Objects, Object Types and Functions

Goals for this Section: "Objects, Types, Functions"

- **Identify** *parts of speech* in code.
- **Determine** the object *type* and *value* of any *variable* / *object*.
- **Describe** an *algorithm* (code block) in English, using specific terms.
- **Compose code** to accomplish basic data calculations.
- **Identify** *exception types* and **debug** simple code issues.

Programs process data.

All programs hold data in *memory registers* and then use the values held there to perform *computations*.

The simplest example is a calculator -- it contains two memory registers -- the **M** memory register and the current calculated value.

Another more visual example is an Excel spreadsheet -- it contains many values stored in cells which can be summed, averaged, etc.

Python stores all of its computational values in abstractions called *objects*.

Object: a data value of a particular *type*

The *object* is Python's abstraction for handling data. We do all computations with objects.

```
var = 100        # assign integer object 100 to variable var

var2 = 100.0     # assign float object 100.0 to variable var2

var3 = 'hello!'  # assign str object 'hello' to variable var3

# NOTE:  'hash mark' comments are ignored by Python.
```

At every point you must be aware of the type and value of every object in your code.

The three object types we'll look at in this unit are **int**, **float** and **str**. They are the "atoms" of Python's data model.

NOTE: variables in these examples will be called **var**, **var2**, **vara**, etc., or **aa**, **bb**, **xx**, etc. These names are arbitrary and determined by the person who created the code. All other names are built-in to Python.

Variable Assignment and Operators

All programs store data as *objects* in *variables* and compute data with *operations*.

```
xx = 10        # assign/initialize integer 10 to variable xx

yy = xx * 2    # compute 10 * 2 and assign integer 20 to variable yy

print yy       # print 20 to screen
```

xx is a *variable* bound to an *integer object* with value **10**
= is an *assignment operator* assigning **10** to **xx**
10 is an *integer object*, value **10**
yy is another *variable* bound to an *integer object* with *value* **20**
***** is a *multiplication operator* computing its *operands* (*integer objects* **10** and **2**)
print is a *statement* that renders its arguments to the screen.

Math Operators with Numbers: +, -, *, /

Math operators behave as you might expect, with one exception: integer division.

```
var = 5
var2 = 10.3

var3 = var + var2    # int plus a float:  15.3, a float

var4 = var3 - 0.3    # float minus a float:  15.0, a float

var5 = var4 / 3      # float divided by an int:  5.0, a float
```

Here is an example of integer division: the result is always an int, which can be confusing:

```
var = 7
var2 = 3

var3 = var / var2    # 2, an int
```

We resolve this by 'converting' one of the operands to a float -- see 'Conversion Functions' coming up.

Reading code: deducing type based on Python's object type rules

Object types are important to Python -- they determine what is possible (and not possible) in the language. Therefore their "behavior" must be thoroughly understood and in some cases memorized to write effective code.

int with int returns an int

```
tt = 5          # assign an integer value to tt
zz = 10         # assign an integer value to zz

qq = tt + zz    # compute 5 plus 10 and assign integer 15 to qq
```

qq is an integer (the sum of two **int** objects)

float with float returns a float

```
aa = 10.0       # assign a float value to aa
bb = 5.0        # assign a float value to bb

cc = aa + bb    # compute 10.0 plus 5.0 and assign float 15.0 to cc
```

cc is a float (the sum of two **float** objects)

int with float returns a float

```
mm = 10         # assign an int value to mm
nn = 10.0       # assign a float value to nn

oo = mm + nn    # compute 10 plus 10.0 and assign float 20.0 to oo
```

oo is a float (the sum of an **int** object and a **float** object)

str plus str return a new, concatenated str

```
kk = 'hello, '  # assign a str value to kk
rr = 'world!'   # assign a str value to rr

mm = kk + rr    # concatenate 'hello, ' and 'world!' to a new str object, assign to mm

print mm        # 'hello, world!'
```

mm is a **str** (two **strs** concatenated)

Remember, the object types determine the behavior with operators:
- an **int** and an **int** in a math expression produces a new **int**
- a **float** and a **float** in a math expression produces a new **float**
- an **int** and a **float** in a math expression produces a new **float**
- a **str** added to a **str** returns a new **str**

Math Operators with Numbers: exponientation (**)

The exponientation operator (**) raises its left operand to the power of its right operand and returns the result as a float or int.

```
var = 11 ** 2    # "eleven raised to the 2nd power (squared)"
print var        # 121

var = 3 ** 4
print var        # 81
```

Math Operators with Numbers: modulus (%)

The modulus operator (%) shows the remainder that would result from division of two numbers.

```
var = 11 % 2     # "eleven modulo two"
print var        # 1   (11/2 has a remainder of 1)


var2 = 10 % 2    # "ten modulo two"
print var2       # 0   (10/2 divides evenly:  remainder of 0)
```

Because a modulo value of 0 shows that one number divides evenly into the other, we can use it for things like **checking to see if a number is even** or **searching for prime numbers**.

+ Operator with Strings: concatenation

The plus operator (+) with two strings returns a concatenated string.

```
aa = 'Hello, '
bb = 'World!'

cc = aa + bb     # 'Hello, World!'
```

Note that this is the same operator (+) that is used with numbers for summing. Python uses the type to determine behavior.

* Operator with One String and One Integer: string repetition

The "string repetition operator" (*) creates a new string with the operand string repeated the number of times indicated by the other operand:

```
aa = '*'
bb = 5

cc = aa * bb     # '*****'
```

Note that this is the same operator (*) that is used with numbers for multiplication. Python uses the type to determine behavior.

Functions

Built-in functions take *argument(s)* and return *return value(s)*. The arguments are objects *passed* to the function and *returned* from the function (return values).

```
aa = 'hello'        # assign string 'hello' to variable xx

bb = len(aa)        # pass string object aa as an argument to function len(),
                    # which returns an integer object as a return value.

print bb            # 5  (bb is an integer object)
```

* All functions are *called*: the parentheses after the function name indicate the call.
* All functions take *argument(s)* and return *return value(s)*.
  - The *argument* (or comma-separated list of arguments) is placed in parentheses.
  - The *return value* of the *function call* can be *assigned* to a new *variable*. (It can also be printed or used in an expression.)

Function: len()

The **len()** function takes a string argument and returns an integer -- the length of (number of characters in) the string.

```
varx = 'hello, world!'

vary = len(varx)    # 13
```

Function: round()

The **round()** function takes a float argument and returns another float, rounded to the specified decimal place.

