Python Core Language Elements

Python Basics

IN OUR FIRST HOUR we'll look at the core concepts and features governing Python's approach to data processing.

- **object**: a unit of data
- **variable**: an object assigned to a name
- **function**: a code "macro" that can be invoked to process objects
- **method**: a special object-related function that works with or modifies an object

All programs process data, and Python is no different. The **object** is at the heart of Python's data processing.

Every data object has a specific *type* and the first types we'll discuss are:

- **integer**: a whole number
- **float**: a floating-point number
- **string**: a sequence of characters

We'll learn how to use objects to add numbers and concatenate and modify strings. We'll learn how to incorporate user input into our programs. We'll also see how much Python cares about type, and learn how to convert one type to another according to Python's requirements.

Most importantly, we'll be exposed to Python code itself, and learn how to read it. Perhaps the most challenging aspect of learning to code for the first time is in getting used to code statements and clarifying in our own mind what's happening. Throughout our time together I'll emphasize the necessity of understanding what Python is doing in each statement -- "resolving" what you see on the screen to the dynamic behavior that it specifies, and deducing the *type* and *value* of each object.

Programs process data.

Most programs read data, process the data, and then write the data.

```
"READ" DATA SOURCES   PROGRAM              "WRITE" DATA SOURCES
database              operations           database
text file      ====>  on data       ====>  text file
user input            stored in            screen
web request           memory               web response
                      (variables)
```

Our programs will read data from a data source, store the data in memory using *variables*, operate on the data using *functions* and *methods*, and then write the data back out to a data source.

Variables

A *variable* is a label or name *bound* to a value. We can also say that the name is a *pointer* to the value. When we talk about data being stored in memory, we're talking about variables. Variables hold all of the data that we will process through operators, functions and methods.

```
#!/usr/bin/env python

var = 10                # value 10 bound to label var
my_xx = 'hello'         # value hello bound to label my_xx
```

Variable names in python can be letters, numbers or underscores, but they must not begin with numbers.

Objects **are of a particular** ***type***

Every value in Python is known as an *object*. For the first part of this course, we will define an object as a *value* of a particular *type* with characteristic *behavior*:

- **a value** (5, 3.5, 'hello', etc.)
- **of a particular type** (int, float, str)
- **with characteristic behavior** (operators, methods (to be discussed))

Everything in Python is an object, including functions, methods, and classes. For now, however, we are focusing on the data object types.

The three object types we'll discuss this week are *integers* (type **int**), *floats* (type **float**) and *strings* (type **str**).

```
var = 10                # an int object      (whole number)

your_val = 100.0        # a float object     (floating-point number)

value_33 = 'hello'      # a str object       (single quotes around characters)

my_name = "David"       # another str object (double quotes around characters)
```

Objects with operators: +

Operators (like +, *, /, ** and %) operate on two objects and return a new object (which we assign to a left-hand label). The plus sign **"+"** is special in that it works with different types of objects. *The types of objects determine the resulting behavior.*

```
sum = 5 + 9                    # 2 integers:  returns a new integer 14

newstring = 'this' + 'that'    # 2 strings:  returns a new string 'thisthat'
                               # assigned to label newstring

this = '5'                     # str with value '5'
that = '9'                     # str with value '9'

sumstring = this + that        # 2 strings: returns a new string

print sumstring                # 59
```

Notice in the above examples that Python cares about type and not value.
- an integer plus an integer returns an integer
- a string plus a string returns a string (even strings that are number characters)

In the below example, compare the variable **this** to the string 'this'. The first is a variable name, a label; the second is a string with a 4-character value ('this'). String values are always written with quotes. Variable names have no quotes.

```
this = '5'

x = 'this'                     # assigning string  'this' to variable x

y = this                       # assigning variable this to variable y (i.e., value '5')
```

Objects with operators: *

* (the multiplication operator) also works with different types of objects.

With a string on the left and an integer on the right, it returns a new string that is the string multiplied that many times:

```
this_string = 'hello' * 5
print this_string              # prints new str object:
                               # 'hellohellohellohellohello'
```

With a number on either side, it returns one value multipled by the other:

```
print 5 * 9                    # prints 45
var = 3 * 1.5                  # assigns float object 4.5 to var
```

Operators: /, **, %

These are math operators that work on numbers. They return new numeric objects (integer or float) depending on how they are used.

```
print 6/3                      # 2 (divides left int by right)

var = 9/3                      # var is integer object, value 3

powers = 8**2                  # powers is integer object, value 64
                               # (brings left int to power of right)

remainder = 6 % 4              # remainder is integer object,
                               # value 2 (remainder of 6/4)

print 13 % 3                   # remainder is integer object,
                               # value 1 (remainder of 13/3)
```

Determining object type: initialization

It is *absolutely necessary* that you be able to identify the type of *any object* at *any point in your code*. One way to determine the type of an object is by how it is *initialized* in code.

Note the syntax of each -- you can ascertain the type of an object by what it looks like when it is initialized.

```
var = 10                # an int object   (whole number)

your_val = 100.0        # a float object  (floating-point number)

value_33 = 'hello'      # a str object (single quotes around characters)

my_name = "David"       # a str object (double quotes around characters -- same as single)
```

Determining object type: object behavior

Earlier we defined an object as "a value of a particular type with characteristic behavior". One of the ways objects exhibit "characteristic behavior" is in the type of object returned from a math operation.

In each of the computations below, note that the *type of the objects involved* determines the type of the resulting object.

Math operation with two **ints** returns an **int**

```
var = 5
var2 = 10

var3 = var * var2

print var3              # (what type?  int)
                        # (what value?  50)
```

Math operation with two **floats** returns a **float**

```
var = 5.5
var2 = 5.8

var3 = var + var2     # (what type?  float)
                      # (what value?  11.3)

var4 = 15.5
var5 = 10.5

var6 = var4 - var5    # (what type?  float)
                      # (what value?  5.0)
```

Math operation with a **float** and an **int** returns a **float**

```
var = 5
var2 = 3.5

var3 = var + var2
print var3            # (what type?  'float')
                      # (what value?  8.5)
```

```
var4 = 5.0
var5 = 5

var6 = var4 + var5
print var6          # (what type?  'float')
                    # (what value?  10.0)
                    # the type is determined strictly on
                    # the types used in the operation,
                    # and *not* on the resulting value
```

+ ("plus" sign) with two strings returns a new string

```
var = 'hello'
var2 = 'world'

var3 = var + var2
print var3          # (what type?  str)
                    # (what value?  'helloworld')
```

***** ("multiplication" sign) with an integer and a string returns a new string

```
var = 'hello'

var2 = var * 3

print var2          # (what type?  str)
                    # (what value?  hellohellohello)
```

Determining Type using the type() built-in function

Our previous methods for determining type (initialization, object behavior, reading code) are preferable to the following method, but it's perfectly fine to use the **type()** built-in function to test any object to determine its type:

```
x = len('hello')    # what type is x?

print type(x)       # type 'int'
```

Functions

A *built-in function* -- so called because it's built-in to Python itself -- is a processing routine, a "little machine" that processes an object and returns an object.

The **len()** function takes a string as an argument and returns the length of the string as a return value.

```
string_length = len(newstring)    # process newstring through len
                                  # and return a new integer, assigned to
                                  # label string_length
```

We can identify a *call* by the parentheses -- when they appear after a method or function name, it means that the routine is being executed.

Arguments and Return Values

Functions may accept *arguments*, which are object(s) placed within the parentheses. Arguments may also affect how the work is done.

A *return value* is produced by a function call. We often place a new label on the left to store the returned object. Sometimes we **print** the return value directly.

The **len()** function takes a string as an argument and returns the integer length of the string as a return value.

```
string_a = "hello"
mylen = len(string_a)
print mylen                    # 5, an int
```

The **round()** function takes a float or integer as an argument and returns the same number value as a **float**, rounded down to the nearest whole number.

```
this_float = 5.3
myvar = round(this_float)
print myvar                    # 5.0, a float
```

(**round()** can also accept a second argument to specify float precision, but we'll discuss that later)

Converting Type: the built-in object constructors

We *call* the name of the object's type to create a new object of that type, or convert a different object to that type.

```
# integer to a string
mystr = str(5)                  # mystr is now '5'

# string to an integer
myint = int('5')                # myint is now 5

# integer to a float
myfloat = float(5)              # myfloat is now 5.0
```

Note the names of the functions, as they don't always correspond to what we call them: **int** is Python's name for an integer; **str** is the name for string; **float**, **list** and **tuple** are aptly named; **dict** is the name for dictionary (these last 3 will be discussed in upcoming sessions)

raw_input()

It's possible to accept input to your program while it is running through the **raw_input()** built-in function. When Python encounters this function in your code, it pauses execution and captures any keystrokes you type before hitting the [Return] key on your keyboard - returning a **str** object:

```
data_input = raw_input('Please type something, then hit "[Return]"  ')
print "You just typed '" + data_input + "'."
```

Try this program out to see the result -- anything you type at the program's prompt is echoed back, concatenated with other strings to form a sentence.

round()

The **round()** function takes a float or integer as an argument and returns the same number value as a **float**, rounded down to the nearest whole number.

```
this_float = 5.3
myvar = round(this_float)
print myvar                    # 5.0, a float
```

round() can also accept a second argument to specify float precision.

```
bigrem = 5/3.0
print bigrem              # 1.66666666667

print round(bigrem, 2)    # 1.67
```

exit()

The **exit()** function can be used to exit your program. An error message can be passed as an argument and will be printed:

```
input = raw_input()
if (input == 'goodbye'):
    exit('I see you entered "goodbye".  Okay!')    # exits with this message

print "I guess you didn't want to exit.  Oh well.'
```

Methods and Functions; Return Values

Now let's extend the idea of an object's "characteristic behavior" to object-specific functions called *methods*. A method is a *routine*, or action, that gets executed when we call it *on an object*.

When we call a method on an object, we expect something to happen - usually for the call to return a new object, or for the object to be changed. In this example, we call the **upper()** method on the **str** object **mystring**, and a new **str** object is returned, which contains the same letters, uppercased:

```
mystring = 'this'
new = mystring.upper()      # call upper on mystring and
                            # return a new string, assigned
                            # to label newstring

print new                   # 'THIS'
```

Compare *method* syntax to that of a *built-in function*:

```
mystr = 'hello'

x = len(mystr)      # call len() and pass mystr

y = mystr.upper()   # call upper() on mystr
```

As you can see, both methods and functions are *called*, which is denoted by the parentheses after the name of the function or method.

You can also see that both the **upper()** method and the **len()** function have *return values*, which are the object(s) returned from a call. The type and value of the object returned is reliable and based on the behavior defined for each in its **Usage** definition.

Why should **len()** a function and not a method? If it works on the string why don't we have **str.len()**? This is because **len()** can work with several different objects, so it has been broken out as a function.

String Methods: *inspectors* - return info on a string

str.isdigit() -- return True if the string is composed of numeric characters

```
mystring = '12345'
if mystring.isdigit():
    print "that string is all numeric characters"
else:
    print "that string is not all numeric characters"
```

str.isalpha() -- return True if the string is composed of alphabetic characters

```
mystring = 'hello' if mystring.isalpha(): print "that string is all alphabetic characters"
```

A *substring* is a portion of a string -- for example, 'hello' contains the substring 'el' (and the substring 'hel', 'ello', etc.). Simple inspection may look for a substring within a string. These methods return True or False, and so are often used in 'if' expressions. Here are some examples:

str.endswith -- return True if the string ends with a substring

```
mystring = 'This is a sentence.  '
if mystring.endswith(' '):
    print "this line has space at the end"
```

str.find -- return indexed position of substring within a string

```
mystring = 'find the name in this string'
position = mystring.find('name')
print position                          # 9 -- the 9th character in mystring
```

<u>String Methods: *mutators* - return a new string based on the value of the string</u>

str.upper() -- return the string uppercased

```
var = 'hello'
newvar = var.upper()

print newvar              # 'HELLO'
```

str.lower() -- return the string lowercased

```
var = 'Hello There'
newvar = var.lower()

print newvar              # 'hello'
```

str.replace() -- replace a string with another string

```
var = 'My name is Joe'

newvar = var.replace('Joe', 'Greta')

print newvar             # My name is Greta
```

<u>String Formatting -- str.format()</u>

When combining object values with strings (i.e., printing them together), it's useful to use string formatting:

```
this_name = 'Jose'
value = 54
print '{} is {} years old'.format(this_name, value)

print '{name} is {age} years old'.format(name=this_name, age=value)
```

The 'template' is the string, which contains bracketed tokens. The string **format** method is called on this bare string, and the arguments to **format** are the objects to be inserted. In the first ("positional") format string, the number of arguments and tokens must match. In the second ("keyword") format string, each of the named tokens must be represented in the keyword arguments list.

Besides a certain clarity, string formatting also provides type conversion -- any object inserted will be automatically converted to a string. Compare the above to regular string concatenation:

```
print name + ' is ' + str(value) + ' years old'
```

It's arguable that this is less clear, requires more typing, and makes it easy to forget type conversion. Those should be reasons enough!

<u>Arguments to a Method?</u>

Looking closely at the syntaxes for a function and for a method, you may have noticed that the method doesn't apparently have an argument.

```
this_float = 5.995
lowerstr= 'hello'

rounded_float = round(this_float)  # rounded_float value is 5.0
uppercase = lowerstr.upper()       # uppercase value is 'HELLO'
```

There is no object in the parentheses for **upper()** so it seems to have no argument. In fact the *string itself* is the argument to **upper()**.

The difference between a function and a method is that a function is a general-purpose routine that can work with different kinds of objects, and a method is a type-specific routine that works only with its intended type. (There is no int.upper(), for example.)

Arguments in Methods and Functions

A method works with the object upon which it is called, but some methods can accept additional arguments as well.

Methods and functions may accept *arguments* placed singly or in a comma-separated list within the parentheses. Arguments may be values to be processed, or they may simply affect how the work is done.

As we have discussed, a *return value* can appear on the left side of the equals sign in any method or function call. We often place a new label on the left to store the returned object.

```
string_a = "how many mans can you manually count, man?"

mans = string_a.count("man")        # str.count takes one argument:
                                    # another str object

                                    # str.count returns an
                                    # integer: the number of times
                                    # "man" was found in string_a

print mans                          # 4
```

Method and Function Return Values in an Expression; Combining Expressions

Oftentimes (though not always), a **method call** or **function call** will result in a new object being produced. We call this new object the *return value*. The most visible form of this is when we say **label = object.method()**, for example:

```
this_string = '55'
newstring = this_string.upper()    # calling upper method of string this_string,
                                   # returns a new string object assigned to label newstring
```

But, a return value from a call is not always assigned to a label. If we see the call placed in an expression, we must imagine the return value in place of the call - for example, when we **print** the return value, or when we place the call as an element of a math expression:

```
letters = "aabbcdefgafbdchabacc"

length = len(letters)              # assign integer object 20 to
                                   # new label length

print length                       # print the integer

print len(letters)                 # same (return value of
                                   # len() is printed)

print letters.count("a")           # print 5, number of times "a"
                                   # appears in letters

print len(letters) / letters.count("a")   # divide 20 by 5, and print it

print float(letters.count("a")) / len(letters) * 100  # percentage of a's in this string
```

Conversion Challenge #1: treating a string like a number

This week in our homework we'll be faced with a problem endemic when working with types: *how to treat a string like a number*.

Consider this example from earlier: two strings added together are concatenated, even if the strings contain number characters:

```
aa = '5'
bb = '5'
```

```
cc = aa + bb
print cc                        # '55', a str
```

But what if we want these two **5**'s to be treated as numbers?

The particular problem at hand is how to write a program that accepts a number from **raw_input()** in order to use it as a number. Remember, **raw_input()** always returns a **str**:

```
this_input = raw_input('please enter an integer and I will double it')

       # raw_input() returns a str --
       # at this point, assume the user enters a 7 character

print this_input * 2            # '77', a str
```

Why does Python do this? Because Python evaluates operations by *type* and not by *value*. So then, how can we get Python to convert our **raw_input()** string into a number? We use the **int()** conversion function:

```
this_input = raw_input('please enter an integer and I will double it')  # raw_input() returns a str

       # raw_input() returns a str --
       # at this point, assume the user enters a 7 character

this_int = int(this_input)      # convert the str this_input to an int
print this_int * 2              # 14, an int
```

Conversion Challenge #2: using an int to do float math

Another challenge we'll face this week is working around Python's default behavior with **ints** and **floats**.

Recall that *any math expression* involving two **int** values produces an **int**.

```
var1 = 5
var2 = 5

var3 = var1 + var2             # var3 is 10, an int
```

But what happens when we divide one **int** into another? Does the rule still hold?

```
var1 = 5
var2 = 2

var3 = var1 / var2
print var3                     # var3 is 2, an int
```

It does! Even though we know 5/2 to be 2.5, Python ignores the value and uses the objects' *types* to determine the resulting type.

Remember, an object is a *value* with characteristic *behavior*. This "**int** used with **int** returns an **int**" behavior is consistent and dependable.

The solution is to convert one of the operands to **float** so that the result is a **float**.

```
var1 = 5
var2 = 2

var3 = var1 / float(var2)
print var3                    # var3 is 2.5, a float
```

You will need to employ the solutions for both of these conversion challenges in the homework.

Sidebar: float precision and the Decimal object (1/3)

Because they store numbers in binary form, all computers are incapable of absolute float precision -- in the same way that decimal numbers cannot precisely represent 1/3 (0.33333333... only an infinite number could reach full precision!)

Thus, when looking at some floating-point operations we may see strange results. (Below operations are through the Python interpreter.)

```
>>> 0.1 + 0.2
0.30000000000000004          # should be 0.3?

>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17        # should be 0.0?

>>> round(2.675, 2)
2.67                         # round to 2 places - should be 2.68?
```

Sidebar: float precision and the Decimal object (2/3)

What's sometimes confusing about this 'feature' is that Python doesn't always show us the true (i.e., imprecise) machine representation of a fraction:

```
>>> 0.1 + 0.2
0.300000000000000004

>>> print 0.1 + 0.2
0.3

>>> 0.3
0.3
```

The reason for this is that in many cases Python shows us a *print representation* of the value rather than the actual value. This is because for the most part, the imprecision will not get in our way - rounding is usually what we want.

Sidebar: float precision and the Decimal object (3/3)

However, if we want to be sure that we are working with precise fractions, we can use the **decimal** library and its **Decimal** objects:

```
from decimal import Decimal          # more on module imports later

float1 = Decimal('0.1')     # new Decimal object value 0.1
float2 = Decimal('0.2')     # new Decimal object value 0.2
float3 = float1 + float2    # now float is Decimal object value 0.3
```

Although these are not **float** objects, **Decimal** objects can be used with other numbers and can be converted back to **floats** or **ints**.

Flow Control

Flow Control

IN OUR SECOND HOUR we'll focus our attention on the *flow* of our code *statements*. Up to now these statements have followed one another in succession, like a recipe. But we may want the code to vary its behavior depending on certain *conditions*, such as whether a number is greater than a threshold, or a string is equivalent to another string.

Conditionals

- **if**: if this condition is True, execute some code
- **elif**: if an **if** condition is False and this other condition is True, execute some code
- **else**: if an **if** and/or **elif** condition are not True, execute some code

We can combine multiple such conditions in a single test using these keywords:

- **and**: if this is True **and** that is True, execute some code
- **or**: if this is True **or** that is True, execute some code
- **not**: if this is **not** True, execute some code

Loops
In some ways programs also operate like a watch, with flywheels and gears that rotate and repeat actions multiple times.

For this we will use a *looping statement* called **while**, which executes the same block of code as many times as desired.

Debugging
However, looping makes it more challenging to determine what our code is doing if we can't see the loop happening. Did a loop execute **one** time or **1,000** times? We'll need to employ *debugging techniques* to elucidate our code's actions and break through what we might call the "fog of code".

'if' statement for flow control

So far, our programs have been pretty linear -- they flow from step A, to step B, to step C, etc. Now, we are going to add a new concept: flow control.

Flow control refers to the ability of our program to make decisions about what to do next, and to repeat certain steps multiple times.

Our programs make decisions about what to do based on the data it reads. The decisions can be made through an **if** test, more generally known as a *conditional statement*.

if executes code in its *block* only if the test is **True**.

```
var_a = 5                                # assign int object to var_a
var_b = 10                               # assign int object to var_b

if var_b > var_a:                        # compare int values for truth
    print "the test was true"            #   do some stuff,
    print "var_b is greater than var_a"  #   only if the test
    var_c = var_a + var_b                #   came out true

if var_a == var_b:                       # if the two values are equal or equivalent
    print 'the two values are equivalent'

print 'this gets printed in any case (i.e., not part of either block)'
```

The block is indicated by a uniform indent (see next). If the test returns **True**, the entire block will be executed.

The end of the block is marked by a return to the margin (or previous indent).

Python block structure using indents

If you are familiar with another programming language, you might expect to see begin- and end-braces that denote blocks. Here is some perl code:

```
# perl code
if ($vara > 5) {                         # start of block
    print "$vara is greater than 5!";
    print "this means something!";
}                                        # end of block
print "now, on with the program...";
```

However in Python, blocks use *whitespace* to show where the block begins and ends:

```
if vara > 5:                             # start of block
    print vara + " is greater than 5!"
    print "this means something!"

print "now, on with the program..."      # end of block
```

A statement that ends in a colon indicates that a block is about to begin. On the next line, Python expects an indent (standard is 4 spaces) that indicates the inside of the block.

Then, when we write a statement that is *de-dented*, or lined up with the previous indent, we are indicating to Python that the block has ended. So Python sees the last indented line as the last statement in the block.

nested blocks increase indent

Blocks can be nested within one another. We simply respect the indent of the block we're currently in, and increase the indent to indicate the nested block:

```
var_a = 5                               # assign int object to var_a
var_b = 5                               # assign int object to var_b

if var_b >= var_a:                      # compare int values for truth
    print "the test was true"           #
    print "var b is at least as large"

    if var_a == var_b:                  # if the two values are equivalent
        print 'the two values are equivalent'

    print "now we're in the outer block but not in the inner block"

print 'this gets printed in any case (i.e., not part of either block)'
```

'and' and 'or' for compound tests

Python uses the operators **and** and **or** to allow compound tests.

```
var = 5
var2 = 10
var3 = 15
var4 = 20
if var > var2 and var3 == var4:         # true if both tests are true
    print 'both comparisons returns true'


if var <= var2 or var3 < var4:          # true if either test is true
    print 'one of the comparisons returns true'
```

negating an 'if' test with 'not'

You can negate a test with the **not** keyword:

```
var_a = 5
var_b = 10

if not var_a > var_b:
    print "var_a is not larger than var_b (well - it isn't)."
```

'else' and 'elif'

An **else** block immediately after an **if** block can provide an exclusive, alternative procedure: i.e., if the test is *not* true:

```
var_a = 5
var_b = 10

if var_a > var_b:
    print "var_a is larger than var_b"
else:
    print "var_a is *not* larger than var_b"
```

Because of the presence of the 'else', you should be able to infer from the above that *one, and only one* of the two blocks will execute.

elif allows us to chain conditionals:

```
var_a = 5
var_b = 10

if var_a > var_b:
    print "var_a is larger than var_b"
elif var_a < var_b:
    print "var_a is less than var_b"
```

```
else:
    print "var_a is by definition equal to var_b"
```

Again, you should be able to infer that *one and only one* of the above blocks will execute.

elif can appear alone with **if**:

```
var_a = 5
var_b = 10

if var_a > var_b:
    print "var_a is larger than var_b"
elif var_a < var_b:
    print "var_a is less than var_b"
```

From this, we can infer that only one block can execute, but it's not guaranteed that either one will execute at all. Why? Because we have no **else**. So if this is true, do this. Otherwise, if this is true, do this. But in this case if both tests are not true, we won't see anything.

while loop

A **while** test causes Python to loop through a block continuously until the test is no longer true.

```
i = 10
while i > 0:
    print i
    i = i - 1
```

Of course, the value being tested must change as the loop progresses - otherwise the loop will cycle indefinitely (endless loop). Or, you can exit using *break* (next).

break and continue for loop control

break is used to exit a loop.

```
x = 10
while x > 0:
    answer = raw_input("do you want loop to break? ")
    if answer == 'y':
        break
    print x
    x = x - 1
```

continue jumps program flow to next loop iteration.

```
x = 0
while x < 10:
    x = x + 1
    if x % 2 != 0:      # modulus operator returns remainder of division (x/2)
        continue
    print x
```

while with **break** provide us with a handy way to keep looping until we feel like stopping. This is useful when taking interactive user input:

```
while True:
    var = raw_input('please enter an integer:  ')
    if var.isdigit():
        break
    else:
        print 'sorry, try again'

print 'thanks for the integer!'
```

Debugging loops: the "fog of code"

The challenge in working with code that contains loops and conditional statements is that it's sometimes hard to tell what the program did. I call this lack of visibility the "fog of code".

