shape to at least read through the code we introduced at the beginning of the session -- The word_count.py function is a little crude but it exercises almost all of the data types we’ve seen so far.

It also introduces some “control of flow’ constructions that we’ll go over in the next couple of lectures -- You have enough at this point, however, to at least read what’s going on.

There is one small thing more we have to say though...
Indentation

Whereas R and many other languages use { }'s to bracket off blocks of code, Python relies on indentation -- in general, the first block of code must begin in the first column.

Consecutive blocks have to be indented; the amount of indentation is up to you, but it has to be consistent.

Aside from indentation, blank spaces are usually ignored in Python statements.
#!/usr/local/bin/python

# counts words from a list of (sgml) recipe files
# counts only words in <p> tags

import sys, re
from BeautifulSoup import BeautifulStoneSoup

recipe_files = sys.argv[1:]
word_totals = {}

for recipe_file in recipe_files:
    bs = BeautifulStoneSoup(open(recipe_file))

    for p in bs.findAll("p"): 
        line = p.getText()
        line = re.sub("\s+"," ",line)
        line = line.strip()

        words = line.split(" ")
        words = [w.lower() for w in words]
        words = [re.sub("\W","",w) for w in words]
        words = [w for w in words if w]

        for w in words:
            if not w in word_totals: word_totals[w] = 0
            word_totals[w] += 1

for w in word_totals: print word_totals[w],w
A little more structure: Control-of-flow

To be seriously dangerous, we need a couple basic constructions in Python; essentially these **direct the flow of operation of our programs**

So far (except for the for-loops from last lecture) we present Python a series of expressions and it executes them all, in order; **sometimes we may want to act conditionally** or we may want to repeat steps some number of times...
if-elif-else statements

Before we can talk meaningfully about if statements, we need to introduce one last built-in type a Boolean value; as you might expect this takes on the values True and False.

You will encounter them when you consider the sizes of numbers or containment in strings and lists; you combine them using so-called Boolean operators.

For convenience, any object can be used to create a Boolean value; nonzero numbers or a non-empty objects are True (why would this be helpful?)
Python 2.5.1 (r251:54863, Jun 17 2009, 20:37:34)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> x = bool(1)
>>> type(x)
<type 'bool'>
>>> x
True
>>> y = bool(0)
>>> y
False
>>> x = bool(['a','b'])
>>> x
True
>>> y = bool([])
>>> y
False

>>> x = 5
>>> y = 7

>>> z = x>3 and y==7

>>> z
True
>>> type(z)
<type 'bool'>

>>> x<10 or y>10 and not x+y<3

True

>>> x and []        # Boolean operators return objects that "evaluate"
[]
>>> {} and y        # to true or false
{}
>>> x and y
?
>>> []
True or False?

Python provides boolean operators (and, or, not) to combine logical operands; the and and or return objects that evaluate to true and false, not just the values 0 or 1 (or True or False)

The order in which you write down these operators may matter!

1. For an or, Python employs short-circuit evaluation; it stops after the first operand that evaluates to true (and returns the associated object)

2. For and, operations also stop once we know the result; it will either return the object associated with the first false operand, or it will return the rightmost object

These operators function like && and || in R
[neolab cocteau] python
Python 2.5.1 (r251:54863, Jun 17 2009, 20:37:34)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> 
>>> x = [1,2,3,4]
>>> y = [2,3,4,5]

>>> x < y  # comparison is done recursively
True

>>> x[0]
1

>>> x[0] = 100

>>> x < y
False

>>> x = range(10)

>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> y = range(10)

>>> x == y
True

>>> 
if-elif-else statements

Back to this basic construction; remember from our work with the for-loop that we had to be mindful of indentation and the same is true here...
>>> x = "... this is the issue that will break your heart over and over again."
>>> if "leg" in x:
...     print "ouch!"
... elif "heart" in x:
...     print "sappy!"
... else:
...     print "no comment"
...

sappy!

>>> x = 0
>>> if x: print "found"

>>> x = 2.3
>>> if x: print "found"

found

>>> x = {}
>>> if x: print "found"

>>> x = {"a":1,"b":[2,3]}
>>> if x: print "found"

found
for loops

Unfortunately, we have a few of these in our previous lectures and it’s time to spell things out

In general, for loops circle over the associated block of statements; two statements, break and continue, can control flow through the loop

Unique to Python is an optional else statement that is executed if it doesn’t exit by break-ing out of the loop
while loops

The other kind of loop you will likely use repeatedly is a while loop

This loop will execute a block of statements repeatedly until a test turns up false

It also comes complete with break, continue and else statements