With one argument (a float), **round()** rounds to the nearest whole number value.

```
aa = 5.9

bb = round(aa)      # 6.0
```

With two arguments (a float and an int), **round()** rounds to the nearest decimal place.

```
aa = 5.9583

bb = round(aa, 2)   # 5.96
```

Function: raw_input()

The **raw_input()** function takes a string argument and then *pauses execution*, allowing you (the person running the program) to type characters. It returns a string containing the typed characters.

```
cc = raw_input('enter name:  ')    # program pauses!  Now the user types...

print cc                           # [a string, whatever the user typed]
```

The string prompt is optional, but it's recommended since without it, it can be hard to know why the program paused execution -- you might think it crashed or is hanging.

Built-in Function: exit()

The **exit()** function terminates execution immediately. An optional error message can be passed as a string.

```
aa = raw_input()
if aa == 'q':
    exit(0)                         # indicates a natural termination

if aa == '':                        # user typed nothing and hit [Return]
    exit('error:  input required')  # indicates an error led to termination
```

Note: the above examples make use of **if**, which we will cover upcoming.

We can also use **exit()** to simply stop program execution in order to debug:

```
aa = '55'
bb = float(aa)
print 'type of bb is', type(bb)
exit()                    # we inserted this to stop the code from continuing;
                                # we'll remove it later

cc = bb * 2                     # program continues, but this code will not be reached
```

Function: type()

The **type()** function takes any object and returns its type.

```
aa = 5
bb = 5.5
cc = 'hello'

print type(aa)    # <type 'int'>
print type(bb)    # <type 'float'>
print type(cc)    # <type 'str'>
```

We can use this in debugging, since Python cares about type in many of its operations.

Conversion Functions: int(), float() and str()

The **conversion functions** are named after their types -- they take an appropriate value as argument and return an object of that type.

int(): return an int based on a float or int-like string

```
# str -> int
aa = '55'
bb = int(aa)         # 55 (an int)
print type(bb)       # <type 'int'>


# float -> int
var = 5.95
var2 = int(var)      # 5:  the rest is lopped off (not rounded)
```

float(): return a float based on an int or numeric string

```
# int -> float
xx = 5
yy = float(xx)      # 5.0


# str -> float
var = '5.95'
var2 = float(var)   # 5.95 (a float)
```

str(): return a str based on any value

```
var = 5
var2 = 5.5

svar = str(var)     # '5'
svar2 = str(var2)   # '5.5'

print len(svar)     # 1
print len(svar2)    # 3
```

Conversion Challenge #1: treating a string like a number

Numeric data sometimes arrives as strings (e.g. from **raw_input()** or a file). Use **int()** or **float()** to convert to numeric types.

Review: two numeric types added together produce a new number.

```
aa = 5
bb = 5.05

cc = aa + bb       # 10.05, a float
```

Review: two string types 'added' together (concatenated) produce a new string.

```
xx = '5.5'
yy = '5.05'

zz = xx + yy
print zz           # '5.55.05', a str
```

Remember that **raw_input()** produces a string. If the intended input is numeric, we must convert:

```
aa = raw_input('enter number and I will double it:  ')

print type(aa)          # <type 'str'>

num_aa = int(aa)        # int() takes the user's input as an argument
                        # and returns an integer

print num_aa * 2        # prints the user's number doubled
```

You can use **int()** and **float()** to convert strings to numbers.

Conversion Challenge #2: treating an int like a float

Division of two integers results in an integer -- any remainder is discarded. Use **float()** to convert one operand to a float.

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.

Exceptions

An **Exception** is *raised* when you ask Python to do something that it can't understand, or that breaks one of its rules.

```
var1 = 'Hello, '
var2 = 'World!
var3 = var1 + var2
print var3
```

the above code raises the following error:

```
  File "./test.py", line 4
    var2 = 'World!
                 ^
SyntaxError: EOL while scanning string literal
```

This exception indicates that a close quote is missing.

Some students develop the habit of simply "trying something else" when they get an error. It's extremely important not to skip reading your exception message, as it is trying to tell you what's wrong with the code.

More importantly, understanding each exception message will help you understand how to "think like Python" and avoid the problem in future.

Exceptions: SyntaxError

A **SyntaxError** is rasied when code syntax is incorrect.

```
var1 = 'Joe
var2 = 'Below'
var3 = var1 + var2
print var3
```

above code raises this error:

```
  File "./test.py", line 3
    var1 = 'Joe
               ^
SyntaxError: EOL while scanning string literal
```

"EOL" means End of Line; "string literal" refers to the intended string 'Joe'. Python is telling us that it got to the end of the line before encountering the close quote that was started with **'Joe**

When you get a SyntaxError, look closely at the line to make sure there are no missing quotes, commas, parentheses, operators, etc.

Exceptions: NameError

A **NameError** is raised when Python sees a variable name that hasn't been created.

```
var1 = 'Joe'
var2 = Below
var3 = var1 + var2
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 4, in
    var2 = Below
NameError: name 'Below' is not defined
```

In line 2 we attempted to create a string **Below** but we forgot to put quotes around it. Python assumed we meant a *variable* named **Below** and complained that we hadn't defined one.

Exceptions: TypeError

A **TypeError** is raised when an object of the wrong type is used in a statement.

```
var1 = 5
var2 = '5'
var3 = var1 + var2
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 5, in
    var3 = var1 + var2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python is telling us that you can't add a string to an integer. If your purpose was to sum these numbers, you'd want to convert the **str** to a **int** using the **int()** function.

```
var1 = 'Joe'
var2 = 55
var3 = len(var1)     # assign int 3 (len() of 'Joe') to var3

var4 = len(var2)
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 7, in
    var4 = len(var2)
TypeError: object of type 'int' has no len()
```

Python is telling us that you can't get the length of an **int**. If your purpose was to find out how many digits were in **55**, you'd want to convert the **int** to a **str** using the **str** function.

Reading code: identifying the 'parts of speech'

The prerequisite for writing code is the ability to read code. Identifying the "parts of speech" is the first step.