Consider this code, which attempts to add all the integers within a range (i.e., 10 + 9 + 8 + 7, etc.):

```
revcounter = 10
while revcounter > 0:

    varsum = 0
    varsum = varsum + revcounter    # add value of revcounter to varsum
    revcounter = revcounter - 1     # reduce value of revcounter by 1

print varsum                        # prints 1
```

It's quite easy to run code like this and not understand the outcome. After all, we are adding each value of **revcounter** to **varsum** and reducing **revcounter** by one until it becomes **0**. At the end, why is the value of **varsum** only 1? What happened to the other values? Why weren't they added to **varsum**?

Now, it is possible to read the code, think it through and model it in your head, and figure out what has gone wrong. But this modeling doesn't always come easily.

Some students approach this problem by tinkering with the code, trying to get it to output the right values. But this is a very time-wasteful way to work, not to mention that it *allows the code to remain mysterious*. In other words, it is not a way of working that enhances understanding, and so it is practically useless for attaining our objectives in this course.

What we need to do is bring some visibility to the loop, and begin to ask questions about what happened. The loop is iterating multiple times, but how many times? What is happening to the variables **varsum** and **revcounter** to produce this outcome? Rather than guessing semi-randomly at a solution, *which can lead you to hours of experimentation*, we should ask questions of our code. And we can do this with **print** statements and the use of **raw_input()**.

```
revcounter = 10
while revcounter > 0:

    varsum = 0
    varsum = varsum + revcounter
    revcounter = revcounter - 1

    print "loop iteration complete"
    print "revcounter value: ", revcounter
    print "varsum value: ", varsum
    raw_input('pausing...')
    print
    print

print varsum                        # 1
```

I've added quite a few statements, but if you run this example you will be able to get a hint as to what is happening:

```
loop iteration complete
revcounter value:  9
varsum value:  10
pausing...                          # here I hit [Return] to continue


loop iteration complete
revcounter value:  8
varsum value:  9
pausing...                          # [Return]


loop iteration complete
revcounter value:  7
varsum value:  8
pausing...                          # [Return]


loop iteration complete
revcounter value:  6
varsum value:  7
pausing...                          # [Return]
```

Just looking at the first iteration, we can see that the values are as we might expect: it seems that **revcounter** was 10, it added its value to **varsum**, and then had its value decremented by 1. So **revcounter** is now **9** and **varsum** is now 10.

So far so good. The code has paused, we hit **[Return]** to continue, and the loop iterates again. We can see that **revcounter** has decreased by 1, as we expected: it is now **8**. But... why is **varsum** only 9? How has its value *decreased*? We are continually *adding* to **varsum**. Why isn't its value increasing?

And if we continue hitting **[Return]**, we'll start to see a pattern - **varsum** is always one higher than **revcounter** for some reason. It doesn't make sense given that we're adding each value of **revcounter** to **varsum**.

Until we look at the code again, and see that we are also *initializing* **varsum** to 0 with every iteration of the loop. Which now makes it clear why we're seeing what we're seeing, and in fact why the final value of **varsum** is 1.

So the solution is to initialize **varsum** *before* the loop and not inside of it:

```
revcounter = 10
varsum = 0

while revcounter > 0:

    varsum = varsum + revcounter
    revcounter = revcounter - 1

print varsum                       # 55
```

This outcome makes more sense. We might want to check the total to be sure, but it looks right.

The hardest part of learning how to code is in designing a solution. This is also the hardest part to teach! But the last thing you want to do in response is to guess repeatedly. Instead, please examine the outcome of your code through print statements, see what's happening in each step, then compare this to what you think should be happening. Eventually you'll start to see what you need to do. Step-by-baby-step!

Processing Record-Oriented Data

Processing Record-Oriented Data

IN OUR FINAL HOUR THIS WEEK we'll begin to do serious work using real source data. The usual pattern for programs is 1) to *read* data from a tabular source (text file or database); 2) to *summarize* this data using summing and counting or other summary process; 3) to *report* or *write* this data to another data resource, like a text file or database.

The general pattern that we'll follow in doing this is:
1. set a *summary variable* (e.g., integer 0) that will summarize selected data
2. open and loop through each row (line) of a text file. for each line:
   a. split the line into fields (a *list of strings*)
   b. select one of the fields for analysis, and add it to the summary variable
3. once the loop is complete, report the value of the summary variable

This pattern is a standard *algorithm* and is central to our work, as source data contains the information we want, but usually not in the form that we want it. Therefore this "reading and selecting" pattern is one that you'll use throughout your career.

To do this work we'll look at some core language features:
- the **file** object for accessing files
- the **for** statement for looping through files, line-by-line
- the string **rstrip()** method for removing the *newline* character from a line
- string *slicing*, which captures a *substring* from the original
- the string **split()** method, which divides a string into a list of strings

We'll also be introduced into object methods that allow us to "slice and dice" files and strings into various forms:
- the file **readlines()** method, for reading a file as a list of strings
- the file **read()** method, for reading a file into a string
- the string **splitlines** method, for splitting a string into a list of string lines

Lastly, we'll discuss the python debugger (**pdb**) which lets us run our script line-by-line in order to gain greater visibility into our program's execution.

Records and fields

Almost all data is stored in *records*, divided into *fields*.

```
jw234  Joe    Wilson   Smithtown    NJ    2015585894
ms15   Mary   Smith    Wilsontown   NY    5185853892
pk669  Pete   Krank    Darkling     VA    8044894893
```

Data formatted like this ("tabular", or "relational" data) is central to what most of us do every day. It can take the form of a text file, a CSV file (a text file with fields separated by commas), an Excel spreadsheet, or a database result set. Here is the same data formatted as a CSV file:

```
jw234,Joe,Wilson,Smithtown,NJ,2015585894
ms15,Mary,Smith,Wilsontown,NY,5185853892
pk669,Pete,Krank,Darkling,VA,8044894893
```

A core task of our programs, both in this course and for most programmers, will be to read this data.

Files, Lists, Strings: Looping, splitting, subscripting, slicing

To *parse* (read and isolate elements of) record/field data, we'll loop through the records and split or slice each record.

This week we will emphasize three object types and five tools that at the core of this work:

Object types: **str**, **list** and **file** (Python object for file access)

1. **file (list/iterator) looping with *for*:** loops through a series of strings, returning each string line from the file

```
fh = open('filename.txt')           # file object (fh)allows looping
                                    # through a series of strings
for file_line in fh:            # file_line is a string
    print file_line             # prints each line of filename.txt
```

2. **str.split()**: splits a string, returns a list of strings

```
file_line = "jw234   Joe    Wilson   Smithtown    NJ  2015585894"
elements = file_line.split()
print elements                  # prints ['jw234', 'Joe', 'Wilson', 'Smithtown', 'NJ', '2015585894']
```

3. **list *subscripting***: isolate one element from a list

```
telephone = elements[5]         # telephone is '2015585894'
```

4. **string and list *slicing***: slices a sequence, returns the sequence as a new object of that type

```
telephone = '2015585894'
areacode = telephone[0:3]       # areacode is '201'
number = telephone[3:]          # plus4 is '5585894'

mylist = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
list_slice = mylist[2:5]              # list_slice is ['c', 'd', 'e']
```

We'll also look at reading files as lists of strings (i.e., one line per element) and as single strings (i.e., the entire text as a string). For that we'll need **len()**, which you'll remember from last week:

len of a string or list

```
mylist = ['a', 'b', 'c', 'd', 'e']
mystring = 'fghij'

print len(mylist)               # 5
print len(mystring)             # 5
```

The string object: single and double quotes

String objects can be created using single or double quotes. There is no functional difference between them. The start and end must match, but you can choose which to use at your discretion.

Some coders like to reserve single quotes for identifiers like dictionary keys; others reserve double quotes for formatted strings (both to be described later).

More practically, if you have a string that contains one type of quotes, you can use the other type to enclose them:

```
print "I wanted to tell him 'use the one you want'."
```

Otherwise, you can *escape* any quotes if within same-type quotes:

```
print "I wanted to tell him \"use the one you 'want'\"."
```

The list object: square brackets with objects separated by commas

A *list* is a *container* object. It holds an ordered *sequence* of objects. A list is **mutable** - that is, it can be lengthened or shortened during program execution. So it's ideal for building up values from a data source. A **list** can hold any single sequence of objects - string user ids, integer values, date objects, lines from a file, etc.

Later we'll see how we can create a list of list objects, something akin to a spreadsheet.

```
intlist1 = [1, 2, 55, 4, 9]                          # list of integers

id_list = ['ab123', 'ns293', 'mx441', 'mk291']     # list of string ids

mixedlist = ['a', 'b', 1, 2, 'c', 'd', 99, 99.5]  # list of objects
                                                     # of mixed type
```

The file object: a "handle" on an open file

As Python scripters we work with text files all the time, as they are a kind of "universal" data format that can be read without special tools (for example, database drivers). Text files are just streams of characters, and when we read them into our programs, they come in as strings. Let's look at three simple options for reading files: looping through the filehandle, calling **file.readlines()** and calling **file.read()**.

```
    ## consider this a file called 'students.txt'
    jw234,Joe,Wilson,Smithtown,NJ,2015585894
    ms15,Mary,Smith,Wilsontown,NY,5185853892
    pk669,Pete,Krank,Darkling,VA,8044894893
```

looping (or *iterating*) through the file object with for

```
fh = open('students.txt')          # file object allows looping through a
                                   # series of strings
for my_file_line in fh:        # my_file_line is a string
    print my_file_line         # prints each line of students.txt
```

This is the most common way to loop through a file. The **file** object is *iterable* which means we can loop through it. This means that this object will feed us a new object every time it loops, assigning each object to **my_file_line**. If the file 'filename.txt' has 3 lines, this loop will execute 3 times, and an assignment to **my_file_line** will occur three times.

In the above example, **my_file_line** is a *control variable*, a variable that is assigned the value of each element provided by the iteration. When we iterate over a file, we are iterating over lines from the file. Each file line will always be a **str**. Therefore **my_file_line** will be of type **str**. If 'filename.txt' is three lines long, **my_file_line** will be initialized 3 times. This is done invisibly - you can't see it in the code. When you see a **for** loop, you must make yourself aware of what is being iterated over, and then imagine that for each element in the iteration (in this case, for each line in the file), the control variable **my_file_line** is getting assigned the element; and the loop block executes once for each element.

Reading a file into a list with file.readlines()

file.readlines() returns a list of lines. Each element in the list is a **str** that is a line from the file. So if the file has three lines, the list returned from **readlines()** will have three elements.

```
filename = 'students.txt'
fh = open(filename)
file_lines = fh.readlines()   # file.readlines() returns a list of strings

for line in file_lines:           # here, we are looping through the list of strings
    print line                    # prints 'jw234,Joe,Wilson,Smithtown,NJ,2015585894'
                                  # then 'ms15,Mary,Smith,Wilsontown,NY,5185853892'  etc.
    print type(line)              # type 'str' -- this test is not necessary,
                                  # only to illustrate that line is a string
```

Note: keep in mind that **file.readlines()** reads the entire file into a **list** variable, which means that the entire file will be held in memory. If the file is extremely large (or your system's memory (i.e. RAM) is very small), you may experience slowness or error if you try to load the entire file into memory. In most instances this isn't an issue, however - because files are small enough and/or memory is large enough. Also, many systems will cache memory overages on disk -- which simply means that the program may not break, but it may execute very slowly.

Looping through a list with 'for'

We saw **for** in action with files, where each element in the file was a string line from the file. **for** works with any iterable sequence, including a **list**:

```
mylist = ['a', 'b', 1, 2, 'hello', 3.598]
for el_val in mylist:              # el_val can be any label
    print "element:  ",
    print el_val                     # element:  a
                                   # element:  b
                                   # element:  1
                                   # element:  2
                                   # element:  hello
                                   # element:  3.598
```

(Of course, since we're looking at each element individually, we aren't really looping *through* the **list**. Instead the object feeds us one elemnt from the list, one element at a time.)

All our containers, and some other object types, support looping with **for**. It's your new best friend - you'll be using it continuously throughout your career. Cheers!

Reading a file into a string with file.read()

read() returns a single string containing the entire file's contents.

```
>>> readfile = open('students.txt')
>>> text = readfile.read()
>>> text                          # 'jw234,Joe,Wilson,Smithtown,NJ,2015585894\nms15,Mary,Smith,Wilsontown,N
```

Note: the '\n' you see in the file is a *newline* character -- we'll discuss shortly

Note 2: as with **file.readlines()**, above, **file.read()** reads the entire file into memory -- so it shouldn't be used with extremely large files (depending on your system's RAM allocation).

str.splitlines() can split the string from **file.read()** on the newline character -- we will discuss shortly.

Opening, reading files in one line

We use the filehandle to access the file object; we also use it to **close()** the file when we're done. Although a file open is non-blocking, it's considered good housekeeping to close it as soon as the file read is done:

```
fh = open('students.txt')                  # open() returns a file object
for line in fh:                            # or, fh.readlines() or fh.read()
    print line
fh.close()
```

However, there are some shorter constructs that you may use if you know your script is going to exit within a short time (i.e., it won't stay running for hours). These constructs are extremely handy and it is fine to use them in most cases:

```
for line in open('students.txt'):          # loop through the file and print each line
    print line

lines = open('students.txt').readlines()   # read the entire file into a list

text = open('students.txt').read()         # read the entire file into a string
```

So since the filehandle is not named, it's not accessible to us; this means we can't close it.

The fact is that all filehandles, database connections, etc., are automatically closed when a script exits. So the necessity of making sure the file is closed before the script exits should be reserved for scripts that may run a long time - these may include daemon processes and the like - scripts that stay running for long periods of time. This means that if your script is expected to exit within a few minutes or even a few hours, leaving the file open until the script is done isn't a terrible transgression.

The newline character and str.rstrip(), str.lstrip() and str.strip()

Earlier we saw that **file.read()** reads out the entire file in a string, and it looks like this:

```
fh = open('students.txt')
    text = fh.read()
fh.close()

print text    # 'jw234,Joe,Wilson,Smithtown,NJ,2015585894\nms15,Mary,Smith,Wilsontown,NY,5185853892\npk669,
```

This file has been read into a single string, but note the '\n' character that marks the boundary between the end of one line in the file, and the start of the next. This is the *newline* character, a special character whose job it is to mark this boundary. Although it is invisible, it is still a character in the succession of characters that make up the file. It will always be included in a reading of any multiline file, and there are situations where we must remove it.

```
var = 'this is a line with a newline\n'
newstring = var.rstrip()
print newstring                # 'this is a line with a newline'
```

The **str.rstrip()**, **str.lstrip()** and **str.strip()** methods can remove any whitespace (spaces, tabs, and newlines), or alternatively, remove specified characters, from the end of a string, the start of a string, or both ends of a string.

```
var = 'this is a line with whitespace     \n'
newstring = var.rstrip()
print newstring                # 'this is a line with whitespace'
```

Note that this line contains a bunch of spaces and a newline character at the end. All of it has been removed (though only from the end, not the middle). We use this often on lines from a file to remove the newline character.

str.lstrip() removes whitespace from the start of the string; **str.strip()** removes whitespace from both ends:

```
var = '    this is a string\n'
print var.strip()               # 'this is a string'
```

When we want to remove a set of characters (such as punctuation) from a string, we simply create a new string of these characters (below, **strip_chars**), and pass it to **rstrip()** (or one of the other strip methods). This new string is used like a list of characters - any one of them in the string will be removed:

```
strip_chars = '.!,.:;'                       # any one of these will be removed in rstrip()

s1 = 'this sentence ends with a period.'
s2 = 'this sentence encds with an exclamation point!'

svar = s1.rstrip(strip_chars)                # period has been removed
svar2 = s2.rstrip(strip_chars)               # ! has been removed
```

Note that we're using a listing of all the characters we'd like to see removed, contained within a single string. **rstrip()** (and its buddies **lstrip()** and **strip()**) removes any that might be there - and more than one if they are there together at the end of the line.

file.read() with str.splitlines(): quick conversion of file to list of strings without newline

The **str.splitlines()** method will split a string into a list of strings, splitting on each line (on the newline character, of course). A convenient side effect of is is the removal of the newline.

Considering the same file as before (**students.txt**)

```
jw234,Joe,Wilson,Smithtown,NJ,2015585894
ms15,Mary,Smith,Wilsontown,NY,5185853892
pk669,Pete,Krank,Darkling,VA,8044894893
```

We can convert the entire file into lines this way:

```
fh = open('students.txt')
text = fh.read()
fh.close()

lines = text.splitlines()
```

Or we even combine these expressions:

```
fh = open('students.txt')
lines = fh.read().splitlines()
fh.close()
```

...and for the totally impatient, we have this little gem:

```
lines = open('students.txt').read().splitlines()
```

In any of these examples, the **lines** list will look like this:

```
print lines
             #  [ 'jw234,Joe,Wilson,Smithtown,NJ,2015585894',
                  'ms15,Mary,Smith,Wilsontown,NY,5185853892',
                  'pk669,Pete,Krank,Darkling,VA,8044894893'   ]
```

Note that the newlines have been removed from each of the string lines.

str.split(): string->list

Working with comma-separated values files, space-separated values files, or the like, we will want to isolate individual elements on a given line so we can work with them.

str.split() splits a string into a list of string elements:

```
mystr = 'jw234,Joe,Wilson,Smithtown,NJ,2015585894'
elements = mystr.split(',')
print elements                # ['jw234', 'Joe', 'Wilson', 'Smithtown', 'NJ', '2015585894']
```

split is called on a string object, takes an (optional) substring as an argument, and splits the string on the substring, dividing it into individual string objects. The substring is removed and does not appear in the string elements.

Without an argument, **split** splits on whitespace - meaning one or more whitespace characters (spaces, tabs, newlines):

```
sentence = "This  is a    sentence"
words = sentence.split()
print words                   # ['This', 'is', 'a', 'sentence']
```

Building a new line: the string join() method

split() takes a string and returns a list, split on a delimeter.

```
record = 'data1:data2:data3:data4'
field_list = record.split(':')
print field_list                 # ['data1', 'data2', 'data3', 'data4']
```

join() takes a list and returns a string, joined on a delimeter:

```
field_list = ['data1', 'data2', 'data3', 'data4']
record = ':'.join(field_list)
print record                 # 'data1:data2:data3:data4'
```

Note that **join** is actually a string method, and we call it on the delimeter itself. We pass it a sequence of some kind (often a **list**) and it returns a single string with all the string elements joined.

Needless to say, all the elements in the list must be strings - **join** doesn't do any conversions for us.

list *subscripting*: Extract an Individual Element

Once we have a string split into individual elements, we are working with a list object. Now it's easy to copy an individual element into a new variable. We index the list starting at 0. Then it's a matter of counting whole numbers to access the element we're looking for.

```
elements = ['jw234', 'Joe', 'Wilson', 'Smithtown', 'NJ', '2015585894']

var = elements[0]               # jw234
var2 = elements[4]              # NJ
var3 = elements[-1]             # 2015585894 (-1 means last index)
```

Note the last line: it's possible to index from the end using a *negative subscript*.

Reading Files: stripping the line, splitting the line

Our first task in reading files is selecting specific data from each line and pouring it into an empty structure - so we can read through, analyze, select from, slice and dice - whatever we want. We'll start with very basic forms of what we will be doing as programmers from now on: opening data sources, reading out data into structures we like, and then doing something with the newly structured data.

Also, most of the time we aren't interested in the newline character ('\n') that sits at the end of each line. So a standard practice is to use the string method **rstrip()** to get rid of it.

In this example, we are opening and reading the haiku, and selecting and storing the last word from each line, using a combination of for, split and subscripting:

```
for line in open('students.txt'):
  line = line.rstrip()
  line_words = line.split()
  print line_words[1]                     # prints each last name in the file
```

String Slicing: extracting a substring by position

Strings are *sequences* of characters, and as such they can be *sliced* by *index*:

```
mystr = '2014-03-13 15:33:00'
year =  mystr[0:4]
month = mystr[5:7]
day =   mystr[8:10]
```

The slice syntax **stringname[x:y]** acts like a function in that that it returns a new string. The *indices* **x** and **y** show the starting and ending point of the new string, counting letters.

Note the index numbers, however: for **year**, the **x** value **0** (called the *lower bound*) is the first character of the string, '2'. By this token, the **y** value, **4** (called the *upper bound*) indicates the 5th letter of the string, '-'. But note that this character is not included in the substring. Indeed, the *upper bound* is *non-inclusive*, meaning that it marks the character *after* the last character in the substring.

That is confusing. So why do they do it this way? Evolutionarily speaking, it comes from the way C programmers have to think about memory. In C, the upper bound marks the *stop character* which indicates the end of the string (and the end of the memory allocation for that string). This character is necessary in memory management, because there is no other indication of when a memory allocation ends (i.e., there is no external representation of the string -- it's just "keep reading until you hit the stop"). In Python, this sort of boundary is handled for us, and we need only indicate where in the string we want to read/slice. And yet we're still required to indicate the upper bound as non-inclusive, as if we had to account for this stop character.

String Slicing: upper and lower bounds (indices)

Another example: we might find something like this in a text file -

```
mystring = 'Serial Number:  000002345'
number = mystring[16:25]             # the 17th through 25th element
print number                         # '000002345'
```

String Slicing: missing lower and/or upper indices

```
mystr = '2014-03-13 15:33:00'
year = mystr[:4]       # slice and return first 4 characters
print year             # '2014'
```

Note the missing lower bound index - '0' is understood here

```
date_time = '2014-02-06 09:35:02'
this_time = date_time[11:]
print this_time                    # 09:35:02
```

Note the missing upper bound index - this slices to the end of the string.

All right then, here's a quiz for you: what if we have no indices at all?

```
mystr = 'This is quite a long string'
newstring = mystr[:]
print newstring                       # ???
```

answer: a copy of the entire string is returned

list slices

It's the same drill as with strings: list slices give us a new list with the selected slice:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
first_four = letters[0:4]
print first_four            # ['a', 'b', 'c', 'd']

# no upper bound takes us to the end
print letters[5:]           # ['f', 'g', 'h']
```

Remember:
1) the 1st index is 0
2) the lower bound is the 1st element to be included
3) the upper bound is one above the last element to be included
4) no upper bound means "to the end"; no lower bound means "from 0"

***stride / step* and negative stride**

A slice *stride* or *step* is an optional 3rd parameter to a slice that causes the slice to skip over elements -- every 2nd element, every 3rd element, etc.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letterslice = letters[0:7:2]
print letterslice              # ['a', 'c', 'e', 'g']  (skipped 'b', 'd', 'f')
```

A *negative stride* causes the slice to be reversed:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print letters[::-1]
         # ['g', 'f', 'e', 'd', 'c', 'b', 'a']
```

len

As we have seen with strings, the **len()** function returns an integer length of a string. Like many other features of Python, it works the same way on different types of sequences -- string, list, tuple, dictionary.

```
mystring = 'hello, there!'
stringlen = len(mystring)          # stringlen is now 13

ltrlist = ['a', 'b', 'c', 'd']
listlen = len(ltrlist)             # listlen is now 4
```

print

print is actually a convenience function - it outputs strings to the screen, but it also adds a newline to the end:

```
print 'hey'         # prints 'hey\n'
print 'there'       # prints 'there\n'
```

if we want to omit the newline, we can add a comma to the end - now a space is added:

```
print 'hey',
print 'there'
                    # prints 'hey there'
```

if we want to avoid anything being added to our strings we can use **sys.stdout.write()** -- more on this later:

```
import sys
sys.stdout.write('hey')
sys.stdout.write('there')     # prints 'heythere'
```

Reading Files: Treating a File as a List

Remember that opening a file and reading it is simple - we call **readlines()** and loop through the result. But since **readlines()** returns a list of strings, we can sometimes easily exclude and include whole lines by using a list slice:

```
fh = open('monty_haiku.txt')
file_lines_list = fh.readlines()      # a list of lines in the file
print file_lines_list
      # [  "Title:  Monty Haiku\n",
      #    "Python is so great\n",
      #    "it's elegant and simple:\n",
      #    "completely different"  ]

wanted_lines = file_lines_list[1:]    # take all but 1st element (i.e., 1st line)
for line in wanted_lines:
  print line.rstrip()                 # Python is so great
                                      # it's elegant and simple
                                      # completely different
```

As you can see, **readlines()** returns a list of strings, each of which is a line from the file. The slice **[1:]**, starting at index **1**, excludes the first line - the title line. Many times reading text files, we encounter header or footer lines that we don't want. Rather that use a counter or a line test, we can treat the whole file as a list and slice out what we do or don't want.

Sidebar: The string object -- triple quotes and newlines

We use triple quotes (""" or ''' -- either will work) for three reasons: to create multiline strings in our code; to apply multi-line comments; and to create "docstrings".

multiline strings
Triple-quoted strings are distinctly different in that they allow the use of newline characters expressed as a carriage return.

What is a newline character? It's a specific character that represents the end of a line, and it can be expressed in different ways. On the screen or printed page, a newline is invisible - but we can see it's there in that the text after it starts at the left side of the screen, one line down. In a file, a newline character is simply a line separator - it exists in the stream of text that is a file, and it simply indicates that a line has ended and a new line is beginning.

In Python, newlines may be visible or invisible. If you print a string that has newlines, Python will drop down a line and start at the left. But if you just look at its value (for example, in the python interactive prompt), you'll see it:

```
# newline in a regular string
file_text = "this is a line\nthis is another line\nthis is a third line"

# newline in a triple-quote string -- identical to above
file_text = """this is a line
this is another line
this is a third line"""
```

As you can see, triple-quoted strings allow us to enter strings with invisible newlines - in other words, multi-line text. This can be useful when we want the text to look right in our code - rather than a jumbly stream of text and newlines.

Sidebar: the string object -- triple quotes as comments and docstrings

The second and third uses of triple-quote strings are to use the strings as comments, and to create special comments called *docstrings*.

As we have seen, we use the hash mark ('#') to indicate a *comment* - that is, a line that Python is expected to ignore:

```
import sys

# this line prints something here
print 'print this thing'

# now exiting
sys.exit()
```

But with triple-quote strings, we can comment out whole sections of code:

```
print "I want to execute this line of code"
"""
print "But this line shouldn't be executed"
print "since it contains some test code"
print "I can easily uncomment to see working again"
"""
print "Okay, back to the part I want to execute"
```

docstrings are used to formally document the program, describe its purpose and show how to use it.

Note our use of the **sys** module -- we'll see many built-in modules that we must import to use. **sys** gives us access to the Python system - and the **exit()** method causes the program to stop and Python to exit.

```
#!/usr/bin/env python

"""
SYNOPSIS
    helloworld [-h,--help] [-v,--verbose] [--version]

DESCRIPTION
    This describes how to use this script.

AUTHOR
    David Blaikie
#   (there are several other headings possible; this is a sample)
"""
```

This level of formality is not required, although it often is in a professional environment. But in any case some explanation may be very useful to others (and yourself months from now).

Containers: Lists, Sets, Tuples; Functions

<u>Building up data in containers</u>

As we've discussed, data coming into most programs comes from *record-oriented* data sources, such as database tables, text sources like CSV (comma-separated values) files, excel spreadsheets, etc. Last week we learned how to loop through a data source and derive summary data from it (as we did with the Fama-French sum / average Mkt-RF value for a given year).

Frequently, we will want to do analysis on data that is not in a convenient form.
- the data source may have hundreds of rows that we don't want to look at, so we may want to filter out data before working with it
- we may want to compile a *unique set* of values from a source that has repeating values
- we may want to link a "key" value (like an id, a date, etc.) with a summary value (like a sum or a count) for any number of keys

To accomplish this, we'll follow a basic logical pattern, in which we:

```
initialize an empty container
loop through the data source
    # inside the loop
    filter for desired data
    parse out the desired data element(s) (split() or slice)
    add the data to a container

# after the above loop ends
optionally sort the data
loop through the container and report on its contents, =or=
summarize the data, sometimes through a summary function (len(), sum(), etc.)
```

Last week we looked at lists. This week we'll take another look as well as look at the other containers available to us for this work:

- **list**: ordered, *mutable* sequence of objects
- **tuple**: ordered, *immutable* sequence of objects
- **set**: unordered, mutable, unique collection of objects
- **dict**: unordered, mutable collection of object *key-value pairs*, with unique keys (discussed next week)

<u>choosing the right container for the job</u>

Each of the four containers we'll look at in this session has its own behavioral characteristics; each is appropriate to hold different classes of data.

- **list**: ordered, single, non-unique values, for example: event values (# of hits counted on an ad server, ip addresses hitting a web server, etc.)
- **tuple**: ordered, single values that don't change, for example: values that we initialize in the code, database row elements, other data coming from an external data source
- **set**: unordered, single and unique values, for example: "ID" type values (student ids in a school, client ids or names in a client database, etc.)

- **dict**: unordered, paired values, for example student ids paired with addresses, client ids paired with last month's spend, etc.) (next week)

When we are loading data from a data source, we will always consider whether the data belongs best in a list, set, dict or tuple. It is not usually hard to choose because the type of data generally lends itself to the right choice.

initializing and adding to containers

Here is the initialization syntax for each of the four containers, as well as the most common method for each that adds a value to the container.

A **list** holds an *ordered sequence* of objects. It is *mutable* - that is, it can be lengthened or shortened during program execution. So it's ideal for building up values from a data source. A **list** can hold any sequence of objects - string user ids, integer values, date objects, lines from a file, etc.

```
intlist1 = [1, 2, 55, 4, 9]                        # list of integers

id_list = ['ab123', 'ns293', 'mx441', 'mk291']     # list of string ids

mixedlist = ['a', 'b', 1, 2, 'c', 'd', 99, 99.5]  # list of objects
                                                   # of mixed type
```

We can add to a list using **list.append()**

```
intlist1.append('hello')

print intlist1                # [1, 2, 55, 4, 9, 'hello']
```

A **set** is an *unordered collection* of objects that holds *unique values*. If we add a duplicate value to a **set**, it is discarded.

```
mixed_set = set(['a', 9999, 4.3])      # initialize a set with a list or tuple
```

We can add to a set using **set.add()**

```
mixed_set.add('a')
mixed_set.add(4.3)

print mixed_set                                    # set(['a', 4.3, 9999])
```

Note that although we added a number of duplicate values, each value in the set appears only once. Also note that the elements are not in a recognizable order - the order is up to Python. Why? Because **sets** are used for *fast lookups* and *list comparisons*.

A **tuple** is an *ordered sequence* of objects, like a **list**; but *unlike* a **list** it is *immutable*. It cannot be changed once initialized.

```
mytuple = (1, 2, 3, 4, 5, 'a', 'hello')
```

A tuple is appropriate for a sequence that isn't intended to change. It my be initialized in our code, or it may be returned from a function or method call.

looping through a data source and building up containers

Once again, most programs loop through record-oriented datasets, split out elements, and then add individual elements to container objects.

Here are code examples for doing this using the **students.txt** dataset we looked at last week:

```
jw234   Joe    Wilson   Smithtown    NJ    2015585894
ms15    Mary   Smith    Wilsontown   NY    5185853892
pk669   Pete   Krank    Darkling     VA    8044894893
dz6     Dean   Zimm     Donohue      NY    5184894893
```

Remember: the basic paradigm is:

```
initialize a container
loop through the data source
    # inside the loop
    filter for desired data
    parse out the desired data element(s) (split() or slice)
    add the data to a container

# after the above loop ends
optionally sort the data
loop through the container and report on its contents, =or=
summarize the data, sometimes through a summary function (len(), sum(), etc.)
```

<u>**list**: ordered sequence of objects</u>
In the below example, we're using a **list** to store the states found on each line in **students.txt**. These state values don't particularly call for the use of a **list**, although sometimes we prefer it as an all-purpose container. So, we're using it here. A more appropriate dataset for a **list** would be one that collects values that should stay in order, like the date and time in an event log.

```
statelist = []                       # initialize an empty list
for line in open('students.txt'):
    # below is known as "multi-target assignment"
    # 6 elements are copied to individual variable names

    id, fname, lname, city, state, tel = line.split()
    statelist.append(state)

print "here are the states as encountered in the 'students.txt' file:  "
for state in statelist:
    print state

print "'NY' appears " + str(statelist.count('NY')) + " times in 'students.txt'"

# (prints 'NY' appears 2 times in 'students.txt')
```

Again, note how we are summarizing our data collection at the end. We can loop through and print the container's contents, or we can use a list method like **list.count()** to report a particular facet.

<u>**set**: unordered collection of objects</u>
Perhaps we're interested in a complete set of IDs from the data. We don't care about the order, we just want to make sure we have a set:

```
ids = set()                                            # initialize an empty set
for line in open('students.txt'):
    id, fname, lname, city, state, tel = line.split()  # note "multi-target assignment" of 6 elements
    ids.add(id)

print "here are unique ids from the students.txt file:  "
for id in ids:
    print id

print "there are " + str(len(ids)) + " unique ids in the 'students.txt' file"
```

<u>looping through containers with for</u>

For any container, once the collection of data is complete and the container has been built up, we often want to step through the container and evaluate each element -- whether to report on events (as with a **list**), or display the unique values (as with a **set**).

Whatever the container we wish to use, the **for** syntax is available for any *iterable*, that is any object that is designed to be looped over. As we saw last week, a **file** object is such a one. And so is each of this week's containers:

```
mylist = ['a', 'b', 'c']
for el in mylist:
    print el

mytup = ('a', 'b', 'c')
for el in mytup:
    print el

myset = set(['b', 'a', 'c'])
```

```
for el in myset:
    print el                              # will be printed in 'random' order
```

checking for membership with in

The **in** operator checks for *membership* within a container -- i.e., it returns **True** if the selected element is a member of the list.

```
mylist = ['a', 'b', 'c', 'd', 'e']

if 'a' in mylist:
    print "'a' is in mylist"
```

aggregate functions: len(), sum(), max(), min()

These built-in functions can give us information about a sequence without requiring us to loop.

```
mylist = [1, 3, 5, 7, 9]
mytup = (99, 98, 95.3)
myset = set([2.8, 2.9, 1.7, 3.8])

print sum(mytup)           # 292.3 sum of values in mytup
print max(myset)           # 3.8 largest value in myset
print min(mylist)          # 1 smallest value in mylist
```

If **max()** or **min()** are used on strings, it will evaluate them in something close to alphabetical order (this is discussed more in **sorting**, coming up soon).

If **sum()** is attempted on a sequence of strings, a **TypeError** exception will be raised.

getting list and tuple slice

As we saw with strings, slicing retrieves a portion of a sequence (in a string's case, a sequence of characters), and the same syntax works with lists and tuples.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
first_four = letters[0:4]
print first_four             # ['a', 'b', 'c', 'd']

# no upper bound takes us to the end
print letters[5:]            # ['f', 'g', 'h']
```

Remember:
1) the 1st index is 0
2) the lower bound is the 1st element to be included
3) the upper bound is one above the last element to be included
4) no upper bound means "to the end"; no lower bound means "from 0"

getting or setting element by subscript

subscript refers to an *index* or *key* placed within square brackets after a sequence's name:

```
mylist = ['a', 'hello', 5, 9]
mytup = [1, 3, 9, 11, 16]

print mylist[0]            # 'a'
print mytup[-1]            # 16
```

Values in lists can be set using subscripts as well (tuples are *immutable*, as you know):

```
mylist[0] = 'new element 99'
print mylist               # ['new element 99', 'hello', 5, 9
```

Note that you can't create a new index and value by specifying the next index:

```
mylist[4] = 'new element?'     # IndexError: list assignment index out of range
```

Sidebar: removing a container element

These methods are usually mentioned alongside methods like **list.append()** and **set.add()**, but I have removed them to a sidebar because they are used so infrequently. The standard workflow is to build up a container, loop through it or perform some sort of aggregation on it, and move on. We're rarely interested in removing elements from a container, although it is needed from time to time.

```
mylist = ['a', 'hello', 5, 9]
myset = set([1, 3, 9, 11, 16])

popped = mylist.pop(0)  # remove the first element from mylist
                        # (argument specifies the index to remove)

mylist.remove(5)        # remove an element by value


myset.pop()             # remove a random element
myset.remove(3)         # remove an element by value
```

Dictionaries and Sorting

dict: unordered collection of key/value pairs

A **dictionary** (or **dict**) is an *unordered collection* of *key/value pairs of objects*. The *keys* in a **dict** are *unique*, and we can use a **key** to obtain the key's associated **value** (which is why a **dict** is also known as a *lookup table*).

```
mydict = {'a':1, 'b':2, 'c':3}  # dict with str keys and int values
```

We can add to a dictionary using *subscript syntax*:

```
mydict['d'] = 4                 # setting a new key and value

print mydict                    # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

We can also *read from a dictionary* using *subscript syntax*:

```
dval = mydict['d']              # dval is 4

print mydict['c']               # prints 3
```

The nature of a dict: unordered pairs, unique keys

```
mydict = {'a': 1, 'c': 3, 'b': 2, 'd': 4}

mydict['d'] = 'heywhat'
mydict['a'] = 'omg'

print mydict             # {'a': 'omg', 'c': 3, 'b': 2, 'd': 'heywhat'}

print mydict['a']        # 'omg'
```

Note that as with a **set**, we added a duplicate key to the **dict** and it simply overwrote the value. Dictionary keys are unique -- a key cannot appear twice. Also note that the ordering of the pairs has changed. Also like a **set**, dictionaries are optimized for *fast lookups* -- so the order is determined, maintained, and changed at will by Python. (If we want to change the order we can sort the keys; or we can transform the dict into a list of *items* (discussed later).)

dict getting and setting uses the same subscript syntax!

Setting a key/value pair in a dict

```
mydict = {}
mydict['a'] = 1         # set a key and value in the dict
mydict['b'] = 2         # same

print mydict            # {'b': 2, 'a': 1}
```

Getting a key/value pair from a dict

```
vala = mydict['a']      # get a value using a key:  vala is 1
myval = mydict['b']     # get a value using a key:  myval is 2
```

Now compare the syntax for setting a key/value pair in a dict to syntax for retrieving a value based on a key.

```
mydict['a'] = 1         # set a key/value
val = mydict['a']       # get a value using a key
```

Because in adding a pair we are not using a method like **append()** or **add()**, a lot of students get hung up on this syntax. You *must memorize* the syntax for adding a key and value, retrieving a key and value, and disambiguate the two syntaxes. Do not use a method (like **update()**) to add a pair!

Building up a dict from a file

Dicts are a convenient structure for *lookups*; they are also useful for *tabulating values associated with multiple keys* (what we may call a *counting dictionary*).

dicts can be built up from a data source in the same way we did with **list**s and **set**s. We can loop through the file, build up a complete dictionary of these values, and then query the dictionary to look up any student's ID:

```
ids_names = dict()                              # initialize an empty dict
for line in open('student_db.txt'):
    id, address, city, state, zip = line.split(':') # note "multi-target assignment" of 5 elements
    ids_names[id] = state                       # key id is paired to student's state

print "here are ids and names from the students.txt file:  "
for id in ids_names:
    print "id " + id + " is from this state:  " + ids_names[id]

print "here is the state for student 'jb29':  "
print ids_names['jb29']                         #  NJ
```

looping through a dict with for

Looping through a **dict** can be done in several ways; the standard way is similar to a file or a list.

```
mydict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key in mydict:
    print key, '=>', mydict[key]           # print the key and its value
```

this prints

```
a => 1
c => 3
b => 2
d => 4
```

The elements being assigned to the *control variable* are the dict's *keys*. We then use the key to get the value using standard subscript syntax.

'in' with a dict

Testing a **dict**, the **in** operator tests for the presence of a **key**:

```
mydict = {'a': 1, 'b': 2, 'c': 3}

if 'a' in mydict:
    print "'a' is a key in mydict"
```

The fact that **in** looks at a dictionary's keys (and not the values, or the pairs) is consistent with the behavior of other common features of containers -- as, for example, we saw with **for**, which loops through the **dict** keys.

An important use of **in** with dicts is to test the presence of a key before attempting to read a value using a key. The reason this is so important is that if we attempt to read a key that doesn't exist in the dict, we'll get a **KeyError** exception:

```
print mydict['zzz']            # KeyError:  'zzz' (a Python exception)
```

Since an exception like this one halts program execution, we need to avoid it. We can use **in** to test to see that a key exists in the **dict** before trying to read it:

```
if 'zzz' in mydict:
    print mydict['zzz']
else:
    print "key 'zzz' not found"
```

There is also an alternative to this check in **dict.get()**:

```
print mydict.get('zzz', None)   # return value associated with 'zzz'
                                # if it exists, None otherwise
```

dict.get() saves us the trouble of checking to see if a key exists in the **dict** before requesting it. With the arguments of a requested key and default value, **dict.get()** returns the value associated with that key in the **dict**, the default value otherwise. This is somewhat more elegant than checking for a key ahead of time.

len() with a dict

Note that similar to **in** working with a dict, **len()** returns the number of keys in a dict.

```
mydict = { 'a': 10, 'b': 20, 'c': 30 }

print len(mydict)                    # 3
```

Reading the dict with dict.keys(), dict.values(), dict.items()

These methods produce the dictionary in different forms. We'll look at **dict.items()** next. **dict.keys()** and **dict.values()** can be useful when we're interested in just the keys or just the values of the dictionary:

```
mydict = {'a': 1, 'b': 2, 'c': 3}

these_keys = mydict.keys()
print these_keys                       # ['a', 'c', 'b']

these_values = mydict.values()
print these_values                    # [1, 3, 2]

these_items = mydict.items()
print these_items                     # [('a', 1), ('c', 3), ('b', 2)]
```

Sidebar: Dictionaries: looping with dict.items

dict.items() returns a list of tuples:

```
print b_dict.items()        # [ ('a', 1), ('c', 3), ('b', 2) ]
```

This is a 2-dimensional structure. It contains all the values in the dictionary as associated pairs, but the pairs are ordered. **items** can easily be converted back to a dictionary by using the **dict()** constructor. The **dict()** constructor requires a list of 2-element tuples -- how convenient!

```
b_dict = {'a':1, 'b':2, 'c':3}
items = b_dict.items()
c_dict = dict(items)
print c_dict                 # {'a':1, 'c':3, 'b':2}
```

A more efficient way to loop through a dictionary, **dict.items()** lets us loop through a list of 2-element tuples:

```
b_dict = {'a':1, 'b':2, 'c':3}

for pair in b_dict.items():                   # loop through pairs
  print "{0} = {1}".format(pair[0], pair[1])

for key, val in b_dict.items():               # same - the two
  print "{0} = {1}".format(pair[0], pair[1])  # elements are assigned
                                              # to vars 'key' and 'val'
```

Sidebar: removing a dict element

We can 'delete' a variable with **del** and the same is possible with a dict key.

```
var = 10

del var     # 'delete' the var variable

print var   # NameError:  'var' does not exist


mydict = {'a': 1, 'b': 2, 'c': 3}

del mydict['a']         # remove {'a': 1} from the mydict

print mydict            # {'c': 3, 'b': 2}
```

True or False -- the *boolean* object type

Like the operators + and *, comparison operators return a new object. The returned object can have a value of True or False -- the **boolean** object type.

Note: this example demonstrates the object -- NOT common usage. Common usage would be placing these tests in an **if** or **while** expression, as in previous examples.

```
var = 5 > 3                     # assign boolean object to label var
print type(var)                 # type <'bool'>
print var                       # True

var = 5 < 3                     # assign boolean object to label var
print type(var)                 # type <'bool'>
print var                       # False
```

This **boolean** object is what is tested when we place the comparison in an **if** or **while** test. We can also test boolean values directly:

```
while(True):
  cont = raw_input('continue? ')
  if cont.startswith('y'):
    break
  print "processing..."
```

The bool() constructor, True and False values, and the meaning of truth

Like **str()** for strings, **int()** for integers, and **list()** and **dict()** for lists and dictionaries, the boolean type has a "constructor", called **bool()**.

We can use **bool()** to convert any value into its boolean (True/False) equivalent. Most values evaluate to **True**, but "empty" values (0, "" (empty string), empty list, empty dict) are **False**.

However, keep in mind that a container with a 0 value in it evaluates to **True**. This is because if the container has any elements at all -- even "False" values like 0 or "" -- it is **True** (or another way to consider it: its len is greater than 0)

```
print bool([])          # False (no elements)
print bool('   ')       # True  (spaces are characters too - not empty)
print bool("")          # False (empty - no characters)
print bool({'a':1})     # True  (has a pair)
print bool({})          # False (no pairs)
```

Integers, strings and containers in a True/False expression

We can also use single objects in a truth expression, such as:

```
my_int = 5
if my_int:
  print "my_int is a true value (True if not 0)"

my_str = 'hello'
if my_str:
  print "my_str is a true value (True if not an empty string)"

mylist = [0, 0, 0, 0]                # a 'True' list (has elements)
if mylist:
  print "mylist is a true value (True if not empty - i.e. len > 0)"

mydict = {'a':0, 'b':1}              # a 'True' dict (has pairs)
if mydict:
  print "mydict is a true value (True if not empty - i.e. len > 0)"
```

if and *while* induce the bool() conversion

So when we say **if var:** or **while var:**, we're testing if **bool(var) == True**, i.e., we're checking to see if the value is a **True** value

Quiz yourself: look at the below examples and say whether the value will test as **True** or **False** in a boolean expression. Beware of tricks!

Remember the rule: if it represents a 0 or empty value, it is False. Otherwise, it is **True**.

```
var   = 5
var2  = 0
var3  = -1
var4  = 0.0000000000000001
varx  = '0'
var5  = 'hello'
var6  = ""
var7  = '    '
var8  = [   ]
var9  = ['hello', 'world']
var10 = [0]
var11 = {0:0}
var12 = {}
```

Booleans: quiz answers

```
var   = 5                        # bool(var):   True
var2  = 0                        # bool(var2):  False
var3  = -1                       # bool(var3):  True (not 0)
var4  = 0.0000000000000001       # bool(var4):  True
varx =  '0'                      # bool(varx):  True (not empty string)
var5  = 'hello'                  # bool(var5):  True
ver6  = ""                       # bool(var6):  False
var7  = '    '                   # bool(var7):  True (not empty)
var8  = [   ]                    # bool(var8):  False
var9  = ['hello', 'world']       # bool(var9):  True
var10 = [0]                      # bool(var10): True (has an element)
```

```
var11 = {0:0}                    # bool(var11): True (has a pair)
var12 = {}                       # bool(var12): False
```

None vs. False

In every language, there is a value that represents the absence of value. It is called **null** in Java and **undef** in Perl. In Python, this value is known as **None**. We use this keyword to represent this "non-value". Paradoxically, **None** really is a value - it is the value that represents no value.

However, although **None** is **False**, many things can be **True**. Here is a list of four false element values:

```
mylist = [False, 0, '', None]
```

Note that **None** is also **False** (i.e., when converted to boolean with **bool()** or in an **if** or **while** test), so we cannot discern between **None** and **False** or **None** and **0** using a truth test. If we find ourselves in a situation where we want to test for **None**, we can use an **is** expression:

```
for i in mylist:
  if not i:
    print "%s is False" % i
  if i is None:
    print "*** %s is None  ***" % i

            ## False is False
            ## 0 is False
            ##  is False
            ## None is False
            ## *** None is None ***
```

Preview: Sorting basics

To sort a sequence we can use two methods: **sorted()** which takes a sequence as an argument and returns a list in sorted order, and **list.sort()** which sorts a list in-place.