```
var1 = 'Joe'         # var1:  variable
                     # 'Joe':  object

var2 = len(var1)     # var2:  variable, also return value
                     # len():  function
                     # var1:  variable, also argument

var3 = var2 * 2      # var3:  variable
                     # var2:  variable
                     # 2:  object

print var3           # print:  statement
                     # var3:  variable
```

Reflect on the above code and note the definitions and terminology for each element. It is required that you be able identify each element of code.

variable a name bound to an object
object a data entity of a particular type
function a named routine called with an argument; returns a return value
operator an entity that computes or processes its object operands
statementsimilar to a function, it processes objects

Scroll down to test yourself on the above.

In the below code:

```
var1 = 'Joe'
var2 = len(var1)
var3 = var2 * 2
print var3
```

Identify the following:

- var1 (2 IDs)
- 'Joe'
- var2 (2 IDs)
- *
- 2
- len()
- var3
- print

Reading code: tracing program execution

Next, reading code means knowing the *type* and *value* of *every object* in our code.

```
var1 = 'Joe'           # str 'Joe' assigned to var1
var2 = 2               # int 2 assigned to var2
var3 = 'Below'         # str 'Below' assigned to var3
var4 = len(var1)       # int 3 (len() of 'Joe') assigned to var4
var5 = var1 + var3     # str 'JoeBelow' (two strs concatenated) assigned to var5
var6 = len(var5)       # int 8 (len() of 'JoeBelow') assigned to var6
var7 = var6 * var2     # int 16 (8 * 2) assigned to var7

print var7             # 16
```

Always insist on knowing the identity (*type* and *value*) of every object in your code.

Writing code: comments

Use hash marks to comment individual lines; use "triple quotes" to comment multiple lines.

```
# here is a comment that will be ignored
var1 = 'Joe'
var2 = 2
var3 = 'Below'
# var4 = len(var1)   (this line was commented because we didn't need it)
var5 = var1 + var3

"""                   # these 2 lines are commented (ignored)
var6 = len(var5)
var7 = var6 * var2
"""
print var5
```

Writing code: debugging Technique

Use print statements to determine the *type* and *value* of every object in your code.

Suppose you're working on the below code. You were expecting 16 (as it was in the prior example) but instead you're getting 24 (as shown below). How do you elucidate?

```
var1 = 'Joe'
var2 = 2
var3 = 'Below'
var4 = len(var1)
var5 = var1 + var3
var6 = len(var5)
var7 = var6 * var4

print var7             # 24
```

You can read the code and keep track of the variable types and values yourself.
This can work, but keep in mind that if you're incorrect about a values resulting from any step, you will be incorrect about values that depend on that value -- as you have been able to see, these values are handed off and passed along from variable to variable.

You can insert print statements.
This is probably better in most situations. The upside is that at any point you can see what your code is doing -- what is the value and/or type of any object in your code. (The only downside is that if your printed output becomes confusing it may confuse what is happening in your code.)

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
0.30000000000000004

>>> 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**.

Glossary

- argument: a value that is passed to a function for processing
- assign: bind an object to a name
- exception: Python's error condition, raised when there is a problem
- function: a "routine" that performs

- initialize: establish (or re-establish) a new variable through assignment
- object: a value of a particular type (int, float, str, etc.)
- operand: the values on either side of an *operator*
- operator: an entity that processes its *operands*
- raise: when Python signals an error
- return value: a value that is returned from a function after processing is done
- statement: a line or portion of code that accomplishes something
- type: a classification of objects. Every object has at ype
- variable: an object assigned ("bound") to a name

Conditionals: if/elif/else and while

Goals for this Section: "Conditionals"

- **Employ** *conditional (if/else) logic* in our code
- **Make use of** *boolean values* and **True**/**False** tests
- **Identify** code block syntax
- **Use** *while loops* to produce repetitive logic

'if' statement

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

```
aa = raw_input('please enter a positive integer: ')
int_aa = int(aa)

if int_aa < 0:                    # test:  is this a True statement?

    print 'error:  input invalid'  # block (2 lines) -- lines are
    exit(1)                       # executed only if test is True

d_int_aa = int_aa * 2             # double the value
print 'your value doubled is ' + str(d_int_aa)
```

The two components of an if statement are the *test* and the *block*. The test determines whether the block will be executed.

'else' statement

An **else** statement will execute its block if the **if** test before it was *not* **True**.

```
xx = raw_input('enter an even or odd number:  ')
yy = int(xx)

if yy % 2 == 0:                # can 2 divide into yy evenly?
    print xx + ' is even'
    print 'congratulations.'

else:
    print xx + ' is odd'
    print 'you are odd too.'
```

Therefore we can say that only one block of an if/else statement will execute.

'elif' statement

elif is also used with **if** (and optionally **else**): you can chain additional conditions for other behavior:

```
zz = raw_input('type an integer and I will tell you its sign:  ')
zyz = int(zz)

if zyz > 0:
    print 'that number is positive'
    print 'we should all be positive'

elif zyz < 0:
    print 'that number is negative'
    print "please don't be negative"

else:
    print '0 is neutral'
    print "there's no neutral come November."
```

if can be used alone with **elif** alone, with **else** alone, together with both, or by itself. **elif** and **else** obviously require a prior **if** statement.

The Python code block

A *code block* is marked by indented lines. The end of the block is marked by a line that returns to the prior indent.

```
xx = raw_input('enter an even or odd number:  ')  # not in any block
yy = int(xx)                                      # ditto


if yy % 2 == 0:                   # the start of the 'if' block
    print 'your number is even'
    print 'even is cool'          # last line of the 'if' block


else:                             # the start of the 'else' block
    print 'your number is odd'
    print 'you are cool'          # last line of the 'else' block


print 'thanks for playing "even/odd number"'      # not in any block
```

Note also that a block is *preceded by an unindented line that ends in a colon.*

The blocks that we'll look at in this unit are:

```
  - if       conditional
  - elif     conditional
  - else     conditional
  - while    conditional loop
```

Blocks we'll see in upcoming lessons:

```
  - for      sequence loop
  - def      function definition
  - class    class definition
  - try      exception handling block
  - except   exception handling block
```

Nested blocks increase indent

Blocks can be nested within one another. A nested block (a "block within a block") simply moves the code block further to the right.