```
mylist = [4, 3, 9, 1, 2, 5, 8, 6, 7]

# sorted returns a new sorted list
sorted_list = sorted(mylist)
print sorted_list                 # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# sort() sorts the list in place
mylist.sort()
print mylist                      # [1, 2, 3, 4, 5, 6, 7, 8, 9]

namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']
print sorted(namelist)            # ['avram', 'jo', 'michael', 'pete', 'zeb']
```

So if we are sorting a list, we have the option of **list.sort()** to sort the list in-place; otherwise we can sort any sequence by putting it through the **sorted()** built-in function.

Functions and Sorting

User-Defined Functions

So far we've seen functions like **len()** and methods (functions associated with an object) like **str.upper()**. We can also create our own functions, which we'll call *user-defined functions*.

Here's a simple example of a user-defined function, which is declared with the **def** keyword. The declaration is followed by the *call*, **print_hello()**.

```
def print_hello():
  print "Hello, World!"

print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
```

When we run this program, we see the greeting printed three times. One advantage of this is that when we want the greeting printed, we can call the function by name instead of actually printing the statement ourselves. If we wanted to change our greeting so that it said "Hello, Earth!" instead, we would just change it in the function - we wouldn't have to change it three times.

User-defined functions: accepting an argument

Functions can be made to run with options or to process values. We can customize our greeting function to greet whomever we wish:

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  print full_greeting

print_hello('Hello', 'World')         # prints 'Hello, World!'
print_hello('Bonjour', 'Python')      # prints 'Bonjour, Python!'
print_hello('squawk', 'parrot')       # prints 'squawk, parrot!'
```

In this code, we say that the string objects placed within the parentheses of the call are *arguments* being *passed* to the **print_hello()** function. These arguments (there can be any number of them) are then assigned to the **greeting** and **person** variables we see in the function definition. We can then use them within the function in any way we see fit.

User-defined functions: returning a value

Functions can also return values, in the same way that the string method **upper()** returns a new string. We use the *return* keyword which is followed with the object(s) we want to return.

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  return full_greeting

msg = print_hello('Bonjour', 'parrot')
print msg                              # 'Bonjour, parrot!'
```

In this way we can think of functions as little black boxes that take data in, process it, and return data out. They may not always take arguments or return values, but they can.

Sorting: basics

To sort a sequence we can use two methods: **sorted()** which takes a sequence as an argument and returns the sequence in sorted order, and **list.sort()** which sorts a list in-place.

```
mylist = [4, 3, 9, 1, 2, 5, 8, 6, 7]

# sorted returns a new sorted list
sorted_list = sorted(mylist)
print sorted_list                # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# sort() sorts the list in place
mylist.sort()
print mylist                     # [1, 2, 3, 4, 5, 6, 7, 8, 9]

namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']
print sorted(namelist)           # ['avram', 'jo', 'michael', 'pete', 'zeb']
```

So if we are sorting a list, we have the option of **list.sort()** to sort the list in-place; otherwise we can sort any sequence by putting it through the **sorted()** built-in function.

The rest of this tutorial will focus on **sorted()**, although everything related to it will also work with the **list sort()** method.

Sorting: reversing a sort with the "reverse=True" parameter

By default, sort orders from lowest to highest. To reverse this order, add a **reverse=True** parameter (argument) to **list.sort()** or **sorted()**

```
mylist = [4, 3, 9, 1, 2, 5, 8, 6, 7]

sorted_list = sorted(mylist, reverse=True)
print sorted_list                # [9, 8, 7, 6, 5, 4, 3, 2, 1]

mylist.sort(reverse=True)
print mylist                     # [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Sorting a dictionary's keys

dict.keys() returns a list of the keys in the dictionary. You can sort the keys and store them in a list - and then loop through the list using the keys to retrieve the values from the dictionary:

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
keys = bowling_scores.keys()
keys.sort()
print keys                        # [ 'janice', 'jeb', 'mike', 'zeb' ]
for key in keys:
  print key + " = " + str(bowling_scores[key])
```

Or, we can even loop through the sorted dictionary keys directly:

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
for key in sorted(bowling_scores.keys()):
  print key + " = " + str(bowling_scores[key])
```

Sorting: "key=" element modification function

Oftentimes we may want to sort a set of values *by some criteria other than the value itself*. For example, we could sort a list of strings alphabetically, or we could sort the strings by length.

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']
print sorted(namelist)                 # ['avram', 'jo', 'michael', 'pete', 'zeb']
print sorted(namelist, key=len)        # ['jo', 'zeb', 'pete', 'avram', 'michael']
```

Lexicographical sort is standard when dealing with strings. But if we want the sort to use another method, *we can pass a function to* ***sorted*** using **key=**.

The **key=** argument does a little bit of magic: it takes each element of **namelist** and applies a function -- in this case **len** -- to get the length of the string. It then sorts each string by its length rather than its character value.

The fact that the calling of ***len()*** *is done invisibly, is the single most challenging part of understanding this sorting option.* Behind the scenes, **sorted()** is calling **len()** on each element. It does this because *we passed* ***len*** *to it*. **sorted()** passes every argument to in the list to **len**, getting back [5, 2, 7, 4, 3]. It then sorts ['avram', 'jo', 'michael', 'pete', 'zeb'] by their corresponding lengths.

Note well that we didn't pass **len()** to the function -- we passed **len**. If we had passed **len()**, what would have happened? We would have *called* **len()** with no argument - an error. *We aren't calling len ourselves* -- we're passing the function to **sorted** and *it* called **len** multiple times!

Sorting: "key=" functions (cont.)

Actually, we have to do this often when sorting alphabetically. Consider this problem of the standard sort:

```
namelist = ['Jo', 'pete', 'Michael', 'Zeb', 'avram']
print sorted(namelist)            # ['Jo', 'Michael', 'Zeb', 'avram', 'pete']
```

sorted is sorting **namelist** with capital letters first - something called an *asciibetical* sort because it uses the *ascii* table for its order.

To sort alphabetically, we need to substitute another value for sort - we can make each element lowercase for the purpose of sorting:

```
namelist = ['Jo', 'pete', 'michael', 'Zeb', 'avram']
print sorted(namelist, key=str.lower)      # ['avram', 'Jo', 'michael', 'pete', 'Zeb']
```

Keep in mind that the **key=** and **reverse=True** arguments can be in any order - they are named specifically so they can be optional and in any order. (Of course the 1st argument, the sequence to be sorted, must come first in **sorted()**)

Sorting: "key=" functions (cont.)

So the **key=** parameter must be the name of a function -- in the previous example, **str.lower** -- which as you know lowercases a string. **sort()** or **sorted()** then loops through the list, and changes each element (i.e., lowercases it temporarily) for sorting purposes. So, the original elements are sorted, but they are sorted by their lowercase equivalent.

Here's another example, sorting our string names by length instead of value:

```
namelist = ['Jo', 'pete', 'michael', 'Zeb', 'avram']
print sorted(namelist, key=len)                        # ['Jo', 'Zeb', 'pete', 'avram', 'michael']
```

Any type of object can be sorted. Here we're sorting a list of lists by their length:

```
lol = [ [1, 2, 3], [1], [1, 2], [1, 2, 3, 4] ]
sortedlol = sorted(lol, key=len)
for this_list in sortedlol:
  print this_list,               # [1]  [1, 2]  [1, 2, 3] [1, 2, 3, 4]
```

Sorting a dictionary's keys by its values

If we wanted to rank the bowling players by their scores, how can we view the dictionary sorted by value?

```
# sort a dictionary's values using the keys
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
sorted_keys = sorted(bowling_scores, key=bowling_scores.get)

print sorted_keys          # ['zeb', 'jeb', 'janice', 'mike']
```

Just to reiterate:

* the 1st argument to **sorted()** is the sequence to be sorted (in this case, a list of keys from **bowling_scores**)
* the 2nd argument to **sorted()** is the **key=** parameter, which in this case is the **dict.get** method, this time called on the **bowling_scores** dict.

So the function takes the **key** to be sorted, calls **bowling_scores.get(key)**, and sorts by the resulting value, which is the value associated with that key.

Sorting with custom function

So far we have used **key=** to sort using existing functions and methods. But we can also build our own functions for producing a sort value.

This function sorts a string name by the last name:

```
def by_lastname(name):
  fname, lname = name.split()
  return lname

names = [ 'Josh Peschko', 'Gabriel Feghali', 'Billy Woods', 'Arthur Fischer-Zernin' ]
print sorted(names, key=by_lastname)   # [ 'Gabriel Feghali', 'Arthur Fisher-Zernin'... ]
```

This function places 'David' above all other names:

```
def mefirst(objstr):
  if objstr == 'David':
    return 'AAAAAAAAAAAAAA'
  return objstr

mylist = ['David', 'Alfred', 'Abercrombie', 'Glenn', 'Baskerville', 'Adam']
print sorted(mylist, key=mefirst)  # ['David', 'Abercrombie', 'Adam'...]
```

In this function, only 'David' gets sorted as 'AAAAAAAAAAAAAA', and the others are sorted as they are normally. We can convert the sortable value into anything we wish - playing favorites all the way.

Sidebar: 'Cascading' sort

To sort by one property or key, and then by another (when two of the first sort key are the same), we sort by the 2nd key first, then the 1st key.

The reason this works is that sorted elements from one sort are guaranteed to keep their order except where required by any additional sorting - in other words, if you sort by first name, and then sort that list by last name, the first names will stay in their relative order when possible.

For example, consider our previous example, but with some repeated last names:

```
def by_lastname(name):
  fname, lname = name.split()
  return lname

names = ['Zeb Will', 'Deb Will', 'Joe Max', 'Ada Max']
fnamesorted = sorted(names)                           # ['Ada Max', 'Deb Will', 'Joe Max, 'Zeb Will']
```

At this point, Ada Max is ahead of Joe Max and Deb Will is ahead of Zeb Will, even if they are not sorted by last name.

```
lnamesorted = sorted(fnamesorted, key=by_lastname)    # ['Ada Max', 'Joe Max', 'Deb Will', 'Zeb Will']
```

When we do sort by last name, these relationships are preserved within the bounds of a last name sort (i.e., names are sorted by last name, but between equal last names, the previous first name sort order is retained.

File and Directory Input / Output (I/O)

Communicating with the Outside World

Our programs rarely work in isolation. We have been reading from files, but we must learn the tools to communicate with the outside world: on the command-line, writing to STDIN and STDOUT as well as to files, and file and directory operations.

sys.argv:

```
import sys

arg1, arg2 = sys.argv[1:3]       # arguments from the command line
```

writing and appending to files:

```
fh = open('writefile.txt', 'w')
fh.write('a line of text')
fh.close()

fh2 = open('appendfile.txt', 'a')
fh2.write('an appended line of text')
fh2.close()
```

command-line I/O with STDIN and STDOUT:

```
import sys

sys.stdout.write('this goes to STDOUT without modification')

text = sys.stdin.read('reading text input from STDIN')
```

os module for directory and file I/O:

```
import os

files = os.listdir('some/directory')

pathname = os.path.join('some/directory', 'some_file.txt')

if os.path.isfile(pathame):
    print '{} is a file'.format(pathname)

bytes = os.path.getsize(filename)
```

Arguments to the program with sys.argv

We have seen numerous functions that take arguments. Our program can also take arguments. We might modify the Fama French file reader to take input at the command line, which would free us from having to enter the input interactively (i.e., through **raw_input()**):

```
mycomputer$  python FFreader.py monthly MktRF 5
Returning top 5 monthly returns for Mkt-RF:
193208:  33.25
193304:  33.23
...etc
```

In the above run of the program, we passed 3 arguments: the word 'monthly', the word 'MktRF' and the number '5'. These arguments are read by the program and used to select the data requested. Here's how the program reads these arguments:

```
import sys

program_call = sys.argv[0]    # this reflects how the program was called, e.g. python test.py
period = sys.argv[1]          # the first argument to the program
factor = sys.argv[2]          # the second argument...
num_results = sys.argv[3]

# or, we could read them in one line with multi-target assignment:
period, factor, num_results = sys.argv[1:4]
```

Most scripts that run with options use command-line args rather than interactive input with **raw_input()**. Besides allowing for faster input, using arguments also allows the program to be run from another program or cron job, i.e. one that can't interact through the keyboard.

Writing and appending to files

Files can be opened for reading *or* writing (or appending, another form of writing). So when we open a file, we can specify that it be opened for reading, writing, or for appending. Since *reading* is the default, we usually only specify **'w'** or **'a'** for writing or appending.

Writing means that the file contents will be replaced (or a new file will be created). *Appending* adds to an existing file - common uses are for log files.

Note that we must add newlines to **write()**:

```
fh = open('new_file.txt', 'w')
fh.write("here's a line of text\n")
fh.write('I add the newlines explicitly if I want to write to the file\n')
fh.close()
```

```
lines = open('new_file.txt').readlines()
print lines
  # ["here's a line of text\n", 'I add the newlines explicitly if I want to write to the file\n']
```

Appending involves the same process - simply use 'a' instead of 'w' in the call to **open**. Now the contents are not replaced, but instead added to:

```
fh = open('new_file.txt', 'a')
fh.write("yet another line\n")
fh.close()

lines = open('log_file.txt').readlines()
print lines
  # ["here's a line of text\n", 'I add the newlines explicitly if I want to write to the file\n', 'yet ano
```

opening and reading a file in 'with' context

```
with open('students.txt') as fh:   # file is opened
    for my_file_line in fh:
        print my_file_line
print 'done'                       # outside the block:
                                   # file is automatically closed
```

The **with** statement marks the beginning of a special block used with some objects. When Python leaves a **with** block, special functionality can be triggered automatically (something called *context management*, or doing something when we leave the **with** context). With a **file** object, **with** triggers the file to automatically close.

While files should be closed explicitly when work with the file is done, many programs fail to do so. This usually isn't a problem because all files will be closed when a script ceases execution. However, it's considered good practice to close the file as soon as possible.

Enter the **with** block. File objects initialized using **with** are configured to automatically close when Python exits the block. This allows you to 'have your cake and eat it too': you don't have to explicitly close the file, because Python is doing it for you.

stdin and stdout

In addition to files, there are three built-in data "streams" that our programs can read from or write to. These "streams" are called *STDIN* ("standard in"), *STDOUT* ("standard out") and *STDERR* ("standard error"). We call them *datastreams*, which simply means they funnel data (usually text) in and out of our programs.

The most common use of **stdout** we have seen is the **print** statement. **print** sends whatever string argument we choose to **stdout**. By default, **stdout** writes to the screen. However, it can be redirected to a different resource, for example to a file. This is often done on the command line - i.e., through a Terminal prompt (a Command Prompt in Windows):

```
mycomputer$  python hello.py
hello, world!
mycomputer$  python hello.py > newfile.txt

mycomputer$  cat newfile.txt                # cat spits out a file's contents

C:\$  type newfile.txt                      # same, but for Windows
hello, world!
```

In this example we see a Unix terminal prompt (in bold) followed by a command to run **hello.py**. We see that **hello.py** prints a greeting. In the next command, the **>newfile.txt** redirect says "anything that is printed through **stdout**, please write to **newfile.txt**". So any print statements in this script will write to **newfile.txt** -- without us having to open the file for writing. Note that new files will be created, and existing file will be overwritten with the **>** redirect.

The **cat** unix command -- and the **type** Windows command -- can reveal that **newfile.txt** has had the greeting written to it.

Writing to stdout directly (not using print)

You may remember that **print** prints a string, and that it automatically appends a newline at the end of each print statement, and print with a comma appends a space. These is are often everything we need. However, there are times when we want to print something raw - in this case, we can use **sys.stdout.write()**:

```
import sys
print "hello!"                    # print the string with a newline appended
print "hello!",                   # (with a comma), append a space instead
sys.stdout.write('hello!')        # do not append any character
```

So only **sys.stdout.write()** will write to **stdout** with no character attached. Since this is the least common choice, it takes a little extra work to perform.

As with most other library functions, **sys** must be imported before any of its attributes can be referenced - otherwise Python will say that 'sys' is not defined.

Reading from stdin

We use **stdin** when we want to direct some data into the program - for example, we might want to feed a file into our program and have it read it:

```
import sys

for line in sys.stdin.readlines():
  print line

## or, alternatively:
## filetext = sys.stdin.read()
```

A program like the above could be called this way:

```
mycomputer$ python readfile.py < filetoberead.txt
```

Why would we want to do this instead of using file **open()** with a submitted filename? The **stdin** datastream can also come from another program and *piped* into the program:

```
ls -l | python readfile.py     # unix
dir | python readfile.py       # windows
```

In this command, the **ls** unix utility (which lists files) outputs to **stdout**, and the *pipe* (the vertical bar) passes this data to **readfile.py**'s **stdin**. It's common practice in unix to pass one program's output to another's input.

Directory Listing with os.listdir()

Just as we look at a file as a list of strings, we look at a directory listing as a list of strings:

```
import os                    # os ('operating system') module talks to the os (for file access & more)
mydirectory = '/Users'

for item in os.listdir(mydirectory):
  print item                 # .localized
                             # Shared
                             # dblaikie
                             # gwilson
                             # tosibodu ...
```

Here we see all the files in my mac's home directory. If we then wanted to open each file and -- for example -- read the contents, we could do it this way:

```
import os
this_dir = '/Users/dblaikie'
for item in os.listdir(this_dir):
  filepath = os.path.join(this_dir, item) # joins directory to file to make /x/x/y
  if os.path.isfile(filepath):                # test to see if it's a regular file
    try:
      text = open(filepath).read()
    except IOError:  continue                 # if file can't be read, IOError exception - just keep going
    print "Contents of " + filepath + ":"
    print text
    print "=" * 10                            # add a line between files
    print "=" * 10
```

Note that we must specify the directory when opening the file - **filepath** is the full path to the file, including the directory. This is because the operating system makes no assumptions about the location of a given file. It searches the current directly only. If we want to refer to a file in another directory, we need to prepend the filename with the path.

Note too: you will find that binary files may produce strange terminal output; detecting whether a file is plaintext or binary can be done somewhat reliably with the **mimetypes** module.

File tests: os.path.isfile(), os.path.isdir()

When we list a directory, we may see files and other directories. (In Unix there are a number of other entities we may encounter -- symbolic links, named pipes, device files, etc. - though we don't have to worry about these.)

If we are only interested in reading files, we may want to check to see if a listing is a file before proceeding. We do this with the **os.path** module.

```
import os

# print the names only of files and directories
this_dir = '/Users/dblaikie'
for listing in os.listdir(this_dir):
  if os.path.isfile(os.path.join(this_dir, listing)):
    print listing.upper()
  elif os.path.isdir(os.path.join(this_dir, listing)):
    print "*" + listing + "*"
```

File test: os.path.getsize()

One common measure of a file is its size, which is given in bytes:

```
import os

this_dir = '/Users/dblaikie'
for listing in os.listdir(this_dir):
    filepath = os.path.join(this_dir, listing)

    if os.path.isfile(filepath):
        print "file {0} is {1} bytes".format(listing, os.path.getsize(filepath))

           ## file file.txt is 34 bytes
           ## file yourfile.txt is 183 bytes
           ## file test.csv is 1803 bytes
```

Sidebar: argparse

Formal program argument validation is provided by **argparse** module. Besides validating your arguments and making them available as object attributes, the module will respond to --help by summarizing available arguments as well as other help text that you can define.

Here's a simple example:

```
import argparse

parser = argparse.ArgumentParser()

# add a boolean flag
parser.add_argument('-b', '--mybooleanopt', action='store_true', default=False)

# add an option with a value
parser.add_argument('-v', '--myvalueopt', action='store')

# 'args' is an object with readable attributes
args = parser.parse_args()

print args.mybooleanopt
print args.myvalueopt
```

Options are available to make some args required, to require a certain type of one of a set of valid values, and much more. See the docs here.

Sidebar: recursive directory listing with os.walk()

Oftentimes we'll want to list an entire *directory tree*, i.e. all the files in this directory and all subdirectories within.

```
import os
root_dir = '/Users'
for root, dirs, files in os.walk(root_dir):     # root string, dirs list, files list
  for dir in dirs:                              # loop through directories in this directory
    print os.path.join(root, dir)               # print full path to dir
  for file in files:                            # loop thgrough files in this directory
    print os.path.join(root, file)              # print full path to file
```

os.walk does something magical (and invisible): it traverses each directory, descending to each subdirectory in turn. Every subdirectory beneath the root directory is visited in turn. So for each loop of the outer **for** loop, we are seeing the contents of one particular directory. Each loop gives us a new list of files and directories to look at; this represents the content of this particular directory. We can do what we like with this information, until the end of the block. Looping back, **os.walk** visits another directory and allows us to repeat the process.

Exceptions

Every error Python encounters results in an *exception*: an error condition identified by a type. We have seen several such exceptions while writing our programs - anytime Python is unhappy with our code, it raises an Exception and tells us the type of exception that was raised:

```
>>> var = 5
>>> print ver
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'ver' is not defined
>>> print 'hello
  File "", line 1
    print 'hello
               ^
SyntaxError: EOL while scanning string literal
>>> mydict = {'a':1, 'b':2, 'c':3}
>>> print mydict['x']
Traceback (most recent call last):
  File "", line 1, in
KeyError: 'x'
```

Exceptions: catching

If we expect a possible exception - for example, if we are looking for a dictionary key that doesn't exist, or if we are calling **int()** on a string value but it turns out not to be a number - we can handle the exception and prevent it from stopping execution of the program. We do this with a **try/except** block:

```
mydict = {'a':1, 'b':2, 'c':3, 'd':4}
key = raw_input('please input a key')
try:
  print "the value for " + key + " is " + str(mydict[key])
except KeyError:
  print "the key " + key + " does not exist"
```

A common and useful idiom for taking user input is to put the **raw_input** call into a **while** loop, asking the user for input repeatedly until the input is in the correct form or value:

```
while (True):
  number = raw_input("please input a number:  ")
  try:
    this_integer = int(number)
    break
  except ValueError:
    print "sorry, that wasn't a number.  try again"
```

When handling expected errors, we have the option of checking first to make sure the operation we are contemplating will go through without error, or just bulling through and catching any exceptions that result. The general consensus in the Python community is that it's better to catch an exception than do a bunch of testing. Testing takes more work to maintain and is not as clear in the code. An exception clearly names the problem being handled, unlike a mere test. (In the end, though, it is up to you the coder to choose the approach that's best for you.)

How do we know which exception to catch? It's easy to identify the exception type -- simply induce the error and see which exception Python reports. This is the concept of *asking for forgiveness* -- we break the dish first, then we set up our try/except block to handle it the next time it happens.

Exceptions: multiple types and chaining

If more than one exception is possible, we can catch all of them at once:

```
try:
    # open file passed at command line
    filename = sys.argv[1]
    fh = open(filename)
except (IOError, KeyError):
    exit('please enter readable file')
```

or, we can chain the exceptions together, handling each one separately:

```
try:
    filename = sys.argv[1]
    fh = open(filename)
except KeyError:
    exit('please enter a filename')
except IOError:
    exit('the file you entered does not exist or is not readable')
```

Sidebar: time.datetime and time.timedelta

Python uses the time.datetime library to allow us to work with dates. Using it, we can convert string representations of date and time (like "4/1/2001" or "9:30") to datetime objects, and then compare them (see how far apart they are) or change them (advance a date by a day or a year).