```
var_a = int(raw_input('enter a number: '))
var_b = int(raw_input('enter another number:  '))

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)'
```

Complex decision trees using 'if' and 'else' is the basis for most programs.

'and' and 'or' for "compound" tests

Python uses the operators **and** and **or** to allow *compound tests*, meaning tests that depend on multiple conditions.

'and' compound statement: both tests must be True

```
xx = raw_input('what is your ID?  ')
yy = raw_input('what is your pin?  ')

if xx == 'dbb212' and yy == '3859':
    print 'you are a validated user'
else:
    print 'you are not validated'
```

'or' compound statement: one test must be True

```
aa = raw_input('please enter "q" or "quit" to quit: ')
if aa == 'q' or aa == 'quit':
    exit()
print 'continuing...'
```

Note the lack of parentheses around the tests -- if the syntax is unambiguous, Python will understand. We can use parentheses to clarify compound statements like these, but they often aren't necessary.

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)."
```

Of course this particular test can also be expressed by replacing the comparison operator **>** with **<=**, but when we learn about new True/False condition types we'll see how this operator can come in handy.

The meaning of **True** and **False**

True and **False** are *boolean* values, and are produced by expressions that can be seen as True or False.

```
aa = 3
bb = 5

if aa < bb:
    print "that is true"


var = 10
var2 = 10.0

if var == var2:
    print "those two are equal"
```

The *if test* is actually different from the *value being tested*. **aa < bb** and **var == var2** look like tests but are actually expressions that resolve to **True** or **False**, which are values of *boolean type*:

```
xx = (5 < 3)
print xx            # True
print type(xx)      # <type 'bool'>

yy = (var == var2)
print yy            # False
print type(yy)      # <type 'bool'>
```

Note that we would almost never assign comparisons like these to variables, but we are doing so here to illustrate that they resolve to boolean values.

while loops

A **while** test causes Python to loop through a block repetitively, as long as the test is True.

This program prints each number between 0 and 4

```
cc = 0            # initialize a counter

while cc < 5:     # "if test is True, enter the block"
    print cc
    cc = cc + 1   # "increment" cc:  add 1 to its current value
                  # WHEN WE REACH THE END OF THE BLOCK,
                  # JUMP BACK TO THE while TEST

print 'done'
```

This means that the block is executing the **print** and **cc = cc + 1** lines multiple times - again and again until the test becomes False.

Here's the execution order of lines, spelled out as if it were successive statements.

```
cc = 0
while cc < 5:    # testing 0 < 5:  True
    print cc         # 0
    cc = cc + 1      # 0 becomes 1
                  # block ends, Python returns to top of loop

while cc < 5:    # testing 1 < 5:  True
    print cc         # 1
    cc = cc + 1      # 1 becomes 2
                  # block ends, Python returns to top of loop

while cc < 5:    # testing 2 < 5:  True
    print cc         # 2
    cc = cc + 1      # 2 becomes 3
                  # block ends, Python returns to top of loop

while cc < 5:    # testing 3 < 5:  True
    print cc         # 3
    cc = cc + 1      # 3 becomes 4
                  # block ends, Python returns to top of loop

while cc < 5:    # testing 4 < 5:  True
    print cc         # 4
    cc = cc + 1      # 4 becomes 5
                  # block ends, Python returns to top of loop

while cc < 5:    # testing 5 < 5:  False.
                 # Python drops to below the loop.

 print 'done'
```

Of course, the value being tested must change as the loop progresses - otherwise the loop will cycle indefinitely (endless loop).

Understanding while loops

while loops have 3 components: the *test*, the *block*, and the *automatic return*.

```
cc = 10

while cc > 0:     # THE TEST (if True, enter the block)

    print cc      # THE BLOCK (execute as regular Python statements)
    cc = cc - 1

    [invisible!]  # THE AUTOMATIC RETURN
                  # (at end of block, go back to the test)

print 'done'
```

Software's *repetitive processing* technique requires that we be able to say "do this thing over and over until a condition is met, or until I tell you to stop".

Can you tell just from reading what this code prints? You'll need to keep track of the value of **cc** and calculate its changes as the code block is executed repetitively.

In order to use **while** loops you must be able to model code execution in your head. This takes some practice but isn't complicated. (See the fully modeled explanation in the previous slide for an example of modeling.)

Loop control: "break"

break is used to exit a loop regardless of the test condition.

```
xx = 0
while xx < 10:
    answer = raw_input("do you want loop to break? ")
    if answer == 'y':
        break             # drop down below the block
    print 'Hello, User'
    xx = xx + 1
    print 'I have now greeted you ', xx, ' times'

print "ok, I'm done"
```

Loop control: "continue"

The **continue** statement jumps program flow to next loop iteration.

```
x = 0
while x < 10:
    x = x + 1
    if x % 2 != 0:   # will be True if x is odd
        continue     # jump back up to the test and test again
    print x
```

Note that **print x** will not be executed if the **continue** statement comes first. Can you figure out what this program prints?

The "while True" loop

while with **True** and **break** provide us with a handy way to keep looping until we feel like stopping.

```
while True:
    var = raw_input('please enter a positive integer:  ')
    if int(var) > 0:
        break
    else:
        print 'sorry, try again'

print 'thanks for the integer!'
```

Note the use of **True** in a **while** expression: since **True** is always **True** this test will be false. Therefore the **break** statement is essential to keep this loop from looping indefinitely.

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 from a range (i.e., 1 + 2 + 3 + 4, etc.). *It has a major bug*. Can you spot it?

```
revcounter = 0
while revcounter < 10:

    varsum = 0                      # set varsum to 0
    revcounter = revcounter + 1     # increment value of revcounter by 1
    varsum = varsum + revcounter    # add value of revcounter to varsum

print varsum                        # prints 11 (should be 55)
```

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 increasing **revcounter** by one until it becomes **10**. At the end, why is the value of **varsum** only 10? 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, and it's easy to make a mistake in your head.

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 = 0
while revcounter < 10:

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

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

print varsum                        # 10
```

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:  1
varsum value:  0
pausing...                          # here I hit [Return] to continue


loop iteration complete
revcounter value:  2
varsum value:  1
pausing...                          # [Return]


loop iteration complete
revcounter value:  3
varsum value:  2
pausing...


loop iteration complete
revcounter value:  4
varsum value:  3
pausing...
```

Just looking at the first iteration, we can see that the values are as we might expect: it seems that **revcounter** was 0, it added its value to **varsum**, and then had its value incremented by 1. So **revcounter** is now **1** and **varsum** is now 0.