```
from datetime import datetime    # load the datetime library
dt = datetime.now()              # create a new date object set to now
dt = datetime.today()            # same

dt = datetime(2011, 7, 14, 14, 22, 29)
                                 # create a new date object by setting
                                 # the year, month, day, hour, minute,
                                 # second (milliseconds if desired)

dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
                                 # create a new date object
                                 # by giving formatted date and time,
                                 # then telling datetime what format we used
```

Once a datetime object has been created, we can view the date in a number of ways

```
print dt                         # print formatted (ISO) date
                                 # 2011-07-14 14:22:29.045814

print dt.strftime("%m/%d/%Y %H:%M:%S")
                                 # print formatted date/time
                                 # using string format tokens
                                 # '07/14/2011 14:22:29'

print dt.year                    # 2011
print dt.month                   # 7
print dt.day                     # 14
print dt.hour                    # 14
print dt.minute                  # 22
print dt.second                  # 29
print dt.microsecond
print dt.weekday()               # 'Tue'
```

Comparing datetime objects

Often we may want to compare two dates. It's easy to see whether one date comes before another by comparing two date objects as if they were numbers:

```
d1 = datetime(2011, 7, 14, 9, 40, 15)
                                  # new date object:  July 14, 2011  9:40:15am

d2 = datetime(2011, 6, 14, 9, 30, 00)
                                  # new date object:  June 14, 2011  9:30:00am

print d1 < d2                     # True

print d2 > d1                     # False
```

"Delta" means change. If we want to measure the difference between two dates, we can subtract one from the other. The result is a timedelta object:

```
td = d1 - d2        # a new timedelta object
print td.days       # 30
print td.seconds    # 615
```

Between June 14, 2011 at 9:30am and July 14, 2011 at 9:45am, there is a difference of 30 days, 10 minutes and 15 seconds. timedelta doesn't show difference in minutes, however: instead, it shows the number of seconds â€“ seconds can easily be converted between seconds and minutes/hours with simple arithmetic.

Changing a date using the timedelta object

Timedelta can also be constructed and used to change a date. Here we create timedelta objects for 1 day, 1 hour, 1 minute, 1 second, and then change d1

```
from datetime import timedelta
add_day = timedelta(days=1)
add_hour = timedelta(hours=1)
add_minute = timedelta(minutes=1)
add_second = timedelta(seconds=1)

print d1.strftime("%m/%d/%Y %H:%M:%S")             # '07/14/2011 9:40:15'

d1 = d1 - add_day
d1 = d1 - add_hour
d1 = d1 - add_minute
d1 = d1 - add_second

print d1.strftime("%m/%d/%Y %H:%M:%S")             # '07/13/2011 8:39:14'
```

Functions and Modules

Review: User-Defined Functions

So far we've seen functions like **len()** and methods (functions associated with an object) like **str.upper()**. We can also create our own functions, which we'll call *user-defined functions*.

Here's a simple example of user-defined function, which is declared with the **def** keyword. The declaration is followed by the *call* (issued three times), **print_hello()**.

```
def print_hello():
  print "Hello, World!"

print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
```

When we run this program, we see the greeting printed three times. One advantage of this is that when we want the greeting printed, we can call the function by name instead of actually printing the statement ourselves. If we wanted to change our greeting so that it said "Hello, Earth!" instead, we would just change it in the function - we wouldn't have to change it three times.

Review: Accepting a function argument

Functions can be made to run with options or to process values. We do this by passing *arguments* to functions. We can customize our **greeting** function to greet whomever we wish:

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  print full_greeting

print_hello('Hello', 'World')         # prints 'Hello, World!'
print_hello('Bonjour', 'Python')      # prints 'Bonjour, Python!'
print_hello('squawk', 'parrot')       # prints 'squawk, parrot!'
```

In this code, we say that the string objects placed within the parentheses of the call are *arguments* being *passed* to the **print_hello()** function. These arguments (there can be any number of them) are implicitly assigned to the **greeting** and **person** variables we see in the function definition. We can then use them within the function in any way we see fit.

Review: Returning a function "return value"

Functions can also return values, in the same way that the string method **upper()** returns a new string.

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  return full_greeting

msg = print_hello('Bonjour', 'parrot')
print msg                              # 'Bonjour, parrot!'
```

In this way we can think of functions as little black boxes that take data in, process it, and return data out. They may not always take arguments or return values, but they can do either.

positional and *keyword* Arguments

There are a number of ways to pass arguments. The default is *positional*, which means that the first argument is assigned to the first label in the **def**, the second to the second, etc.:

```
def myfunc(firstname, lastname):
  print "Your name is " + firstname + " " + lastname

myfunc('Joe', 'Wilson')               # Your name is Joe Wilson
```

Another way is *keyword*: these are *optional* arguments, which may or may not appear in the call. The example we have seen so far is in the **sorted** function:

```
sorted_vals = sorted(mylist)
sorted_vals = sorted(mylist, reverse=True)
sorted_vals = sorted(mylist, key=len, reverse=True)
sorted_vals = sorted(mylist, reverse=True, key=str.upper)
```

As you can see, we can *optionally* include one or either of the **key** or **reverse** arguments, but we don't need to include either. The list to sort (in this example **mylist**), is required. It is **positional** -- so we can combine positional with keyword arguments.

In the **def**, keyword arguments look like this -- they feature a default value that will be used if the argument is not passed:

```
def sorted(list_to_sort, reverse=False, key=None):
  ...
```

Here, we use the same format **name=value** in the arguments list, but in this case the format defines the default value that will be used if the argument is not passed. In the above example, if the sequence to be sorted is the only argument passed, we'll see **list_to_sort** as the sequence that is passed, **reverse** will be set to **False**, and **key** will be set to **None**.

Of course this means that we can design our own functions so that some arguments are "positonal" and thus required, and others are "keyword" and optional.

Arbitrary Number of Arguments

Another argument pattern allows for an arbitrary number of arguments to be passed. We do this with an asterisk before the argument name -- now the caller can pass 1, 2 or 20 arguments:

```
def print_args(*passed_args):
  for i in passed_args:
    print i,

print_args(1, 2, 3)        # 1 2 3
print_args(1)              # 1
print_args(4, 5, 6, 7, 8)  # 4 5 6 7 8
```

In this example, the variable **passed_args** is a **tuple**.

Function Scoping

scoping refers to the availability of variables. A variable created in a scope is available only in that scope. In other languages like **C** and **Java**, *block scope* is employed:

```
if (1) {
  int a = 10;
  a++;
}
System.out.println(a);          # error:  'a' does not exist here
```

The braces define the block, and the block defines the scope. If a variable was created inside the block, it isn't available outside the block.

In Python, standard blocks (**if**, **for** and **while**) don't create a variable scope:

```
if True:
  a = 10

print a      # 10
```

Instead, Python enforces scoping only in function **def** blocks:

```
def myfunc():
  a = 10
  return a

var = myfunc()      # var is now 10
print a             # error:  'a' does not exist here
```

What this confers upon us is the ability to use functions to sequester portions of our code from other parts. Variables will not interfere with one another; we can treat each little function "machine" as a separate entity that does its job and doesn't increase the complexity of the overall program.

LEGB rule

For the most part, our variable names will not clash with one another. Hopefully judicious design -- use of user-defined functions and modules -- will divide code into functioning parts that work independently. But it can be helpful to know about Python's *lookup scopes* -- the "places" Python looks for our variable names.

The lookup scopes tell us where Python will look when we reference a variable -- and they happen in order:

1. local scope (i.e., inside a function)

2. enclosing scope (inside a nested function)

3. global scope (i.e., the program itself)

4. built-in scope

In other words, if we are inside a function, and we reference the variable **a**, Python looks inside the function for the variable. If it can't find it there, it looks outside of the function. If it can't find the variable in the program level, it looks in the **builtin** scope:

```
def myfunc():
  len = 'inside myfunc'
  print len

print len                 # prints '<built-in function len>'

len = 'in global scope'
print len                 # prints 'in global scope'

myfunc()                  # prints 'inside myfunc'

print len                 # prints 'in global scope'

del len
print len                 # prints '<built-in function len>'
```

As you can see, **len** is not a *reserved word* -- i.e., Python has no problem if you reassign the label to some other object. We rely on the *LEGB lookup* to determine the value of a given label.

In fact, other labels that we rely on -- **list**, **tuple**, **bool**, **type**, and many others that we have come to rely on -- can be reassigned. It seems to have been a design decision by the Python architects to allow label reassignment - as a result, we must be aware of the built-in labels and take care not to use them if we intend to use them for their built-in purpose.

global keyword and a global/local gotcha

Occasionally we may try to use a global within a local (i.e., function) scope and encounter an error:

```
x = 99
def selector():
  x = x + 1             # written out:  var = x;  x = var + 1
selector()
Traceback (most recent call last):
  File "test.py", line 1, in
  File "test.py", line 2, in selector
UnboundLocalError: local variable 'x' referenced before assignment
```

Here we're trying to add one to the global **x**. But the problem -- something that confuses Python -- is that we are both retrieving the value of **x** and then setting the value of **x** within the same function. When we assign a value to **x** inside a function, we are identifying to Python that the variable is a local variable. But when we try to get the value of **x** before setting the value of **x**, Python balks -- it doesn't understand why we would try to retrieve the value before setting it. It seems out of order, but all we need to do is say **x =** and Python identifies the variable **x** as local -- and so it doesn't even look for the global value of **x**.

The solution is to tell Python explicitly that **x** is a *global* variable, not local to the function. We do this with a **global** declaration:

```
x = 99
def selector():
  global x
  x = x + 1
selector()
print x             # 100
```

built-in scope

All the "built-in functions" with which we are familiar are part of the *built-in scope* (the **B** in **LEGB**) and they can be viewed through the **builtin** module. Note that all the **Exception** classes are here (**KeyError**, **TypeError**, **ZeroDivisionError**), as well as some familiar functions: **len**, **print** and **dir** as well as built-in object converter / constructor functions: **str**, **list**, **dict**, etc.

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'De
```

Interestingly, we must **import __builtin__** before we can reference it - even though built-ins are always available as variables.

nested scope

Since functions are objects like any other, and can go anywhere objects can go (for example inside container objects), it's also possible to define a function inside the scope of another function. A good example is a **sort** function that you want to use in only one place. Here we might use a **lambda**, but we might prefer a named function to do the work. But, we only want to use it here - it doesn't make sense to make it available to the whole program:

```
def process_names(names):

  def by_last_name(name):
    return name.split()[1]

  if not names:
    return None
  sorted_names = sorted(names, key=by_last_name)
  return sorted_names

process_names(['Marina Sirtis', 'Gates McFadden', 'Wil Wheaton'])

last_name = by_last_name('George Wilson')           # ERROR:  function does not exist
```

Note that **by_last_name** isn't available outside of **process_names** -- it's inside that scope, so it can't be found outside the function.

The *module* system for sharing code

Like most other languages, Python allows for storage of code in a *library*. The **lib** folder is sometimes the place where our own library code is stored.

This library code is stored in Python programs that we call *modules*. In fact all Python programs are known as **modules**, though we sometimes use the term to mean external code that we want to import into our own programs.

Publicly available modules that do useful things are made available to us through Python's package library. **NumPy** and **pandas** allow for efficient data transformations and formulation. **datetime** helps us work with dates. **Tk** allows us to create a Graphical User Interface.

Privately (for ourselves or for our company) we may create our own libraries that do useful things specific to our own work. We would simply store these functions and/or other objects in a regular python program with a **.py** extension, and save the program in a central location. Our other programs can then *import* the code at will, and make use of it.

writing a module, import and the module namespace

To write a module, we write a Python program that doesn't actually *do* anything - it just sets variables - in this case, function defs.

```
"""messages.py:  write to logs and print error messages."""

import sys

def print_warning(msg):
  """write a message to the error stream"""
  sys.stderr.write(msg)

def log_message(msg):
  """write a message to the log file"""
  try:
    fh = open('log.txt', 'a')
    fh.write(str(msg) + '\n')
  except IOError:
    print_warning('log file not readable')
```

To use this module in another program, we can *import* it:

```
#!/usr/bin/env python
"""test.py:  test program."""

import messages

print "test program running..."

messages.log_message('this is an important message')
messages.print_warning("I think we're in trouble.")
```

Note that our module file is named **messages.py** but we're importing it with **import messages**. Python assumes the **.py** extension and in fact would be confused if we included it.

Note also that using variables from another module requires that we prepend the module name to the variable name (e.g. **messages.log_message**). When we import a module, we often don't want all of its variables to interfere with our own. Namespaces keep a library's variables separate from our own. If we wish to use a module's names, we can do so (next).

from <modulename> import <attributenames>

We can import variable names directly into our code if we wish. Generally, we want to do it explicitly, or with the module writer's permission.

```
#!/usr/bin/env python

from messages import print_warning, log_message

print "test program running..."

log_message('this is an important message')
print_warning("I think we're in trouble.")
```

We can also import all attributes with the **from messages import *** but the assumption is that we know what attributes are coming in and we'll handle any potential conflicts ourselves.

import <module> as <easyname>

A simple, useful and common shortcut is to import the module under a different name. We can avoid bringing variables into our own namespace, but still make the typing more convenient, at least - the **as** keyword renames **pandas** so we don't have to type the name out each time.

```
import pandas as pd
users = pd.read_table('myfile.data', sep=',', header=None)
```

We can even import variables under different, more convenient names:

```
from pandas import read_table as rt

users = rt('myfile.data', sep=',', header=None)
```

module attributes; __dict__ and dir()

Remember the rule from our first lesson: everything is an object, and every object has attributes.

The variables that exist as **globals** inside a module become **attributes** of the module when we import it. So if we can say **import sys** and then **print sys.argv[0]**, this means that **argv** is a global variable (a **list**) named **argv** inside the **sys** module.

We can see what attributes are part of a module using the standard function **dir()**; another attribute, **__dict__**, lists out the attributes of the module, similar to (but not always the same as) the **dir()** function.

Actually, most objects have a **__dict__** attribute that lists out the object's other attributes; this is the first thing that **dir()** reads. (The __ before and after the name indicates a variable that is not usually intended to be read directly, although we are free to if we wish.)

module search path

When we say **import modulename**, Python will look for **modulename.py** in the current (present) working directory. If it doesn't find it there, it won't find it unless **modulename.py** is in the *module search path*.

The module search path is composed of a number of possible elements:

1. present working directory -- i.e., the location from where the script was run

2. standard library directories -- these are set by Python during installation and are not generally used by us

We can see all the directories in the module search path by looking at the **sys.path** variable.
* the PYTHONPATH system environment variable -- a UNIX or Windows environment variable that you can use to set module directories for your computer or your company's system
* **.pth** files -- these are special text files that can be placed anywhere on the search path; they can contain additional paths to search (these can used for portability)

Note that some of these locations are directories, like **python2.7** below, which contains many **.py** modules such as **urllib.py** and **cgi.py**; other entries are **.egg** files, which are archives containing the library and setup scripts and other information. Python searches each of these until it finds the module, whenever it encounters an **import** statement.

```
>>>for i in sys.path:  print i
...
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/distribute-0.6.34-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/numpy-1.7.0-py2.7-macosx-10.
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/pandas-0.10.1-py2.7-macosx-1
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/pytz-2012j-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/python_dateutil-2.1-py2.7.eg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/six-1.2.0-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/xlrd-0.9.0-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/xlwt-0.7.4-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/openpyxl-1.6.1-py2.7.egg
/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-tk
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-old
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages
/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/setuptools-0.6c11-py2.7.egg-
```

manipulating the module search path

The easiest way to make a modules available for **import** is to add its path directly to **sys.path**:

```
#!/usr/bin/env python

import sys
sys.path.append('/Users/dblaikie/lib')     # adding my lib directory to search path

import myclevermodule                      # located a /Users/dblaikie/lib/myclevermodule.py
```

However, if you have a permanent **lib** from which you're expecting to do imports, you should probably establish the **PYTHONPATH** environment variable and have this path added automatically whenever you run Python.

module names, module extensions

All Python programs can act as modules and vice versa - this is why our Python script may also be called a **module**. There are other Python files that we may encounter - it's useful to know them by their file extensions:
* **.py**: a Python script/module
* **.pyc**: a compiled module. Python creates this file when you import a module for the first time, and saves it in the same location as the module. Each subsequent time the module is imported, Python will use this file instead of the origional **.py**, which saves loading time. You won't see this file generated for your python script, only for imported modules.
* **.pyo**: an "optimized" script/module. This marks a script compiled with the **-o** flag - *assert* statements are disabled, which may or may not be present.

if __name__ == '__main__':

The working difference between a python *script* and a python *module* is that a **script** is run directly, and a **module** is imported. But this distinction depends only on how we use them - specifically, how we call them.

A script can *also* be a module, which would be the case if we wrote a script that contains functions that we'd like to use elsewhere. We can **import** the script and use its functions in our own code.

A module can also be a script, which would be the case if we wrote test code for the module, and placed it within the module.

We draw a distinction between *running* a script and *importing* a script using the **if** statement shown at the top of this slide. This code in the below example isn't very useful, but you can think of **return_int()** and **return_str()** as if they were useful library functions.

```
#!/usr/bin/env python

def return_int():
  return 5

def return_str():
  return 'hello'

def main():
  """run test suite"""
  if type(return_int()) is not int:
    print "ERROR:  return_int() did not return an int"
  else:
    print "return_int() test succeeded"
  if type(return_str()) is not str:
    print "ERROR:  return_str() did not return a str"
  else:
    print "return_str() test succeeded"

print "value of __name__:  " + __name__
if __name__ == '__main__':
  main()
```

If you save this script in a file called **test.py** and run it, you can see how the value of **__name__** differs when it is run (where its value is **__main__**) or when it is imported from another script (where its value is **test**).

Firstly, note the value of **__name__**. Then notice that only running the script runs the test suite. Importing the script makes **return_int()** and **return_str()** available, but doesn't run the test code, because of the **if** test.

The template and design for all of our programs

Our longer programs will be made up of many small functions, each of which does a small task. Well-designed programs are generally made this way.

In addition, we will want to use the **if __name__ == '__main__':** template when we create a new Python module / program. Here is a "Hello, World!" app using the most basic template, including that same function that we will now always include, called **main()**:

```
#!/usr/bin/env python   # points to python on your system

'''
Created on Feb 14, 2013
@author: David Blaikie
'''

def print_hello(arg):
  print "Hello, " + arg + "!"

def main():
  print_hello('World')

if __name__ == '__main__':
  main()
```

Since the **def** statements are actually just function definitions (and aren't run until they are called), the program actually begins executing with the 'if' statement. We'll discuss what this means later, but for now, just include it. Then, inside the 'if' block (i.e., indented underneath it), call **main()**. Thus our own code begins with **main()**.

As we consider the logic that will accomplish our objectives, it may be helpful to think in terms of functions that each do little things; and a **main()** that calls those functions.

installing modules: pip and easy_install

Many Python modules are not part of the default distribution and must be downloaded and installed. Although this work can be done manually, the programs **pip** and **easy_install** both can make a module installation nearly effortless.

On Unix or Mac, either program must be run with *root permissions*. This usually requires that you use the command **sudo** ahead of either **pip** or **easy_install**, but you must have **sudo** access and have a password. (If you own a mac and know the password for installing programs, this is usually the password you need for **sudo**.

Sample commands (note: you use *either* **pip** *or* **easy_install** to do an installation. At times one may be successful when the other isn't.)

Special note: **sudo** is needed for Unix installs because the installer must have root permissions. On Windows, this is not necessary. Simply omit the **sudo** part of these commands.

```
sudo pip search pandas            # searches for pandas in the PyPI repository
sudo pip install pandas           # installs pandas

sudo easy_install pandas          # installs pandas
```

Structuring and Sorting Complex Data

Multidimensional Structures: structuring data according to its nature

We have learned about how the nature and purpose of data can inform our choosing which of Python's data structures to use for holding the data in memory (a set for a unique set of values, a list for ordered sequence of values, etc.).

However, most data is more complex in structure than the basic containers will allow. Tabular data or result-set data is *2-dimensional* and can't be easily held within a list or dictionary. (A dict, being essentially a set of pairs, is in a sense 2-dimensional -- we can store for example names associated with IDs, but how would we store multiple columns?)

Fortunately, a Python container can hold any object, the same as it has been doing with integers and strings. A *list of lists* looks like this -- this is a list that holds four lists. Presumably this data came directly from a tabular source, like a database result set or text file.

```
lol = [
        [  '', 'US', 'Europe', 'Asia'],
        [  '2015-04-08', 23, 15, 103  ],
        [  '2015-04-09', 28, 7, 128 ],
        [  '2015-04-10', 22, 19, 107 ]
      ]
```

A *dictionary of dictionaries* looks like this -- this is a dict with two keys, each of which is associated with a dict of 3 keys and values. This would also have come from a tabular source, but it's far more conveniently organized -- a convenient dict of keys and values, each associated with an ID.

```
dod = {
         'db13':  {
                      'fname': 'Joe',
                      'lname': 'Wilson',
                      'tel':   '9172399895'
                  },
         'mm23':  {
                      'fname': 'Mary',
                      'lname': 'Doodle',
                      'tel':   '2122382923'
                  }
       }
```

We use these structures in order to better represent the relationships between elements in our data. Either one of these structures could have come from tabular data, but the nature of the data, and the way in which we'd like to use it, determine the design of the structure we use to pull into memory.

Multidimensional structures - lists within a list

Again, consider this list of lists, and note the syntax - four "inner" lists nested within a single "outer" list:

```
value_table =       [
                       [ 1, 2, 3 ],
                       [ 4, 5, 6 ],
                       [ 7, 8, 9 ],
                       [10, 11, 12 ]
                                  ]
```

The way to consider a structure like this is to consider that a single element of **value_table** is a list. (Literally, it is a list reference.) If we print the first element, we see a list.

```
print value_table[1]                 # [ 4, 5, 6 ]
```

If we want to access an element within the list, we can take a subscript of *that*:

```
print value_table[1][2]              # 6
```

As you can see in the last line, we can access the inner list simply by referencing it from **value_table[1]**. Thus any element within any of the lists is accessible through the outer list **value_table**.

Multidimensional structures - a dictionary of dictionaries

A dictionary holding other dictionaries offers another way to conveniently structure data.

```
date_values = {
  '19260701':   { 'MktRF':  0.09,
                  'SMB':   -0.22,
                  'HML':   -0.30,
                  'RF':    0.009 },
  '19260702':   { 'MktRF':  0.44,
                  'SMB':   -0.35,
                  'HML':   -0.08,
                  'RF':    0.009 },
}
```

In reading a single value from this structure, following the same train of thought as we did with the list of lists, we can associate an "inner" dictionary (by reference) to one of the keys of the "outer" dictionary. Then we can combine the two and access an "inner" value using a double subscript:

```
MktRF_thisday = date_values['19260701']['MktRF']   # value is 0.09

print date_values['19260701']['SMB']               # -0.22
print date_values['19260701']['HML']               # -0.30
```

As you can see, this "double-jointed" dictionary lets us look up any value simply by entering the date and the factor name into the subscripts. We are also enabled to take sums, averages, rolling sums and averages, etc. - anything we want.

We could also write to the structure using the same syntax:

```
date_values['19260701']['MktRF'] = 0.10
date_values['19260701']['HML'] += 1              # adding 1 to existing value
date_values['19260701']['new_key'] = 'new value'  # adding a brand new key to inner dict
                                                  # assocaited with '19260701'
```

Multidimensional structures - building

Now let's extrapolate from what we've learned in order to loop through and print out every element in a 2-dimensional structure. Going back to the list of lists, we'll first loop through each element in the "outer" structure, **value_table**:

```
value_table =       [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ], [10, 11, 12 ] ]

print value_table[0]       # [ 1, 2, 3 ]
print value_table[1]       # [ 4, 5, 6 ]

for this_list in value_table:
  print this_list               # prints [1, 2, 3], then [4, 5, 6], etc.
```

If **this_list** refers to each list in turn, then we can loop through each list as it comes up -- this calls for a loop within a loop:

```
for this_list in value_table:
  for el in this_list:
    print str(el) + ":",      # prints 1:2:3: then 4:5:6:, etc.
  print                       # adds an extra line between rows
```

As we loop through the "outer" list, we are assigning each "inner" list to a label (**this_list**); and then looping through **this_list** we assign each element in **this_list** to another label, **el** - and we can loop through the "inner" list using the label.