So far so good. The code has paused, we hit **[Return]** to continue, and the loop iterates again. We can see that **revcounter** has increased by 1, as we expected: it is now **2**. 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 = 0
varsum = 0
while revcounter < 10:

    revcounter = revcounter + 1
    varsum = varsum + revcounter

print varsum
```

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!

Object Methods

Goals for this Section: "Object Methods"

- **Use** object methods to *process* object values

```
var = 'Hello, World!'
var2 = var.replace('World', 'Mars')  # replace substring, return a str
print var2                           # Hello, Mars!
```

- **Use** object methods to *inspect* object values

```
var = 'Hello, World!'
var2 = var.count('l')     # count the 'l' characters, return an int
print var2                # 3
```

Objects are capable of *behaviors*.
Objects are much more than values. They come equipped with *behaviors*. We call these behaviors *methods*.

Some methods are used to manipulate or process the object's value. Others are used to *inspect* the object and tell us things about its value.

Methods

Methods are *custom functions* that are used *only with a particular type*.

upper() is a **str** method. It is only used with strings.

```
var = 'hello!'       # str object assigned to var

var2 = var.upper()   # call the upper method on str object var

print var2           # HELLO!
```

Note the syntax: *object*.*method*(). We *call* the method in the same way we call functions: with parentheses.

Methods vs. Functions

Compare *method* syntax to *function* syntax.

```
mystr = 'HELLO'

x = len(mystr)        # call len() and pass mystr, returning int 5

y = mystr.count('L')  # call count() on mystr, pass str L, returning int 2

print y                           # 2
```

Methods and functions are both *called* (the parentheses after the name of the function or method).

Both also may take an *argument* and/or may return a *return value*.

String Methods: upper() and lower()

These methods return a new string with a string's value uppercased or lowercased.

upper() string method

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

print newvar           # 'HELLO'
```

lower() string method

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

print newvar           # 'hello there'
```

String Methods: replace()

The **replace()** string method takes two arguments - a substring to be replaced, and the string with which to replace it. The string with replacements is returned as a new string object.

```
var = 'My name is Joe'

newvar = var.replace('Joe', 'Greta')   # pass 2 arguments to this method:
                                       # substring and replacement string

print newvar                           # My name is Greta
```

String Methods: format()

The string **format()** method performs string substitution, placing *any* value (even numbers) within a new, completed string.

```
aa = 'Jose'
var = 34
bb = '{} is {} -- {} years old'.format(aa, var, var)  # 3 arguments to replace 3 {} tokens
print bb                                    # Jose is 34 years old -- 34


cc = '{name} is {age} years old -- {age}'.format(name=aa, age=var)  # 2 arguments to replace 3 tokens (2 differ
```

format() is a string method, but it is often called on a *literal string* (that is, a string that is written out literally in your code).

The string must contain *tokens* (marked by curly braces) that will be replaced by values. The values are passed as arguments to **format()**.

format() is the preferred way for combining strings with other values. Concatenation or commas are usally too "busy" for such purposes:

```
print aa + ' is ' + str(var) + ' years old'
```

String Methods: isdigit() and isalpha()

These *inspector* methods return **True** if a string is all digits or all alphabetic characters.

Since they return **True** or **False**, they are used in an **if** or **while** expression.

isdigit(): return **True** if this string contains all digit characters

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

isalpha(): return **True** if this string is composed of all alphabetic characters

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

String Methods: endswith() and startswith()

These inspector methods return **True** if a string starts with or ends with a substring.

endswith(): return **True** if the string ends with a substring

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

startswith(): return **True** if the string starts with a substring

```
cc = raw_input('continue? ')
if cc.startswith('y') or cc.startswith('Y'):
    print 'thanks!'
else:
    print "ok, we'll wait."
```

String Methods: count() and find()

These inspector methods return integer values.

count(): return the number of times a substring appears in this string

```
aa = 'count the substring within this string'
bb = aa.count('in')
print bb             # 3 (the number of times '3' appears in the string)
```

find(): return the *index position* (starting at 0) of a substring within this string

```
xx = 'find the name in this string'
yy = xx.find('name')
print yy                            # 9 -- the 10th character in mystring
```

Method and Function Return Values in an Expression; Combining Expressions

The return value of an expression can be used in another expression.

```
letters = "aabbcdefgafbdchabacc"

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

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

varc = vara / varb           # 20 / 5, or .25, the percentage of a's in the string


print len(letters) / letters.count("a")   # the above lines combined


print float(letters.count("a")) / len(letters) * 100    # 25

# the above line adds one more calcluation:
# the whole-number percentage of a's in this string
```

Sequences: Strings, Lists and Files

Table Data: Rows and Fields

Tables consist of *records* (rows) and *fields* (column values).

Tabular text files are organized into rows and columns.

comma-separated values file (CSV)

```
19260701,0.09,0.22,0.30,0.009
19260702,0.44,0.35,0.08,0.009
19270103,0.97,0.21,0.24,0.010
19270104,0.30,0.15,0.73,0.010
19280103,0.43,0.90,0.20,0.010
19280104,0.14,0.47,0.01,0.010
```

space-separated values file

```
19260701    0.09    0.22    0.30   0.009
19260702    0.44    0.35    0.08   0.009
19270103    0.97    0.21    0.24   0.010
19270104    0.30    0.15    0.73   0.010
19280103    0.43    0.90    0.20   0.010
19280104    0.14    0.47    0.01   0.010
```

Our job for this lesson is to *parse* (separate) these values into usable data.

Table Data in Text Files

Text files are just sequences of characters. *Newline* characters separate text files into lines. Python reads text files *line-by-line* by separating lines at the newlines.

If we print a CSV text file, we may see this:

```
19260701,0.09,0.22,0.30,0.009
19260702,0.44,0.35,0.08,0.009
19270103,0.97,0.21,0.24,0.010
19270104,0.30,0.15,0.73,0.010
19280103,0.43,0.90,0.20,0.010
19280104,0.14,0.47,0.01,0.010
```

However, here's what a text file really looks like under the hood:

```
19260701,0.09,0.22,0.30,0.009\n19260702,0.44,0.35,0.08,
0.009\n19270103,0.97,0.21,0.24,0.010\n19270104,0.30,0.15,
0.73,0.010\n19280103,0.43,0.90,0.20,0.010\n19280104,0.14,
0.47,0.01,0.010
```

The newline character separates the **records** in a CSV file. The *delimeter* (in this case, a comma) separates the fields.

When displaying a file, your computer will translate the newlines into a line break, and drop down to the next line. This makes it seem as if each line is separate, but in fact they are only separated by newline characters.

Goals for this Unit

These objectives encapsulate the core, more central approach to processing data in Python.

The Summing Average

Can we sum a column of values from a table? Our data analysis will resemble that of **excel**: it will read a column of value from a tabular file and *sum up* the values in that column.

Here's how it works:

- We set a summing float variable to 0.
- We call function **open()** with a filename, returning a **file object**.
- We loop through the file by using **for** with the file object.
- Inside the **for** block, each line is a string. We **split()** the string on the delimeter (here, a comma), returning a **list of strings**.
- We **subscript** the list to select the field value we want -- this is a string object with a numeric value.
- We convert the string to a float object.
- We add the float object to the summing float variable.
- The loop continues until the file is exhausted. **The Counting** *Can we parse and count the lines, words and characters in a file?* We will emulate the work of the Unix **wc** (word count) utility, which does this work. Here's how it works:
- We call function **open()** with a filename, returning a file object.
- We call **read()** on the file object, returning a string object containing the entire file text.
- We call **splitlines()** on the string, returning a list of strings. **len()** will then tell us the number of lines.
- We call **split()** on the same string, returning a list of strings. **len()** will then tell us the number of words.
- We call **len()** on the string to count the number of characters.

Summary: File Object

3 ways to read strings from a file.

for: read line-by-line

```
fh = open('../python_data/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

fh.close()                                # close the file
```

read(): read entire file as a single string

```
fh = open('../python_data/students.txt')  # file object allows reading
text = fh.read()                          # read() method called on file object returns a string
fh.close()                                # close the file

print text
```

The above prints:

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

readlines(): read as a list of strings

```
fh = open('../python_data/students.txt')
file_lines = fh.readlines()               # file.readlines() returns a list of strings
fh.close()                                # close the file
print file_lines
```

The above prints:

```
['jw234,Joe,Wilson,Smithtown,NJ,2015585894\n', 'ms15,Mary,Smith,Wilsontown,
NY,5185853892\n', 'pk669,Pete,Krank,Darkling,NJ,8044894893\n']
```

Summary: String Object

Strings: 4 ways to manipulate strings from a file.

split() a string into a list of strings

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

slice a string

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

strip() a string

```
xx = 'this is a line with a newline at the end\n'

yy = xx.rstrip()              # return a new string without the newline

print yy                      # 'this is a line with a newline at the end'
```

splitlines() a multiline string

```
fh = open('../python_data/students.txt')  # open the file, return a file object
text = fh.read()                          # read the entire file into a string
                                          # (of course this includes newlines)

lines = text.splitlines()                 # returns a list of strings
                                          # (similar to fh.readlines(),
                                          # except without newlines)
```

Summary: List Object

Lists: selecting individual elements of a list.

A list is a sequence of objects of any type:

initialize a list: lists are initalized with square brackets and comma-separated objects.

```
aa = ['a', 'b', 'c', 3.5, 4.09, 2]
```

subscript a list: using the list name, square brackets an an element *index*, starting at 0

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

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

Summary: len() function for string and list length

len() can be used to measure lists as well as strings.

```
mystr = 'hello'
mylist = [1.3, 1.9, 0.9, 0.3]

lms = len(mystr)                # 5  (number of characters in mystr)
lml = len(mylist)               # 4  (number of elements in mylist)
```

Because it can measure lists or strings, **len()** can also measure files (when rendered as a list of strings or a whole string).

Summary: repr() function for "true" representations of strings

repr() takes any object and shows a more "true" representation of it. With a string, **repr()** will show us the newlines at the end of each line

```
aa = open('../python_data/small_db.txt')  # open a file, returns a file object

xx = aa.read()                # read() on a file object, returns a single string

print repr(xx)                # the string with newlines visible:  '101:Acme:483982.90\n102:Boon:119001.94\n103
```

Reading a file: options

(Note that the remaining slides repeat some of the same material, but from a more practical perspective.)

for loop: loop line-by-line

The **for** loop repeats execution of its block until the file is completely read.

Note that the **for** block is very similar to the **while**. The difference is that **while** relies on a test to continue executing, but **for** continues until it reaches the end of the file.

```
fh = open('../python_data/students.txt')    # file object allows looping through a
                                            # series of strings

for xx in fh:                               # xx is a string

    print xx                                # prints each line of students.txt

fh.close()                                  # close the file
```

"my_file_line" is called a *control variable*, and it is *automatically reassigned* each line in the file as a string.

break and **continue** work with **for** as well as **while** loops.

readlines(): work with the file as a list of string lines
To capture the entire file into a list of lines, use the file **readlines()** method:

```
fh = open('../python_data/students.txt')

lines = fh.readlines()

for line in lines:
    line = line.rstrip()
    print line

print lines[0]             # the first line from the file

print len(lines)           # the number of lines in the file
```

We can then loop through the list, or perform other operations (select a single line or slice, get the number of lines with **len()** of the list, etc.)

read() with **splitlines()**: an easy way to drop the newlines
A handy trick is to **read()** the file into a string, then call **splitlines()** on the string to split on newlines.

```
fh = open('../python_data/students.txt')

text = fh.read()
lines = text.splitlines()

for line in lines:
    print line
```

This has the effect of delivering the entire file as a list of lines, but with the newlines removed (because the string was split on them with **splitlines()**).

Table *Records* Are Read from a file as String Objects

As Python reads files line-by-line, it handles each line as a string object.

```
fh = open('../python_data/students.txt')   # file object allows looping through a
                                           # series of strings
for bb in fh:
     print type(bb)                        # <type 'str'>
```

Again, the control variable **bb** is *reassigned for each iteration of the loop*. This means that if the file has 5 lines, the loop executes 5 times and **bb** is reassigned a new value 5 times.

Stripping a file line with **rstrip()**

When reading a file line-by-line, we should strip off the newline with the string method **rstrip()**.

```
fh = open('../python_data/students.txt')    # file object allows looping through a
                                            # series of strings
for xx in fh:                    # xx is a string

    xx = xx.rstrip()             # remove "whitespace" from end of the line
                                 # and return a new string with the string removed

    print xx                     # prints each line of students.txt

fh.close()                       # close the file
```

String slicing

A string can be *sliced* by position: we specify the start and end position of the slice.

Indices start at 0; the "upper bound" is *non-inclusive*

```
mystr = '19320805    3.62   -2.38    0.08   0.001'
year =  mystr[0:4]             # '1932'
month = mystr[4:6]             # '08'
day = mystr[6:8]               # '05'
```

To slice to the end, omit the upper bound

```
mystr = '19320805    3.62   -2.38    0.08   0.001'

rf_val = mystr[32:]            # '   0.001'
```

Table *Fields* Are Parsed from File Line Strings into *Lists of Strings*

The string **split()** method returns a *list of strings*, each string a field in a single record (row or line from the table).

The *delimeter* tells Python how to split the string. Note that the delimeter does *not* appear in the list of strings.

```
line_from_file = 'jw234:Joe:Wilson:Smithtown:NJ:2015585894\n'

xx = line_from_file.split(':')

print xx                         # ['jw234', 'Joe', 'Wilson', 'Smithtown', 'NJ', '2015585894\n']
```

If no delimeter is supplied, the string is split *on whitespace*:

```
gg = 'this is a file    with    some     whitespace'

hh = gg.split()                  # splits on any "whitespace character"

print hh                         # ['this', 'is', 'a', 'file', 'with', 'some', 'whitespace']
```

Table Fields Are Selected from a List Using *List Subscripts*

Each table record (row or line from the table) when rendered as a list of strings (from **split()**) is *addressable* by *index*.

The index starts at 0. A *negative index* (-1, -2, etc.) will count from the end.

```
gg = '2016:5.0:5.3:5.9:6.1'

hh = gg.split(':')            # splits on any "whitespace character"

print hh                      # ['2016', '5.0', '5.3', '5.9', '6.1']

kk = hh[0]                    # '2016'   (index starts at 0)

mm = hh[1]                    # '5.0'

zz = hh[-1]                   # '6.1'    (negative index selects from the end of the list)

yy = hh[-2]                   # '5.9'
```

Portions of a string can be "sliced" using *string slicing*

Special *slice syntax* lets us specify a *substring* by position.

split() separates a string based on a delimeter, but some strings have no delimeter but must be parsed by position:

```
mystr = '20140313'
year =  mystr[0:4]             # '2014'  (the 0th through 3rd index)
month = mystr[4:6]             # '03'    (the 4 and 5 index values)
day =   mystr[6:]              # '13'    (note that no upper index means slice to the end)
```

Note that the upper index is *non-inclusive*, which means that it specifies the index *past* the one desired.

stride and *negative stride*

A third value, the *stride* or *step* value, allows skipping over characters (every 2nd element every 3rd element, etc.)

```
mystr = '20140303'

skipper = mystr[0:7:2]        # '2100'
```

The negative stride actually *reverses* the string (when used with no other index):

```
mystr = '20140303'

reverser = mystr[::-1]      # '0304102'
```

Modules: Leveraging Internet Data

Importing Python Modules for Versatility and Power

A Module is Python code (a *code library*) that we can *import* and use in our own code -- to do specific types of tasks.

```
import datetime                         # make datetime (a library module) part of our code

dt = datetime.date.today()              # generate a new date object (dt)
print dt                                # prints today's date in YYYY-MM-DD format
dt = dt + datetime.timedelta(days=1)
print dt                                # prints tomorrow's date
```

Once a module is imported, its Python code is made available to our code. We can then call specialized functions and use objects to accomplish specialized tasks.

Python's module support is profound and extensive. Modules can do powerful things, like manipulate image or sound files, munge and process huge blocks of data, do statistical modeling and visualization (charts) and much, much, much more.

CSV

The CSV module parses CSV files, splitting the lines for us. We read the CSV object in the same way we would a file object.

```
import csv
fh = open('../python_data/students.txt', 'rb')  # second argument: read in binary mode
reader = csv.reader(fh)

for record in reader:     # loop through each row

    print 'id:{};  fname:{}; lname: {}'.format(record[0], record[1], record[2])
```

This module takes into account more advanced CSV formatting, such as quotation marks (which are used to allow commas within data.)

The second argument to **open()** ('rB') is sometimes necessary when the csv file comes from Excel, which output newlines in the Windows format (**\r\n**), and can confuse the **csv** reader.

Writing is similarly easy:

```
import csv
wfh = open('some.csv', 'w')
writer = csv.writer(wfh)
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 as a web client: the urllib2 module

A Python program can take the place of a browser, requesting and downloading CSV, 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, 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 (keep in mind that many objects that can deliver file-like string output can be read with this same-named method:

```
for line in readobj.readlines():
  print line
readobj.close()
```

The text that is downloaded is CSV, HTML, Javascript, and possibly other kinds of data.

Encoding Parameters: urllib2.urlencode()

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>
```

Containers: Lists, Sets and Tuples

Introduction: Collections of values can be used for various types of analysis.

With a collection of numeric values, we can perform many types of analysis that would not be possible with a simple count or sum.

We will summarize a year's worth of data in the Fama-French file as we did previously, but be able to say much more about it.

```
var = [1, 4.3, 6.9, 11, 15]                       # a list container

print 'count is {}    '.format(len(var))           # count is 5
print 'sum is {}      '.format(sum(var))           # sum is XXXX
print 'average is {}  '.format(sum(var) / len(var)) # average is XXXX

print 'max val is {}'.format(max(var))             # max val is 15
print 'min val is {}'.format(min(var))             # min val is 1

print 'top two: {}, {}'.format(var[3], var[4])     # top two: 11, 15

print 'median is {}   '.format(var[len(var) / 2])  # median is XXXX
```

Introduction: Collections of values can be used to determine membership.

Checking one list against another is a core task in data analysis. We can validate arguments to a program, see if a user id is in a "whitelist" of valid users, see if a product is in inventory, etc.

We will apply membership testing to a *spell checker*, which simply checks every word in a file against a "whitelist" of correctly spelled words.