Usually, we don't initialize multi-dimensional structures within our code. Sometimes one will come to us, as with **dict.items()**, which returns a list of tuples. Database results also come as a list of tuples.

Most commonly, we will build a multi-dimensional structure of our own design based on the data we are trying to store. For example, we may use the Fama-French file to build a dictionary of lists - the key of the dictionary being the date, and the value being a 4-element list of the values for that date.

```
outer_dict = {}                                         # new dict
for line in open('F-F.txt').read().splitlines()[4:-3]:
  columns = line.split()                                # split each line into a list of string values
  date = columns[0]                                     # the first value is the date
  values = columns[1:]                                  # slice this list into a list of floating-point va
  outer_dict[date] = values                             # so values is a list, assigned as value to key da
                                                        # thus we have built a dictionary of lists (each k
```

Perhaps we want to be more selective - build the inner list inside the loop:

```
outer_dict = {}                                        # new dict
for line in open('F-F.txt').read().splitlines()[4:-3]:
  inner_list = []                                       # a new, empty 'inner' list
  columns = line.split()                                # split the line into a list of string values
  date = columns[0]                                     # the first value is the date
  inner_list.append(columns[1])                         # add the 1st numeric value after the date to the
  inner_list.append(columns[4])                         # add the 4th numeric value to the 'inner' list
  outer_dict[date] = inner_list                         # now assign this newly built 'inner' list as valu
```

Inside the loop we are creating a temporary, empty list; building it up; and then finally associating it with a key in the "outer" dictionary. The work inside the loop can be seen as "the life of an inner structure" - it is built and then added to the inner structure - and then the loop moves ahead one line in the file and does the work again with a new inner structure.

Sorting Multidimensional Structures

Having built multidimensional structures in various configurations, we should now learn how to sort them -- for example, to sort the keys in a dictionary of dictionaries by one of the values in the inner dictionary (in this instance, the last name):

```
def by_last_name(key):
    return dod[key]['lname']

dod = {
         'db13':  {
                     'fname': 'Joe',
                     'lname': 'Wilson',
                     'tel':   '9172399895'
                  },
         'mm23':  {
                     'fname': 'Mary',
                     'lname': 'Doodle',
                     'tel':   '2122382923'
                  }
      }

sorted_keys = sorted(dod, key=by_last_name)
print sorted_keys                              # ['mm23', 'db13']
```

The trick here will be to put together what we know about obtaining the value from an inner structure with what we have learned about custom sorting.

Sorting review

A quick review of sorting: recall how Python will perform a default sort (numeric or *ASCII*-betical) depending on the objects sorted. If we wish to modify this behavior, we can pass each element to a function named by the **key=** parameter:

```
mylist = ['Alpha', 'Gamma', 'episilon', 'beta', 'Delta']

print sorted(mylist)                    # ASCIIbetical sort
                                        # ['Alpha', 'Gamma', 'Delta', 'beta', 'epsilon']

mylist.sort()                           # sort mylist in-place

print sorted(mylist, key=str.lower)     # alphabetical sort
                                        # (lowercasing each item by telling Python to pass it
                                        # to str.lower)
```

```
                                          # ['Alpha', 'beta', 'Delta', 'epsilon', 'Gamma']

print sorted(mylist, key=len)             # sort by length
                                          # ['beta', 'Alpha', 'Gamma', 'Delta', 'epsilon']
```

Sorting review: sorting dictionary keys by value: dict.get

When we loop through a dict, we can loop through a list of *keys* (and use the keys to get values) or loop through *items*, a list of (key, value) tuple pairs. When sorting a dictionary by the values in it, we can also choose to sort **keys** or **items**.

To sort keys, **mydict.get** is called with each key - and **get** returns the associated value. So the keys of the dictionary are sorted by their values.

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_sorted_keys = sorted(mydict, key=mydict.get)
for i in mydict_sorted_keys:
    print "{0} = {1}".format(i, mydict[i])

                 ## z = 0
                 ## c = 1
                 ## b = 2
                 ## a = 5
```

Sorting dictionary items by value: operator.itemgetter

Recall that we can render a dictionary as a list of tuples with the **dict.items()** method:

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_items = mydict.items()                       # [(a, 5), (c, 1), (b, 2), (z, 0)]
```

To sort dictionary **items** by value, we need to sort each two-element tuple by its second element. The built-in module **operator.itemgetter** will return whatever element of a sequence we wish - in this way it is like a subscript, but in function format (so it can be called by the Python sorting algorithm).

```
import operator
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
mydict_items = mydict.items()                       # [(a, 5), (c, 1), (b, 2), (z, 0)]
mydict_items.sort(key=operator.itemgetter(1))
print mydict_items                                  # [(z, 0), (c, 1), (b, 2), (a, 5)]
for key, val in mydict_items:
    print "{0} = {1}".format(key, val)

                    ## z = 0
                    ## c = 1
                    ## b = 2
                    ## a = 5
```

The above can be conveniently combined with looping, effectively allowing us to loop through a "sorted" dict:

```
for key, val in sorted(mydict.items(), key=operator.itemgetter(1)):
    print "{0} = {1}".format(key, val)
```

Database results come as a list of tuples. Perhaps we want our results sorted in different ways, so we can store as a list of tuples and sort using **operator.itemgetter**. This example sorts by the third field, then by the second field (last name, then first name):

```
import operator
items =[ (123, 'Joe', 'Wilson', 35, 'mechanic'),
         (124, 'Sam', 'Jones', 22, 'mechanic'),
         (125, 'Pete', 'Jones', 40, 'mechanic'),
         (126, 'Irina', 'Bibi', 31, 'mechanic'),
       ]
items.sort(key=operator.itemgetter(2,1)) # sorts by last, first name
for this_pair in items:
  print "{0} {1}".format(this_pair[1], this_pair[2])

       ## Irina Bibi
       ## Pete Jones
       ## Sam Jones
       ## Joe Wilson
```

Multi-dimensional structures: sorting with custom function

Similar to **itemgetter**, we may want to sort a complex structure by some inner value - in the case of **itemgetter** we sorted a whole tuple by its third value. If we have a list of dicts to sort, we can use the custom sub to specify the sort value from inside each dict:

```
def by_dict_lname(this_dict):
  return this_dict['lname'].lower()

list_of_dicts = [
  { 'id': 123,
    'fname': 'Joe',
    'lname': 'Wilson',
  },
  { 'id': 124,
    'fname': 'Sam',
    'lname': 'Jones',
  },
  { 'id': 125,
    'fname': 'Pete',
    'lname': 'abbott',
  },
]
list_of_dicts.sort(key=by_dict_lname)      # custom sort function (above)
for this_dict in list_of_dicts:
  print "{0} {1}".format(this_dict['fname'], this_dict['lname'])

# Pete abbot
# Sam Jones
# Joe Wilson
```

So, although we are sorting dicts, our sub says "take this dictionary and sort by this inner element of the dictionary".

Power Tools: list comprehensions, lambdas, enumerate(), range()

Multi-dimensional structures: sorting with lambda custom function

Functions are useful but they require that we declare them separately, elsewhere in our code. A *lambda* is a function in a single statement, and can be placed in data structures or passed as arguments in function calls. The advantage here is that our function is used exactly where it is defined, and we don't have to maintain separate statements.

A common use of lambda is in sorting. The format for lambdas is lambda **arg**: **return_val**. Compare each pair of regular function and lambda, and note the argument and return val in each.

```
def by_lastname(name):
  fname, lname = name.split()
  return lname

names = [ 'Josh Peschko', 'Gabriel Feghali', 'Billy Woods', 'Arthur Fischer-Zernin' ]
sortednames = sorted(names, key=lambda name:  name.split()[1])


list_of_dicts = [
  { 'id': 123,
    'fname': 'Joe',
    'lname': 'Wilson',
  },
  { 'id': 124,
    'fname': 'Sam',
    'lname': 'Jones',
  },
  { 'id': 125,
    'fname': 'Pete',
    'lname': 'abbott',
  },
]
```

```
def by_dict_lname(this_dict):
  return this_dict['lname'].lower()

sortedlenstrs = sorted(list_of_dicts, key=lambda this_dict:  this_dict['lname'].lower())
```

In each, the label after **lambda** is the argument, and the expression that follows the colon is the return value. So in the first example, the lambda argument is **name**, and the lambda returns **name.split()[1]**. See how it behaves exactly like the regular function itself?

Again, what is the advantage of lambdas? They allow us to design our own functions which can be placed inline, where a named function would go. This is a convenience, not a necessity. **But** they are in common use, so they must be understood by any serious programmer.

Lambda expressions: breaking them down

Many people have complained that lambdas are hard to grok (absorb), but they're really very simple - they're just so short they're hard to read. Compare these two functions, both of which add/concatenate their arguments:

```
def addthese(x, y):
  return x + y

addthese2 = lambda x, y:  x + y

print addthese(5, 9)        # 14
print addthese2(5, 9)       # 14
```

The function definition and the lambda statement are equivalent - they both produce a function with the same functionality.

Lambda expression example: dict.get and operator.itemgetter

Here are our standard methods to sort a dictionary:

```
import operator
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
for key, val in sorted(mydict.items(), key=operator.itemgetter(1)):
    print "{0} = {1}".format(key, val)

for key in sorted(mydict, key=mydict.get):
    print "{0} = {1}".format(key, mydict[key])
```

Imagine we didn't have access to **dict.get** and **operator.itemgetter**. What could we do?

```
mydict = { 'a': 5, 'b': 2, 'c': 1, 'z': 0 }
for key, val in sorted(mydict.items(), key=lambda keyval:  keyval[1]):
    print "{0} = {1}".format(key, val)

for key in sorted(mydict, key=lambda key:  mydict[key]):
    print "{0} = {1}".format(key, mydict[key])
```

These lambdas do exactly what their built-in counterparts do:

in the case of **operator.itemgetter**, take a 2-element tuple as an argument and return the 2nd element

in the case of **dict.get**, take a key and return the associated value from the dict

List comprehensions: filtering a container's elements

A simple code facility, list comprehensions abbreviate simple loops into one line.

Consider this loop, which filters a list so that it contains only positive integer values:

```
myints = [0, -1, -5, 7, -33, 18, 19, 55, -100]
myposints = []
for el in myints:
  if el > 0:
    myposints.append(el)

print myposints                  # [7, 18, 19, 55]
```

This loop can be replaced with the following one-liner:

```
myposints = [ el for el in myints if el > 0 ]
```

See how the looping and test in the first loop are distilled into the one line? The first **el** is the element that will be added to **myposints** - list comprehensions automatically build new lists and return them when the looping is done.

The operation is the same, but the order of operations in the syntax is different:

```
# this is pseudo code
# target list = item for item in source list if test
```

Hmm, this makes a list comprehension less intuitive than a loop. However, once you learn how to read them, list comprehensions can actually be easier and quicker to read - primarily because they are on one line.

This is an example of a *filtering* list comprehension - it allows some, but not all, elements through to the new list.

List comprehensions: transforming a container's elements

Consider this loop, which doubles the value of each value in it:

```
nums = [1, 2, 3, 4, 5]
dblnums = []
for val in nums:
  dblnums.append(val*2)

print dblnums                         # [2, 4, 6, 8, 10]
```

This loop can be distilled into a list comprehension thusly:

```
dblnums = [ val * 2 for val in nums ]
```

This *transforming* list comprehension transforms each value in the source list before sending it to the target list:

```
# this is pseudo code
# target list = item transform for item in source list
```

We can of course combine filtering and transforming:

```
vals = [0, -1, -5, 7, -33, 18, 19, 55, -100]
doubled_pos_vals = [ i*2 for i in vals if i > 0 ]
print doubled_pos_vals                # [14, 36, 38, 110]
```

List comprehensions: examples

If they only replace simple loops that we already know how to do, why do we need list comprehensions? As mentioned, once you are comfortable with them, list comprehensions are much easier to read and comprehend than traditional loops. They say in one statement what loops need several statements to say - and reading multiple lines certainly takes more time and focus to understand.

Some common operations can also be accomplished in a single line. In this example, we produce a list of lines from a file, stripped of whitespace:

```
stripped_lines = [ i.rstrip() for i in open('FF_daily.txt').readlines() ]
```

Here, we're only interested in lines of a file that begin with the desired year (1972):

```
totals = [ i for i in open('FF_daily.txt').readlines() if i.startswith('1972') ]
```

If we want the MktRF values for our desired year, we could gather the bare amounts this way:

```
mktrf_vals = [ float(i.split()[1]) for i in open('FF_daily.txt').readlines() if i.startswith('1972') ]
```

And in fact we can do part of an earlier assignment in one line -- the sum of market ref values for a year:

```
mktrf_sum = sum([ float(i.split()[1]) for i in open('FF_daily.txt').readlines() if i.startswith('1972') ])
```

From experience I can tell you that familiarity with these forms make it very easy to construct and also to decode them very quickly - much more quickly than a 4-6 line loop.

List Comprehensions with Dictionaries

Remember that dictionaries can be expressed as a list of 2-element tuples, converted using **items()**. Such a list of 2-element tuples can be converted back to a dictionary with **dict()**:

```
mydict =  {'a': 5, 'b': 0, 'c': -3, 'd': 2, 'e': 1, 'f': 4}

my_items = mydict.items()      # my_items is now [('a',5), ('b',0), ('c',-3), ('d',2), ('e',1), ('f',4)]
mydict2 = dict(my_items)       # mydict2 is now   {'a':5,   'b':0,   'c':-3,   'd':2,   'e':1,   'f':4}
```

It becomes very easy to filter or transform a dictionary using this structure. Here, we're filtering a dictionary by value - accepting only those pairs whose value is larger than 0:

```
mydict = {'a': 5, 'b': 0, 'c': -3, 'd': 2, 'e': -22, 'f': 4}
filtered_dict = dict([ (i, j) for (i, j) in mydict.items() if j > 0 ])
```

Here we're switching the keys and values in a dictionary, and assigning the resulting dict back to **mydict**, thus seeming to change it in-place:

```
mydict = dict([ (j, i) for (i, j) in mydict.items() ])
```

The Python database module returns database results as tuples. Here we're pulling two of three values returned from each row and folding them into a dictionary.

```
# 'tuple_db_results' simulates what a database returns
tuple_db_results = [
  ('joe', 22, 'clerk'),
  ('pete', 34, 'salesman'),
  ('mary', 25, 'manager'),
]

names_jobs = dict([ (name, role) for name, age, role in tuple_db_results ])
```

range() and enumerate()

The **range()** function produces a *new list of consecutive integers* which can be used for counting:

```
intlist = range(10)

for val in intlist:
    print val,            # comma:  keep print output on one line
                          # 0 1 2 3 4 5 6 7 8 9


for val in range(5,10):
    print val,            # 5 6 7 8 9


print sum(range(100))     # sum up the values from 0 to 99
```

enumerate() takes any iterable (thing that can be looped over) and "marries" it to a range of integers, so you can keep a simultaneous count of something

```
mylist = ['a', 'b', 'c', 'd']

for count, element in enumerate(mylist):
    print "element {}:  {}".format(count, element)
```

This can be handy for example with a filehandle. Since we can loop over a file, we can also pass it to **enumerate()**, which would render a line number with each line:

```
fh = open('file.txt')

for count, line in enumerate(fh):
    print "line {}:  {}".format(count, line.rstrip())

    ## (stripping the line above for clean-looking output)
```

Regular Expressions: String Pattern Matching

<u>Text Matching: Why is it so Useful?</u>

Short answer: *searching*, *validation* and *extraction* of *formatted text*. Either we want to see whether a string contains the pattern we seek, or we want to pull selected information from the string.

In the case of *fixed-width* text, we have been able to use a **slice**.

```
line = '19340903  3.4 0.9'
year = line[0:4]                 # year == 1934
```

In the case of *delimited* text, we have been able to use **split()**

```
line = '19340903,3.4,0.9'
els = line.split(',')
yearmonthday = els[0]            # 193400903
MktRF = els[1]                   # 3.4
```

In the case of *formatted* text, there is no obvious way to do it.

```
# how would we extract 'Jun' from this string?
log_line = '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
```

We may be able to use **split()** and slicing in some combination to get what we want, but it would be awkward and time consuming. So we're going to learn how to use *regular expressions*.

<u>Preview: a complex example</u>

Just as an example to show you what we're doing, the following regex pattern could be used to pull 'Jun' from the log line:

```
import re

log_line = '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'

reg = re.search(r'\d{2,3}\.\d{2,3}\.\d{2,3}\.\d{2,3} - - \[\d\d\/(\w{3})\/\d{4}', log_line)

print reg.group(1)   # Jun
```

Reading from left to right, the pattern (shown in the **r''** string) says this: "2-3 digits, followed by a period, followed by 2-3 digits, followed by a period, followed by 2-3 digits, followed by a period, followed by 2-3 digits, followed by a space, dash, space, dash, followed by a square bracket, followed by 2 digits, followed by a forward slash, followed by 3 word characters (and this text grouped for extraction), followed by a slash, followed by 4 digit characters."

Now, that may seem complex. Many of our regexes will be much simpler, although this one isn't terribly unusual. But the power of this tool is in allowing us to describe a complex string in terms of the *pattern* of the text -- not the specific text -- and either pull parts of it out or make sure it matches the pattern we're looking for. This is the purpose of regular expressions.

<u>Python's *re* module and *re.search()*</u>

import re makes regular expressions available to us. Everything we do with regular expressions is done through **re**.

re.search() takes two arguments: the *pattern string* which is the regex pattern, and the string to be searched. It can be used in an **if** expression, and will evaluate to **True** if the pattern matched.

```
# weblog contains string lines like this:
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'

# script snippet:
for line in weblog.readlines():
  if re.search(r'~jjk265', line):
    print line                      # prints 2 of the above lines
```

not for negating a search

As with any **if** test, the test can be negated with **not**. Now we're saying "if this pattern does *not* match".

```
# again, a weblog:
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'

# script snippet:
for line in weblog.readlines():
  if re.search(r'~jjk265', line):
    print line                      # prints 1 of the above lines -- the one without jjk265
```

The raw string (r'')

The raw string is like a normal string, but it does not process *escapes*. An escaped character is one preceded by a backslash, that turns the combination into a special character. **\n** is the one we're famliar with - the escaped **n** converts to a *newline character*, which marks the end of a line in a multi-line string.

A raw string wouldn't process the escape, so **r'\n'** is literally a backslash followed by an n.

```
var = "\n"            # one character, a newline
var2 = r'\n'          # two cahracters, a backslash followed by an n
```

The re Beastiary

We call it a "bestiary" because it contains many strange animals. Each of these animals takes the shape of a special character (like $, ^, |); or a regular text character that has been *escaped* (like \w, \d, \s); or a combination of characters in a group (like {2,3}, [aeiou]). Our bestiary can be summarized thusly:

Type of Beast	Example Beasts
Anchor Characters and the Boundary Character	$, ^, \b
Character Classes	\w, \d, \s, \W, \S, \D
Custom Character Classes	[aeiou], [a-zA-Z]
The Wildcard	.
Quantifiers	+, *, ?
Custom Quantifiers	{2,3}, {2,}, {2}
Groupings	(parentheses groups)

Patterns can match anywhere, but must match on consecutive characters

Patterns can match any string of consecutive characters. A match can occur anywhere in the string.

```
import re

str1 = 'hello there'
str2 = 'why hello there'
```

```
str3 = 'hel lo'

if re.search(r'hello', str1):  print 'matched'   # matched
if re.search(r'hello', str2):  print 'matched'   # matched
if re.search(r'hello', str3):  print 'matched'   # does not match
```

Note that 'hello' matches at the start of the first string and the middle of the second string. But it doesn't match in the third string, even though all the characters we are looking for are there. This is because the space in **str3** is unaccounted for - always remember - matches take place on *consecutive characters*.

Anchors and Boundary

Our match may require that the search text appear at the beginning or end of a string. The anchor characters can require this.

This program lists only those files in the directory that end in '.txt':

```
import os, re
for filename in os.listdir(r'/path/to/directory'):
  if re.search(r'\.txt$', filename):     # look for '.txt' at end of filename
    print filename
```

This program prints all the lines in the file that don't begin with a hash mark:

```
for text_line in open(r'/path/to/file.py'):
  if not re.search(r'^#', text_line):        # look for '#' at start of filename
    print text_line
```

When they are used as anchors, we will always expect **^** to appear at the start of our pattern, and **$** to appear at the end.

Character Classes

A *character class* is a special regex entity that will match on any of a set of characters. The three built-in character classes are these:

[0-9]	(Digits, represented by \d)
[a-zA-Z0-9_]	(Word characters -- letters, numbers or underscores -- represented by \w)
[\n\t]	('Whitespace' characters -- spaces, newlines, or tabs -- represented by \s)

So a **\d** will match on a **5**, **9**, **3**, etc.; a **\w** will match on any of those, or on **a**, **Z**, **_** (underscore).

Keep in mind that although they match on any of several characters, a single instance of a character class matches on only one character. For example, a **\d** will match on a single number like '5', but it won't match on both characters in '55'. To match on 55, you could say **\d\d**.

Built-in Character Classes: digits

The **\d** character class matches on any digit. This example lists only those files with names formatted with a particular syntax -- YYYY-MM-DD.txt:

```
import re
dirlist = ('.', '..', '2010-12-15.txt', '2010-12-16.txt', 'testfile.txt')
for filename in dirlist:
  if re.search(r'^\d\d\d\d-\d\d-\d\d.txt$', filename):
    print filename
```

Here's another example, validation: this regex uses the pattern ^\d\d\d\d$ to check to see that the user entered a four-digit year:

```
import re
answer = raw_input("Enter your birth year in the form YYYY\n")
if re.search(r'^\d\d\d\d$', answer):
  print "Your birth year is ", answer
else:
  print "Sorry, that was not YYYY"
```

Built-in Character Classes: "word" characters

A word character casts a wider net: it will match on any number, letter or underscore.

In this example, we require the user to enter a username with any word characters:

```
username = raw_input()
if not re.search(r'^\w\w\w\w\w$', username):
  print "use five numbers, letters, or underscores\n"
```

As you can see, the anchors prevent the input from exceeding 5 characters.

Built-in Character Classes: spaces

A space character class matches on any of three characters: a space (' '), a newline ('\n') or a tab ('\t'). This program searches for a space anywhere in the string and if it finds it, the match is successful - which means the input isn't successful:

```
new_password = raw_input()
if re.search(r'\s', new_password):
  print "password must not contain spaces"
```

Note in particular that the regex pattern **\s** is not anchored anywhere. So the regex will match if a space occurs anywhere in the string.

You may also reflect that we treat spaces pretty roughly - always stripping them off. They always get in the way! And they're invisible, too, and still we feel the need to persecute them. What a nuisance.

Negative Character Classes

These are more aptly named *inverse* character classes - they match on anything that is *not* in the usual character set.

Not a digit: **\D**
So **\D** matches on letters, underscores, special characters - anything that is not a digit. This program checks for a non-digit in the user's account number:

```
account_number = raw_input()
if re.search(r'\D', account_number):
  print "account number must be all digits!"
```

Not a word character: **\W**
Here's a username checker, which simply looks for a non-word:

```
account_number = raw_input()
if re.search(r'\W', account_number):
  print "account number must be only letters, numbers, and underscores"
```

Not a space character: **\S**
These two regexes check for a non-space at the start and end of the string:

```
sentence = raw_input()
if re.search(r'^\S', sentence) and re.search(r'\S$', sentence):
  print "the sentence does not begin or end with a space, tab or newline."
```

Custom Character Classes

Consider this table of character classes and the list of characters they match on:

class designation	members
\d	[0123456789]

\w	[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOOPQRSTUVWXYZ0123456789_] or [a-zA-Z0-9_]
\s	[\t\d]

In fact, the bracketed ranges can be used to create our own character classes. We simply place members of the class within the brackets and use it in the same way we might use **\d** or the others.

A custom class can contain a *range* of characters. This example looks for letters only (there is no built-in class for letters):

```
import re
input = raw_input("please enter a username, starting with a letter:  ")
if not re.search(r'^[a-zA-Z]', input):
  exit("invalid user name entered")
```

This custom class [.,;:?!] matches on any one of these punctuation characters, and this example identifies single punctuation characters and removes them:

```
import re
text_line = 'Will I?  I will.  Today, tomorrow; yesterday and before that.'
for word in text_line.split():
  while re.search(r'[.,;:?! -]$', word):
    word = word[:-1]
  print word
```

Negative Custom Character Classes

Like **\S** for **\s**, the inverse character class matches on anything *not* in the list. It is designated with a carrot just inside the open bracket:

```
import re
for text_line in open('unknown_text.txt'):
  for word in text_line.split():
    while re.search(r'[^a-zA-Z]$', word):
      word = word[:-1]
    print word
```

It would be easy to confuse the carrot at the start of a string with the carrot at the start of a custom character class -- just keep in mind that one appears at the very start of the string, and the other at the start of the bracketed list.

The Wildcard (.)

The *ultimate* character class, it matches on every character *except* for a newline. (We might surmise this is because we are often working with line-oriented input, with pesky newlines at the end of every line. Not matching on them means we never have to worry about stripping or watching out for newlines.)

```
import re
username = raw_input()
if not re.match(r'^.....$', username):   # five dots here
  print "you can use any characters except newline, but there must \
  be five of them.\n"
```

Quantifiers: specifies how many to look for

A quantifier appears immediately after a character, character class, or grouping (coming up). It describes how many of the preceding characters there may be in our matched text.

We can say three digits (\d{3}), between 1 and 3 word characters (\w{1,3}), one or more letters [a-zA-Z]+, zero or more spaces (\s*), one or more x's (x+). Anything that matches on a character can be quantified.

```
+      : 1 or more
*      : 0 or more
?      : 0 or 1
{3,10} : between 3 and 10
```

In this example directory listing, we are interested only in files with the pattern **config_** followed by an integer of any size. We know that there could be a config_1.txt, a config_12.txt, or a config_120.txt. So, we simply specify "one or more digits":

```
import re
filenames = ['config_1.txt', 'config_10.txt', 'notthis.txt', '.', '..']
wanted_files = []
for file in filenames:
  if re.search(r'^config_\d+\.txt$', file):
    wanted_files.append(file)
```

Here, we validate user input to make sure it matches the pattern for valid NYU ID. The pattern for an NYU Net ID is: two or three letters followed by one or more numbers:

```
import re
input = raw_input("please enter your net id:  ")
if not re.search(r'^[A-Za-z]{2,3}\d+$', input):
  print "that is not valid NYU Net ID!"
```

A *simple* email address is one or more word characters followed by an @ sign, followed by a period, followed by 2-4 letters:

```
import re
email_address = raw_input()
if re.search(r'^\w+@\w+\.[A-Za-z]{2,}$', email_address):
  print "email address validated"
```

Of course email addresses can be more complicated than this - but for this exercise it works well.

flags: re.IGNORECASE

We can modify our matches with qualifiers called *flags*. The **re.IGNORECASE** flag will match any letters, whether upper or lowercase. In this example, extensions may be upper or lowercase - this file matcher doesn't care!

```
import re
dirlist = ('thisfile.jpg', 'thatfile.txt', 'otherfile.mpg', 'myfile.TXT')
for file in dirlist:
  if re.search(r'\.txt$', file, re.IGNORECASE):   #'.txt' or '.TXT'
    print file
```

The flag is passed as the third argument to **search**, and can also be passed to other **re** search methods.

re.search(), re.compile() and the compile object

re.search() is the one-step method we've been using to test matching. Actually, regex matching is done in two steps: *compiling* and *searching*. **re.search()** conveniently puts the two together.

In some cases, a pattern should be compiled first before matching begins. This would be useful if the pattern is to be matched on a great number of strings, as in this weblog example:

```
import re
access_log = '/home1/d/dbb212/public_html/python/examples/access_log'
weblog = open(access_log)
patternobj = re.compile(r'edg205')
for line in weblog.readlines():
  if patternobj.search(line):
    print line,
weblog.close()
```

The **pattern object** is returned from **re.compile**, and can then be called with **search**. Here we're calling **search** repeatedly, so it is likely more efficient to compile once and then search with the compiled object.

Grouping for Alternates: Vertical Bar

We can group several characters together with parentheses. The parentheses do not affect the match, but they do designate a part of the matched string to be handled later. We do this to allow for alternate matches, for quantifying a portion of the pattern, or to extract text.

Inside a group, the vertical bar can indicate allowable matches. In this example, a string will match on any of these words, and because of the anchors will not allow any other characters:

```
r'^(y|yes|yeah|yep|yup|yu-huh)$'   # matches any of these words
```

A group may indicate alternating choices within a larger pattern:

```
r'(John|John D.)\s+Rockefeller'
                           # matches John Rockefeller or John D. Rockefeller
```

Grouping for Quantifying

Another reason to group would be to quantify a sequence of the pattern. For example, we could search for the "John Rockefeller or John D. Rockefeller" pattern by making the 'W.' optional:

```
r'John\s+(D\.\s+)?Rockefeller'
```

Grouping for Extraction: Memory Variables

We use the **group()** method of the **match** object to extract the text that matched the group:

```
string1 = "Find the nyu id, like dbb212, in this sentence"
matchobj = re.search(r'([a-z]{2,3}\d+)', string1)
id = matchobj.group(1)                           # id is 'dbb212'
```

Here's an example, using our log file. What if we wanted to capture the last two numbers (the status code and the number of bytes served), and place the values into structures?

```
log_lines = [
'66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449',
'216.39.48.10 - - [09/Jun/2003:19:57:00 -0400] "GET /~rba203/about.html HTTP/1.1" 200 1566',
'216.39.48.10 - - [09/Jun/2003:19:57:16 -0400] "GET /~dd595/frame.htm HTTP/1.1" 400 1144'
]

import re
bytes_sum = 0
for line in log_lines:
  matchobj = re.search(r'(\d+) (\d+)$', line) # last two numbers in line
  status_code = matchobj.group(1)
  bytes = matchobj.group(2)
  bytes_sum += int(bytes)                     # sum the bytes
```

groups()

If you wish to grab all the matches into a tuple rather than call them by number, use **groups()**. You can then read variables from the tuple, or assign **groups()** to named variables, as in the last line below:

```
import re

name = "Richard M. Nixon"
matchobj = re.search(r'(\w+)\s+(\w)\.\s+(\w+)', name)
name_tuple = matchobj.groups()
print name_tuple    # ('Richard', 'M', 'Nixon')

(first, middle, last) = matchobj.groups()
print "%s, %s, %s" % ( first, middle, last )
```

findall() for multiple matches

findall() with a groupless pattern
Usually **re** tries to a match a pattern once – and after it finds the first match, it quits searching. But we may want to find as many matches as we can – and return the entire set of matches in a list. **findall()** lets us do that:

```
text = "There are seven words in this sentence";
words = re.findall(r'\w+', text)
print words  # ['There', 'are', 'seven', 'words', 'in', 'this', 'sentence']
```

This program prints each of the words on a separate line. The pattern **\b\w+\b** is applied again and again, each time to the text remaining after the last match. This pattern could be used as a word counting algorithm (we would count the elements in **words**), except for words with punctuation.

findall() with groups
When a match pattern contains more than one grouping, **findall** returns multiple tuples:

```
text = "High: 33, low: 17"
temp_tuples = re.findall(r'(\w+):\s+(\d+)', text)
print temp_tuples                       # [('High', '33'), ('low', '17')]
```

sub() for substitutions

Regular expressions are used for matching so that we may inspect text. But they can also be used for substitutions, meaning that they have the power to modify text as well:

```
string = "My name is David"
pattern = re.compile(r'David')
string = pattern.sub('John', string)

print string                            # 'My name is John'
```

matching on multi-line files

This example opens and reads a web page (which we might have retrieved with a module like **urlopen**), then looks to see if the word "advisory" appears in the text. If it does, it prints the page:

```
file = open('http://www.nws.noaa.gov/view/prodsByState.php?state=NY&prodtype=state')
text = file.read()
if re.search(r'advisory', text, re.I):
  print "weather advisory:  ", text
```

re.MULTILINE: ^ and $ can match at start or end of line

We have been working with text files primarily in a line-oriented (or, in database terminology, *record-oriented* way, and regexes are no exception - most file data is oriented in this way. However, it can be useful to dispense with looping and use regexes to match within an entire file - read into a string variable with **read()**.

In this example, we surely can use a loop and split() to get the info we want. But with a regex we can grab it straight from the file in one line:

```
# passwd file:
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false

# python script:
import re
passwd_text = open('/etc/passwd').read()
mobj =  re.search(r'^root:[^:]+:[^:]+:[^:]:([^:]+):([^:]+)', passwd_text, re.MULTILINE)
if mobj:
    info = mobj.groups()
    print "root:  Name %s, Home Dir %s" % (info[0], info[1])
```

We can even use findall to extract all the information rfrom a file - keep in mind, this is still being done in two lines:

```
import re
passwd_text = open('/etc/passwd').read()
lot = re.findall(r'^(\w+):[^:]+:[^:]+:[^:]+:[^:]+:([^:]+)', passwd_text, re.MULTILINE)

mydict = dict(lot)

print mydict
```

re.DOTALL -- allow the wildcard (.) to match on newlines

Normally, the wildcard doesn't match on newlines. When working with whole files, we may want to grab text that spans multiple lines, using a wildcard.

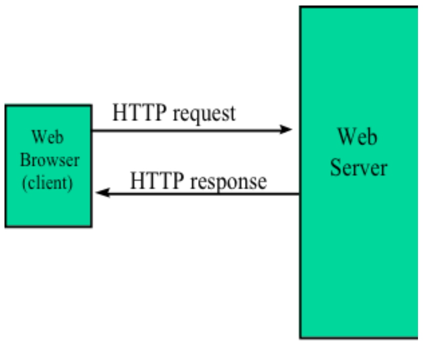
```
# search file sample.txt
some text we don't want
==start text==
this is some text that we do want.
the extracted text should continue,
including just about any character,
until we get to
==end text==
other text we don't want

# python script:
import re
text = open('sample.txt').read()
matchobj = re.search(r'==start text==(.+)==end text==', text, re.DOTALL)
print matchobj.group(1)
```

Retrieving data from the Web

The Client-Server Model

Any *web* transaction (i.e., when you visit a web page) involves the *HTTP*, or *HyperText Transport Protocol*. That's what the **http://** on your browser refers to.

An *HTTP Request* is initiated when you visit a link or submit a form from your **browser** (the *client*). The request is received by the *web server*, which is another program located on another computer on the network.



As clients and servers are only programs, we can write a script that works as either the client or server. We can write a Python script that takes the place of a web browser and downloads pages (which we can then read and parse using regular expressions), or one that works with the web server to receive requests and process information.

Evaluating Web Page Source from a Browser

All HTML web pages are downloaded as text. The text consists of HTML tags surrounding string content, page formatting information like CSS, references to images and ads, and other features. They also contain *Javascript*, a language that is during and after the page is loaded and can add functionality to the page.

To view the HTML, CSS and Javascript loaded on a page, wait for the page to complete loading and then find your browsers "View Source" or "Page Source" option:

On Firefox: **Tools > Web Developer > Page Source (or Cmd-U)**
On Chrome: **View > Developer > View Source (or Option-Cmd-U)**

(All browsers support this functionality - Google "View Page Source" to discover the method for other browsers, or in the event your version of the above browsers has located this feature differently.

Web Scraping

The HTML tags encase the text information on a page. But because relatively few tags are used to denote structure and most tags are used to specify display, it can be challenging to extract information based on them. For example, note below that **<h1>** tags appear to correctly delineate the name of the company on this page, but that the **<b>** tags appear in two places -- one which contains info we might want, and one that does not.

```
<html>
  <head>
    <title>Boomblerg:  Alphacorp Communications, Inc.</title>
  </head>
  <body>
    <h1>Alphacorp Communications, Inc.</h1>
    Alphacorp is a <i>New Hampshire</i> corporation.
    Its <b>gross revenues</b> for last year were <b>$2.6Bn.</b>
  </body>
</html>
```

Web scraping is the art of pulling info from pages based in part on style and formattting information. Since the author of the page can change the formatting at any time, it is inherently unreliable as a data format, which is why data taken from a page on an ongoing basis needs to be verified.

That's a lotta tags

Perhaps because HTML is designed to be very straightforward for beginners, elaborate formatting can cause commercial web pages to be extremely complex. Here for example is the page source of the New York Times Business section page.

Python as a web client: the urllib2 module

A Python program can take the place of a browser, requesting and downloading HTML pages and other files. Your Python program can work like a web spider (for example visiting every page on a website looking for particular data or compiling data from the site), can visit a page repeatedly to see if it has changed, and can visit a page once a day to compile information for that day, etc.

urllib2 is a full-featured module for making web requests. Although the **requests** module is strongly favored by some for its simplicity, it has not yet been added to the Python builtin distribution.

The **urlopen** method takes a url and returns a file-like object that can be **read()** as a file:

```
import urllib2
my_url = 'http://www.nytimes.com'
readobj = urllib2.urlopen(my_url)
text = readobj.read()
print text
readobj.close()
```

Alternatively, you can call **readlines()** on the object. (Many objects that deliver file-like string output can be read with **readlines()** and **read()**):

```
for line in readobj.readlines():
  print line
readobj.close()
```

curl: command-line web client

curl is a command-line client available on many Unix distributions, including on the Mac. It's an easy way to obtain the text of a web page using its URL.

```
curl http://www.nytimes.com > nytimes_home_source.html
```

As this utility writes to **STDOUT** its output is usually redirected to a file, as we're doing above with **nytimes_home_source.html** (a filename of my choosing).

On Windows, the **wget** command-line app provides similar functionality.

Evaluating Dynamic Web Page Source

Although most of the data displayed on a web page can be retrieved from a page's source, this can be tricky to retrieve and view if the data was loaded *dynamically*.

A *dynamic request* occurs when the page itself (more specifically, the page's javascript) makes *additional, asynchronous* requests to load data that thus isn't part of the original web page request.

Since dynamically loaded data is not part of the original page source, it needs to be found using a "developer tool" like the one built into the Chrome browser, or the **Firebug** add-on for Firefox.

Here's one procedure for finding dynamically loaded source using **Chrome**:

```
1. From the menu, select View > Developer > Developer Tools.
   A multi-pane window appears on one side of the browser window.
2. From the horizontal menu at the top of this window, choose Elements.
3. In the window directly beneath the horizontal window you will see a
   list of collapsable HTML elements.  As you mouse over the various
   elements you'll see the corresponding parts of the page highlighted.
   For example, if you mouse over the "body" tag you'll see the whole
   page highlighted.  (If you expand this element's contents by clicking
   on the arrow you can isolate the various other sections of the page
   as well.)
4. Mousing over "body" and seeing the entire body highlighted, right-
   click and select Copy > Copy OuterHTML.
5. Paste the copied text into a blank text document.
```

Recursion for traversing nested "nodes"

A *recursive function* is a function that calls itself. We use recursion in situations where we're dealing with "nested" data -- data elements that are linked to each other in a hierarchy, a structure that can be said to have "depth".

A good example of nested data are the files and directories in a filesystem. If our purpose is to list all the files and directories in a directory tree, we will begin by listing out the files and directories in the root directory. Then for each directory we encounter, we'll list out the files and directories there as well.

For each directory we encounter, therefore, the process is the same -- loop through and list the files. This suggests a *recursive*, or repeating, process, that will occur as many times as warranted by the data.

Below, **listtree()** takes a directory argument, lists the contents of the directory, and prints the name and size of each file it finds there. But when a directory is found, the function *calls itself* and passes the new directory as argument. (In this way its behavior is similar to that of **os.walk**, which most likely also uses recursion.)

```
import os

root_dir = '/Users/me/tree'

def listtree(this_dir):
    for item in os.listdir(this_dir):
        itempath = os.path.join(this_dir, item)
        if os.path.isfile(itempath):
            print "{}:  {}".format(itempath, os.path.getsize(itempath))
        elif os.path.isdir(itempath):
            listtree(itempath)
listtree(root_dir)
```

Here's another version of the same function, that compiles a full list of files and directories. Once the initial call to the function finally returns, it has compiled a full list of all items in tree:

```
def listtree(this_dir):
    import os
    file_list = []
    items = os.listdir(this_dir)
    for item in items:
        itemp = os.path.join(this_dir, item)
        if os.path.isfile(itemp):
            file_list.append(itemp)
        elif os.path.isdir(itemp):
            file_list.append(itemp)
            file_list.extend(listtree(itemp))
    return file_list

root_dir = '/Users/me/tree'

tree_list = listtree(root_dir)
print "{} items total".format(len(tree_list))

for item in tree_list:
    print item
```

Encoding and Decoding URL Parameters and HTML text

urllib2.urlencode() and **urllib2.urldecode()**

When including parameters in our requests, we must *encode* them into our request URL. The **urlencode()** method does this nicely:

```
import urllib
params = urllib.urlencode({'choice1': 'spam and eggs', 'choice2': 'spam, spam, bacon and spam'})
print "encoded query string: ", params
f = urllib.urlopen("http://i5.nyu.edu/~dbb212/cgi-bin/pparam.cgi?%s" % params)
print f.read()
```

this prints:

```
encoded query string: choice1=spam+and+eggs&choice2=spam%2C+spam%2C+bacon+and+spam

choice1:  spam and eggs<BR>
choice2:  spam, spam, bacon and spam<BR>
```

Encoding and Decoding display HTML
Occasionally we may find HTML text with "escaped" tags (as I noticed with some of the XLRB data). The tag characters (like < and > have been translated into HTML "entities" that need to be decoded.

```
&lt;html&gt;
  &lt;head&gt;
    &lt;title&gt;Boomblerg:  Alphacorp Communications, Inc.&lt;/title&gt;
  &lt;/head&gt;
  &lt;body&gt;
    &lt;h1&gt;Alphacorp Communications, Inc.&lt;/h1&gt;
    Alphacorp is a &lt;i&gt;New Hampshire&lt;/i&gt; corporation.
    Its &lt;b&gt;gross revenues&lt;/b&gt; for last year were &lt;b&gt;$2.6Bn.&lt;/b&gt;
  &lt;/body&gt;
&lt;/html&gt;
```

We can decode this data using the **HTMLParser** library.

```
import HTMLParser
h = HTMLParser.HTMLParser()
print h.unescape('&pound;682m')    # £682m
```

Decoding the above example renders:

```
<html>
  <head>
    <title>Boomblerg: Alphacorp Communications, Inc.</title>
  </head>
  <body>
    <h1>Alphacorp Communications, Inc.</h1>
    Alphacorp is a <i>New Hampshire</i> corporation.
    Its <b>gross revenues</b> for last year were <b>$2.6Bn.</b>
  </body>
</html>
```

Sidebar: Sending email with smtplib

Sending mail is simple when you have an SMTP server running and available on your host computer. Python's **smtplib** module makes this easy:

```
#!/usr/bin/env python

# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Create a text/plain message formatted for email
msg = MIMEText('Hello, email.')

from_address = 'dbb212@nyu.edu'
to_address = 'david.beddoe@gmail.com'
subject = 'Test message from a Python script'

msg['Subject'] = subject
msg['From'] =    from_address
msg['To'] =      to_address

s = smtplib.SMTP('localhost')
s.sendmail(from_address, [to_address], msg.as_string())
s.quit()
```

Parsing Nested Data: HTML, XBLR, etc.

Web Scraping with Regexes

Since **urllib2.urlopen()** delivers text, we can easily run a regex to retrieve the information we want. This script uses a simple regex to extract all of the links to all of the articles on the home page of the NY Times:

```
import re
import urllib
my_url = 'http://www.nytimes.com'
readobj = urllib.urlopen(my_url)
text = readobj.read()
readobj.close()

# the current date should be substituted for 2012\/03\/25
urls = re.findall(r'http:\/\/www.nytimes.com\/2013\/03\/25[^>]+', text)

for url in urls:
  print match
```

HTML and XML Parsing with Beautiful Soup

Beautiful Soup is a Python module/library for extracting data from HTML and XML files. It transforms a complex HTML document into a complex "parse tree" of Python objects, and allows navigating, searching, and modifying the tree. It commonly saves programmers hours or days of work.

The **python-xbrl** module uses Beautiful Soup extensively, so what we learn here will be applicable in parsing XBRL documents as well.

This overview covers the most common approaches to reading documents. The two most useful objects in Beautiful Soup are the **BeautifulSoup** object (which represents the entire document) and the **tag** object, which represents any given tag.

(Examples and some text in this section are adapted from the Beautiful Soup documentation.)

Consider this simple document, **dormouse.html**:

```
<html>
    <head>
        <title>Alice's Adventures in Wonderland:  the Dormouse's Story</title>
    </head>
    <body>
        <p class="title heading">The <b>Dormouse's</b> Story</p>
        <p class="story">
            Once upon a time there were three little sisters; and their names were
            <a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
            <a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
            <a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
            and they lived at the bottom of a well.
        </p>
        <p class="story">Elsie, Lacie and Tillie were very happy.  No problems.</p>
        <p class="story" id="last"><a href="http://example.com/end"><b>The End?</b></a></p>
    </body>
</html>
```

Many HTML tags are *containers* that actually consist of *two* tags: a *start tag* and an *end tag*. Container tags are usually *nested* within each other. Thus an HTML document comprises a *hierarchy* of tags. Each tag within an HTML document can be considered a *node* within the hierarchy.

Any given tag can have *children*, or other tags placed within its begin and end tags; as well as *attributes*, which are key/value pairs that appear in the tag's begin tag.

The purpose of Beautiful Soup is to parse the document and maintain its hierarchical structure, while allowing for searching and collecting of data across nodes.

Parsing a Beautiful Soup document

Beautiful Soup reads an HTML document passed as an open **file** object, or a string (as might be returned from the **file read()** method).

```
import bs4

# pass a filehandle
soup = bs4.BeautifulSoup(open("dormouse.html"), 'html.parser')

# alt:  pass a string
# soup = bs4.BeautifulSoup(open("dormouse.html").read(), 'html.parser')
```

In these examples, **soup** is a Beautiful Soup **object**, which we will use to search and read the document.

We can easily print such a document in an easy-to-read or "pretty" format:

```
print soup.prettify()             # prints doc out in a format similar to sample above
```

Finding a tag by name

The **BeautifulSoup** object's attributes can easily access the *first* tag found with a particular name. The tag is a **BeautifulSoup tag** object, which also contains attributes that reveal the information in the tag.

The first **<p>** tag looks like this:

```
<p class="title heading">The <b>Dormouse's</b> Story</p>
```

The below code isolates and then queries it:

```
ptag = soup.p            # the first <p> tag in the document
print type(ptag)         # <class 'bs4.element.Tag'>
```

Reading a tag

All tags can be queried for the various values contained within it: the tag name itself, the text between the start and end tag, the key=value pairs (called attributes).

```
<p class="title heading" id="myid" >The <b>Dormouse's</b> Story</p>
```

We can retrieve these values using the following syntaxes:

```
print ptag.name           # p
print ptag.text           # The Dormouse's story
print ptag.attrs          # {u'class': [u'title', u'heading'], u'id': [u'myid']}
print ptag['id']          # [u'myid']
print ptag.get('id')      # [u'myid']
print ptag['class']       # [u'title', u'heading']
print ptag.get('class')   # [u'title', u'heading']
```

find_all() and find()

The **BeautifulSoup** object's **find_all()** method returns a list of *all* tags with that name:

```
ptags = soup.find_all('p')

print ptags

[
  <p class="title heading">The <b> </b> Story</p>,
  <p class="story">\n            Once upon a time there were three little sisters; and their names were\n
  <p class="story">Elsie, Lacie and Tillie were very happy.  No problems.</p>,
  <p class="story" id="last"><a href="http://example.com/end"><b>The End?</b></a></p>
]
```

find() works similarly, finding just the first match:

```
ptag = soup.find('p')   # finds the first <p> tag, same as soup.p, above
```

find() and **find_all()** support several ways of filtering for / searching for tags:

a string: as above: tag(s) with the specified name

```
ptag = soup.find('p')
```

a regular expression: tag(s) whose name matches the pattern

```
btags = soup.find_all(re.compile('^b'))
print btags
```

a list: find tags that match any of the strings

```
soup.find_all(["a", "b"])
print btags
```

True: find any/all tags

```
all_tags = soup.find_all(True)
print all_tags
```

a function: call the function with each tag as an argument. If returns **True**, include the tag

```
def has_class_but_no_id(tag):
    if tag.get('class') and not tag.get('id'):  # if tag has a class= attribute but not an id= attribute
        return True
    return False

cbni = soup.find_all(has_class_but_no_id)
print cbni
```

Here's another example, using a lambda. Consider this tag again:

```
<p class="title heading" id="myid" >The <b>Dormouse's</b> Story</p>
```

To select a tag like this based on its attributes, we can specify the values we expect (indenting here is done for clarity):

```
cb = soup.find_all(lambda tag: tag.name == 'p'
                               and tag.get('class') == ['title', 'heading']
                               and tag.get('id') == ['myid'])
```

The use of **get()** saves us from asking for an attribute that doesn't exist, very similar to a dictionary. Note well that the value for an attribute name is always a list, that the list will have one string element for attributes with only one value, such as **id="myid"**, but will have multiple string elements for those attributes that have more than one value (as noted in the example above).

keyword argument: another approach to select tags with a specified attribute -- can also be **True** (if the tag is present), **string** (if the tag has the specified value), **regex** (if pattern matches on tag's value); or **function** (if the tag's value passed to the function returns **True**

```
link_tags = soup.find_all(id=lambda x: x.startswith('link'))
print link_tags
```

All tag calls apply to all tags "beneath" the current tag

In our examples above, we read attributes and called methods on the **Beautiful Soup** object, which represents the entire document, so the entire document was parsed.

However, any tag may be queried in the same way as the main object. Consider this **<p>** tag, which has nested tags **<a>** and **<b>**:

```
        <p class="story" id="last"><a href="http://example.com/end"><b>The End?</b></a></p>
```

Like the Beautiful Soup object we called **soup**, the **<p>** responds to queries regarding itself (and tags nested beneath it):

```
last_ptag = soup.find('p', id='last')  # find the above <p> tag

a_tag = last_ptag.find('a')  # <a href="http://example.com/end"><b>The End?</b></a>
b_tag = last_ptag.find('b')  # <b>The End?</b>
```

Tag Hierarchy Navigation

Besides its search capabilities, Beautiful Soup can also allow navigation through a tag hierarchy in cases where a tag must be located by relative position to other tags:

addressing tags by nesting

```
print soup.body.p.b       # <b>Doormouse's</b>
```

moving upwards to a parent

```
ptag = soup.body.p

print ptag.parent.name    # body
```

looping through tags within a tag

```
for tag in soup.body:
    print tag             # prints out the 4 <p> tags within <body>
    print '======'
```

looping through "parent" tags

```
ptag = soup.body.p                # prints all ancestors recursively
for tag in ptag.parents:
    print tag.name
```

A "sibling" tag is one that sits at the same "level" with another tag:

```
<body>
  <p>Brother to my brother</p>
  <p>Sister to my sister</p>
</body>
```

accessing sibling tags (to "right" and "left")

```
ltag = soup.body.p
rtag = ltag.next_element              # moving right among siblings
print rtag.string                     # Sister to my sister

lltag = rtag.previous_element         # moving left among siblings
print lltag.string                    # Brother to my brother
```

looping through all strings beneath a tag

```
for string in soup.body.strings:
    print repr(string)              # repr() shows us a 'literal' representation of a string
```

XBRL Format

XBRL (eXtensible Business Reporting Language) is a freely available and global standard for exchanging business information. XBRL allows the expression of semantic meaning commonly required in business reporting.

XBRL is a standards-based way to communicate and exchange business information between business systems.

XBRL was originally the work of the AICPA, created as a way to advance financial reporting and facilitate the global exchange of financial data. XBRL International was created to ensure compatibility and agreement among industry members and to promote adoption of the standard around the world.

XBRL on Wikipedia

XBRL detailed spec

XBRL viewer from SEC

Slideshows about XBRL parsing

Download and process directly from EDGAR

XBRL: Introduction and Overview

Parsing .xbrl Documents with the Python xbrl module

```
from xbrl import XBRLParser, GAAPSerializer
import pdb

aapl_10k = 'documents/xbrl/apple/aapl-20150926.xml'
alle_10k = 'documents/xbrl/allegion/alle-20141231.xml'

for tenK in [aapl_10k, alle_10k]:
    xbrl_parser = XBRLParser()
    parsed_file = xbrl_parser.parse(file(tenK))
    print 'name:  ' + tenK

    # print parsed_file.prettify()

    # parse using the unique parser
    # parsed_obj = xbrl_parser.parse_unique(parsed_file)
    # pdb.set_trace()

    # parse using the GAAP Parser
    gaap_obj = xbrl_parser.parseGAAP(parsed_file,
                                     doc_date="2015-09-26",
                                     doc_type="10-K",
                                     context="360",
                                     ignore_errors=0)

    # 'serialize' means to convert to a form that can be saved
    serialized = GAAPSerializer(gaap_obj)
    data = serialized.data
    for key, val in data.items():
        print '{}: {}'.format(key, val)
        print

    print
    print
```

XBRL docs

Lightweight data storage with JSON

JSON (JavaScript Object Notation) is a simple "data interchange" format for sending structured data through text.

Structured simply means that the data is organized into standard programmatic containers (lists and dictionaries). In fact, JSON uses the same notation as Python (and vice versa) so it is immediately recognizable to us.

Here is some simple JSON with an arbitrary structure, saved into a file called **mystruct.json**:

```
{
   "key1":  ["a", "b", "c"],
   "key2":  {
               "innerkey1": 5,
               "innerkey2": "woah"
            },
   "key3":  55.09,
   "key4":  "hello"
}
```

Initializing a Python structure read from JSON
We can load this structure from a file read from a string:

```
fh = open('mystruct.json')
mys = json.load(fh)           # load from a file
fh.close()

fh = open('mystruct.json')
file_text = fh.read()

mys = json.loads(file_text)   # load from a string
```

```
fh.close()

print mys['key2']['innerkey2']   # woah
```

Note: although it resembles Python structures, JSON notation is slightly less forgiving than Python -- for example, double quotes are required around strings, and no trailing comma is allowed after the last element in a dict or list (Python allows this).

For example, I added a comma to the end of the outer dict in the example above:

```
  "key4":  "hello",
```

When I then tried to load it, the **json** module complained with a helpfully correct location:

```
ValueError: Expecting property name: line 9 column 1 (char 164)
```

Dumping a Python structure to JSON

json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])

json.dump(['streaming API'], io)

CSV

Although we can often use **string split.(',')** to parse a CSV file, the csv standard is more complex than just comma delimeters - for example, if a value itself has a comma, it calls for quotes to be placed around the value.

```
import csv
fh = open('some.csv')
reader = csv.reader(fh)

for record in reader:
    print "field 1: ", record[0], " field 2: ", record[1], " field 3: ", record[2]
```

You can configure **csv** to read a different "dialect", for example Excel-generated **.csv** files, colon-or-other delimited files, etc.

Writing is similarly easy:

```
import csv
wfh = open('some.csv', 'w')
writer = csv.writer(wfh, dialect='excel')        # dialect is not required
writer.writerow(['some', 'values', "boy, don't you like long field values?"])
writer.writerows([['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']])
wfh.close()
```

python object serialization with 'pickle'

Python has modules that can *serialize* (save on disk) Python structures directly. This works very similarly to **JSON** encoding except that a pickle file is not human-readable.

The process is simply to pass the structure to pickle for dumping or loading. Like

```
import pickle
br = [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
F = open('datafile.pickle', 'w')
pickle.dump(br, F)
F.close()

## later
G = open('datafile.txt')
J = pickle.load(G)
print J
    #  [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
```

This code simply pickles and then unpickles the br data structure. It shows that the structure is stored in a file and can be retrieved from it and brought back into the code using pickle.

The other main difference between **pickle** and **json** is that you can store objects of any type. You can even pickle custom objects, although the class to which they belong must be imported first.

To serialize functions, classes, modules, etc. without any prior importing, it is possible to serialize Python into bytecode strings using the **marshal** module. There are certain caveats regarding this module's implementation; it is possible that marshal will not work passing bytecode between different Python versions.

Leveraging Module Code

Something's Not Quite Right...

After having some questions and noticing some irregularities in the **xbrl** module, I decided to investigate.

Here is how we used **xbrl**:

```
from xbrl import XBRLParser, parseGAAP

aapl_10k = 'documents/xbrl/apple/aapl-20150926.xml'

xbrl_parser = XBRLParser()
parsed_file = xbrl_parser.parse(open(aapl_10k))

# parse using the GAAP Parser
gaap_obj = parseGAAP(parsed_file,
                     doc_date="2015-09-26",  # date used for ... ?
                     doc_type="10-K",        # this doesn't seem to
                                             # affect behavior
                     context="360",
                     ignore_errors=0)

# parse using the 'unique' parser
# parsed_obj = xbrl_parser.parse_unique(parsed_file)  # this object is None!

serialized = GAAPSerializer(gaap_obj)
data = serialized.data
for key, val in data.items():
    print '{}: {}'.format(key, val)
    print
```

To investigate, I wanted to read the module code. I located the file using the module's magic **__file__** attribute.

Object attributes and dir()

All objects have *attributes*, which we access using the same syntax we have been using throughout this course:

```
object.attribute
```

Most often we have been accessing *callable* attributes, which we know as **methods**.:

```
mystr = 'hello'
print mystr.upper()
```

Attributes can be any type of value. Methods are functions associated with an object. But the **sys.copyright** attribute is a string:

```
import sys
print sys.copyright      # a string

## Copyright (c) 2001-2015 Python Software
## Foundation.
## All Rights Reserved.
```

The **dir()** function can show us a list of an object's attributes. We can thus *inspect* any object to learn more about it.

```
>>> var = 'hello'
>>> dir(var)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

hasattr() to check for the existence of an attribute in an object

There are times when we need to check to see if an attribute exists in an object. For this we can use a built-in function called **hasattr()**:

```
obj = 'hello'
if hasattr(obj, 'upper'):
    print obj.upper
else:
    print 'cannot uppercase a non-string'
```

Finding the location of a module with its __file__ attribute

Magic or *private attributes* are ones that are not usually used directly. They are used implicitly when we perform certain operations (such as the string **__add__** method, which is called implicitly when we use the **+** operator).

A module's **__file__** attribute can tell us its location within our filesystem:

```
>>> import xbrl
>>> xbrl.__file__
'/usr/lib/python2.7/site-packages/xbrl/__init__.pyc'
```

__file__ tells us that it executes **xbrl** by importing **__init__.pyc**. **__init__** is another magic method, that refers to a special *package file*, which is launched first when the module is imported.

The **.pyc** extension is a special compiled version of the file; it can be ignored and the **.py** of the same name, **__init__.py**, can be used.

A module sometimes consists of multiple files, sometimes in multiple directories, that we call a *package*. We can view the files by looking at the directory specified in the **__file__** attribute.

```
$ ls /usr/lib/python2.7/site-packages/xbrl/
__init__.py     __init__.pyc    gaap.py         gaap.pyc        xbrl.py         xbrl.pyc
$
```

These files are python code, essentially the same as any other Python file we might write or read.

The xbrl class
Here again is how we started our work with **xbrl**:

```
from xbrl import XBRLParser()
xbrl_parser = XBRLParser()
```

As we said from the start, an *object* of a particular *type* (**str**, **list**, etc.) can also be said to be of a particular *class* (string class, list class, etc.)

Therefore the object we're working with is an **XBRLParser** object.

Locating the **xbrl** module and reading its code, we find the following elements:

class definition and __init__() method
The class is defined a little like a function **def** -- only it uses a **class** declaration:

```
class XBRLParser(object):

    def __init__(self, precision=0, ignore_errors=False):
        self.precision = precision
        self.ignore_errors = ignore_errors
```

The **__init__()** method is a "magic" method, one that is called implicitly when a new object of this class is created.

At this stage there isn't a lot we need to know about the **class** definition other than it defines how the object will behave by defining its methods.

methods
Within the class definition we find **function defs** -- these are the **methods** of objects of this type. Here is the start of the **parse()** method:

```
    @classmethod
    def parse(self, file_handle):
        """
        parse is the main entry point for an XBRLParser. It takes a file
        handle.
        """
```

The first argument, **self**, is the first argument in all object methods. It is the object itself, the **xbrl** object upon which we called the method:

```
parsed_file = xbrl_parser.parse(open(aapl_10k))
```

The second argument to **parse()** is the open filehandle to read from, which you can see in the method definition.

A more complicated module exploration: Beautiful Soup

Beautiful Soup's **__file__** attribute tells us its location within our filesystem:

```
>>> import bs4
>>> bs4.__file__
'/usr/lib/python2.7/site-packages/bs4/__init__.pyc'
```

In the above example, **__init__** refers to a special *package file*, which is launched first when the module is imported. The **.pyc** extension is a special compiled version of the file; it can be ignored and the **.py** of the same name, **__init__.py**, can be used.

A module sometimes consists of multiple files, sometimes in multiple directories, that we call a *package*. We can view the files by looking at the directory specified in the **__file__** attribute.

```
$ ls /usr/lib/python2.7/site-packages/bs4/
builder     dammit.pyo   element.pyo   __init__.pyo  testing.pyo
dammit.py   element.py   __init__.py   testing.py    tests
dammit.pyc  element.pyc  __init__.pyc  testing.pyc
$
```

We can read the **./py** files at our pleasure. We win again.

The BeautifulSoup class
As we said, an *object* of a particular *type* (**str**, **list**, etc.) can also be said to be of a particular *class* (string class, list class, etc.)

Here is how we used the class **BeautifulSoup**:

```
import bs4

soup = bs4.BeautifulSoup(open('myfile.html'))

ptags = soup.find_all('p')   # extract all <p> tags
```

Locating the **Beautiful Soup** module and reading its code, we find the following elements:

class definition
The class is defined a little like a function **def** -- only using a **class** declaration:

```
class BeautifulSoup(Tag):   # inherits from the Tag class
    """
    This class defines the basic interface called by the tree builders.

    ... (module continues ...

    """
    ROOT_TAG_NAME = u'[document]'

    # If the end-user gives no indication which tree builder they
    # want, look for one with these features.
    DEFAULT_BUILDER_FEATURES = ['html', 'fast']
```

At this stage there isn't a lot we need to know about the **class** definition other than it defines exactly how the object will behave.

methods
Within the class definition we find **function defs** -- these are the **methods** of objects of this type. Here is the start of the **find_all()** method:

```
    def find_all(self, name=None, attrs={}, recursive=True, text=None,
                 limit=None, **kwargs):
        """Extracts a list of Tag objects that match the given
        criteria.  You can specify the name of the Tag and any
        attributes you want the Tag to have.
```

The first argument, **self**, is the first argument in all object methods. It is the object itself, the **BeautifulSoup** object upon which we called the method:

```
ptags = soup.find_all('p')   # extract all <p> tags
```

However, locating the **find_all()** method was a little more complicated, as the method was found in the **Tag** class. As mentioned above, the **BeautifulSoup** class *inherits* from the **Tag** class. Inheritance allows the **BeautifulSoup** object to access any method found in **Tag**. To find it, I had to locate the **Tag** class in the **element.py** file within the file hierarchy. I did this using the Unix utility **grep**, which searches a file or files for a string pattern. I executed the below command from the root folder of the Beautiful Soup documentation.

```
$ grep -R 'class Tag' *
element.py:class Tag(PageElement):
```

This showed that the **class Tag** class definition could be found in the **element.py** file.

__init__() constructor

The **__init__()** method is a "magic" method, one that is called implicitly when a new object of this class is created.

Again, this is the code we have used to create a new Beautiful Soup object:

```
soup = bs4.BeautifulSoup(open("dormouse.html"), 'html.parser')
```

In the Beautiful Soup class, the **__init__** method looks lik this:

```
    def __init__(self, markup="", features=None, builder=None,
                 parse_only=None, from_encoding=None, **kwargs):
        """The Soup object is initialized as the 'root tag', and the
        provided markup (which can be a string or a file-like object)
        is fed into the underlying parser."""
```

We can conclude that the open filehandle in the call will be the **markup** variable, and the **html.parser** object will be the **features** variable.

Docstrings

A *docstring* is a Python string used for documentation. These strings are embedded within Python scripts and can be read by *docstring readers* to build documentation.

```
#!/usr/bin/env python

"""myprog.py:  show a sample of docstrings"""

def hello():
    """greet the user"""
    print 'yahoo!'

def main():
    """primary function and starting point"""
    myfunc()

if __name__ == '__main__':
    main()
```

What makes the above bolded strings docstrings is their position. Note that each one is one of the following:

- the first string in the script / module
- the first string inside a function
- the first string inside a class

PEP257 defines the overall spec for docstrings.

Python and docwriters are aware of docstrings

The presence of this string *as the very first statement* in the module/class/function/method populates the **__doc__** attribute of that object.

```
#!/usr/bin/env python

"""myprog.py:  show a sample of docstrings"""

def myfunc():
    """this is myfunc"""

print __doc__                # 'myprog.py:  show a sample of docstrings'
print myfunc.__doc__         # 'this is myfunc'
```

All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the __init__ constructor) should also have docstrings. A package may be documented in the module docstring of the __init__.py file in the package directory.

It is also allowable to add a docstring after each variable assignment inside a module, function or class. Everything except for the format of the initial docstring (which came from a module I wrote for AppNexus) is an example from PEP257:

```
"""
NAME
    populate_account_tables.py

DESCRIPTION
    select data from SalesForce, vertica and mysql
    make selected changes
    insert into fiba mysql tables

SYNOPSIS
    populate_account_tables.py
    populate_account_tables.py table1 table2     # process some tables, not others

VERSION
    2.0 (using Casserole)

AUTHOR
```

```
    David Blaikie (dblaikie@appnexus.com)
"""

num_tried = 3
"""Number of times to try the database before failing out."""

class Process(object):

    """ Class to process data. """

    c = 'class attribute'
    """This is Process.c's docstring."""

    def __init__(self):
        """Method __init__'s docstring."""

        self.i = 'instance attribute'
        """This is self.i's docstring."""

def split(x, delim):
    """Split strings according to a supplied delimeter."""

    splitlist = x.split(delim)
    """the resulting list to be returned"""

    return splitlist
```

These docstrings can be automatically read and formatted by a docstring reader. **docutils** is a built-in processer for docs, although the documentation for *this* module is a bit involved. We'll discuss the more-easy-to-use **Sphinx** next.

Sphinx: easy docs from docstrings

In order to generate docs for a project, you must set up a separate Sphinx project, which includes a config file and a build file. The Sphinx docs are a bit obscure, but this tutorial covers the steps pretty clearly.

Code Versioning and Maintenance

git: version control system

Version control refers to a system that keeps track of all changes to a set of documents (i.e., the revisions, or *versions*). Its principal advantages are: 1) the ability to restore a team's codebase to any point in history, and 2) the ability to *branch* a parallel codebase in order to do development without disturbing the **main** branch, and then *merge* a development branch back into the **main** branch.

Git was developed by Guido van Rossum to create an open-source version of **BitKeeper**, which is a distributed, lightweight and very fast VCS. Other popular systems include **CSV**, **SVN** and **ClearCase**.

Git's docs are very clear and extensive. Here is a quick rundown of the highlights:

Creating or cloning a new repository
- **git init** to create a brand-new repository
- **git clone** to create a local copy of a repository on the remote server (a *github*)

Adding, changing and committing files to the local repository
- **git add** to add a new (*or changed*) file to the *staging area*
- **git status** to see what files have been changed but not added, or changed and added
- **git diff** to see what changes have been made but not committed, or changes between commits
- **git commit** to commit the added changes to the local repository

Reviewing commit history with git log

- **git log** to review individual commits

pylint

Lint refers to little aberrations in your coding style or usage that don't break the code, but can make it more challenging to read or maintain. A *linter* points out these issues so we can create clearer, more compliant code.

Quite honestly it might be easier to use an IDE like **PyCharm to point out code inconsistencies, because it can tell us about them on the fly.**

pylint is a command-line app that reads your code and produces a report of style and usage issues.

Basic Usage

```
$ pylint [options] module_or_package_to_be_checked
```

Pylint features a set of *checkers* which can be turned on or off (all are enabled by default). As Pylint is considered to be pretty thorough (read: picky), you may find that certain types of checks are unwelcome. You can specify which to turn on or off in [options] above, as specified in examples below.

To disable a particular checker:

```
--disable=,...
```

To disable most and enable just a few checkers:

```
--disable=all --enable=,...
```

Notable Options

Here are some basic options I found useful.

display report	--reports=<y or n>	With 'n', does not display a statistical report on how many errors of which type.
errors only	-E	Only displays error messages by checkers that have them.
disable checker(s)	--disable=<id1>,<id2>...	Disable one or more reports (all are enabled by default). Use --disable='all' to disable all
enable checker(s)	--enable=,...	Enable one or more reports (i.e., if--disable='all' has also been used).
display message help	--help-msg=<msg-id>	Display help on a particular message type.
include ids	--include-ids=<y or n>	Include message ids in output

Pylint Checkers
Here is a list of checkers by their command-line option IDs (for disabling or enabling):

elif	object can be used directly in boolean test; too many nested blocks (default 5)
multiple_types	variable changes type within a function or method
imports	bad import style, repeated import, import deprecated module, module importing itself, etc.
variables	unused variables, undefined variables, illegal or improper renaming, etc.
design	improper design such as too many args to a function or method, too many statements in a function, too many locals, etc.
stdlib	very limited misuse of standard library modules
string_constant	backslash found in string seems to indicate r'' was needed
basic	grab bag of errors, omissions and illogicals
newstyle	related to proper use of new-style classes
iterable_check	non-iterable used in an iterable context; non-mapping value used in a mapping context
string	string format errors, string strip errors
format	bad indenting, unnecessary semicolon, trailing whitespace, mixed line endings (LF/CRLF), superfluous parens, bad line continuation spec
miscellaneous	bad format encoding, found FIXME or XXX warning message in code
metrics	"Raw metrics" (no other info found)
spelling	misspelling or invalid characters in docstring or comment
python3	I believe this may be a future-proof check
logging	logging format string interpolation errors
typecheck	mostly type checking that the interpreter usually also does
classes	errors or omissions related to class and metaclass spec and design
similarities	duplicate code
exceptions	misuse or style issues in exception raising or handling

async	async context manager omission of __aenter__/__aexit__, or yield inside async function

team efficiency

Your team will build a code base that must be carefully maintained and husbanded. Some essential components to maintainable code:

- **documentation: proper documentation allows for efficient maintenance and extension by any member of the team, and makes useful library code more accessible to them.**

- **knowledge base: when a new technique, issue, gotcha, etc. is discovered, it should be entered in a document that is part of a corporate wiki or other knowledge base application, so others can benefit from the discovery.**

- **code review: take the time to review each others' code. Make it mandatory. The discipline you make part of your team culture will pay off handsomely.**

- **python lib: a function that is deemed useful outside of the particular script for which it was created, should be placed in a Python library file (a module or standard Python script), properly documented.**