```
valid_actions = ['run', 'stop', 'search', 'reset']

input = raw_input('please enter an action:  ')

if input in valid_actions:                    # if string can be found in list
    print 'great, I will "{}"'.format(input)
else:
    print 'sorry, action not found'
```

Objectives for this Unit (Containers: Lists, Sets and Tuples)

Containers broaden our data analysis powers significantly over simple looping and summing. We can:

- Use lists to build up sequences of non-unique values.
- Use sets to build up collections of unique values.
- Use summary functions to summarize numeric data in containers (sum, max, min, etc.)
- Use sorting and slicing to do ordered analysis (top 5, median, etc.)
- Use membership analysis (**in**) to check a value against a collection of values.

Container Objects: List, Set, Tuple

Compare and contrast the characteristics of each container.

- **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 upcoming)

Summary for object: "List" container object

A **list** is an *ordered sequence* of values.

Initialize a List

```
var = []                    # initialize an empty list

var2 = [1, 2, 3, 'a', 'b']   # initialize a list of values
```

Append to a List

```
var = []

var.append(4)        # Note well! call is not assigned
var.append(5.5)      #            list is changed in-place

print var            # [4, 5.5]
```

Slice a List (compare to string slicing)

```
var2 = [1, 2, 3, 'a', 'b']     # initialize a list of values

sublist = var2[2:4]            # [3, 'a']
```

Subscript a List

```
mylist = [1, 2, 3, 'a', 'b']   # initialize a list of values

xx = mylist[3]                 # 'a'
```

Get Length of a List (compare to **len()** of a string)

```
mylist = [1, 2, 3, 'a', 'b']

yy = len(mylist)               # 5 (# of elements in mylist)
```

Test for membership in a List

```
mylist = [1, 2, 3, 'a', 'b']

if 'b' in mylist:                         # this is True for mylist
    print "'b' can be found in mylist"    # this will be printed

print 'b' in mylist                       # "True":  the in operator actually
                                          #          returns True or False
```

Loop through a List (compare to looping through a file)

```
mylist = [1, 2, 3, 'a', 'b']

for var in mylist:
    print var                 # prints 1, then 2, then 3, then a, then b
```

Sort a List: **sorted()** returns a list of sorted values

```
mylist = [4, 9, 1.2, -5, 200, 20]

smyl = sorted(mylist)              # [-5, 1.2, 4, 9, 20, 200]
```

Summary for object: "Set" container object

A **set** is an *unordered*, *unique* collection of values.

Initialize a Set

```
myset = set()                      # initialize an empty set

mixed_set = set(['a', 9999, 4.3])  # the arg to set() is a list!
```

Add to a Set

```
myset = set()                       # initialize an empty set

mixed_set.add('a')                  # note well not assigned to a variable
mixed_set.add(4.3)

print mixed_set                     # set([4.3, 'a'])     (order is not
```

Get Length of a Set

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

setlen = len(mixed_set)              # 3
```

Test for membership in a Set

```
myset = set(['b', 'a', 'c'])
if 'c' in mytup:                     # test is True
    print "'c' is in mytup"          # this will be printed
```

Loop through a Set

```
myset = set(['b', 'a', 'c'])
for el in myset:
    print el                         # will be printed in 'random' order
```

Sort a Set: **sorted()** returns a list of sorted object values

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

zz = sorted(myset)                   # ['a', 'b', 'c']
```

Summary for object: "Tuple" container object

A **tuple** is an *immutable ordered* sequence of values. Immutable means it cannot be changed once initialized.

Initialize a Tuple

```
var = ('a', 'b', 'c', 'd')    # initialize an empty tuple
```

Slice a Tuple

```
var = ('a', 'b', 'c', 'd')
varslice = var[1:3]           # ('b', 'c')
```

Subscript a Tuple

```
mytup = ('a', 'b', 'c')
last = mytup[2]              # 'c'
```

Get Length of a Tuple

```
mytup = ('a', 'b', 'c')
tuplen = len(mytup)

print tuplen              # 3
```

Test for membership in a Tuple

```
mytup = ('a', 'b', 'c')
if 'c' in mytup:
    print "'c' is in mytup"
```

Loop through a Tuple

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

Sort a Tuple

```
xxxx = ('see', 'i', 'you', 'ah')

yyyy = sorted(xxxx)             # ('ah', 'i', 'see', 'you')
```

Summary for functions: len(), sum(), max(), min()

Summary functions offer a speedy answer to basic analysis questions: how many? How much? Highest value? Lowest value?

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

print len(mylist)                  # 5
print sum(mytup)                   # 292.3 sum of values in mytup
print min(mylist)                  # 1 smallest value in mylist
print max(myset)                   # 3.8 largest value in myset
```

Summary for function: sorted()

The **sorted()** function takes any sequence as argument and returns a list of the elements sorted by numeric or string value.

```
x = set([1.8, 0.9, 15.2, 3.5, 2])

y = sorted(x)                  # [0.9, 1.8, 2, 3.5, 15.2]
```

Irregardless of the sequence passed to **sorted()**, a list is returned.

Summary task: Adding to Containers

We can add to a list with **append()** and to a set with **add**.

Add to a list

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

intlist1.append('hello')

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

Add to a set

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

mixed_set.add('a')                     # not added - duplicate
mixed_set.add('cool')

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

We cannot add to a tuple, of course, since they are immutable!

Summary task: Looping through Containers

We can loop through any container with **for**, just like a file.

Loop through a List

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

Loop through a Set

```
myset = set(['b', 'a', 'c'])
for el in myset:
    print el                              # will be printed in 'random' order
```

Loop through a Tuple

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

Loop through a String(??)

```
mystr = 'abcdefghi'

for x in mystr:
    print x         # what do you see?
```

Summary task: Subscripting and Slicing Containers

We can slice any *ordered* container -- for us, list or tuple.

```
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 the rules with slices:
   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"
```

We cannot subscript or slice a set, of course, beacuse it is unordered!

Summary task: Checking for Membership

We can check for value membership of a value within any container with **in**.

```
mylist = [1, 2, 3, 'a', 'b']

if 'b' in mylist:                       # this is True for mylist
    print "'b' can be found in mylist"   # this will be printed
```

Summary task: Sorting Containers

Sorting allows us to rank values, find a median, and more.

```
mylist = [9.3, 2.1, 0.8]
xxx = sorted(mylist)                    # a list:  [0.8, 2.1, 9.3]

names = set(['David', 'George', 'Adam'])
yyy = sorted(names)                     # a list:  ['Adam', 'David', 'George']

ints = (5, 9, 0, 8)
zzz = sorted(ints)                      # a list:  [0, 5, 8, 9]
```

No matter what sequence is passed to **sorted()**, a list is returned. What about a string?!

Summary Exception: AttributeError

An **AttributeError** exception usually means calling a method on an object type that doesn't support that method.

Summary Exception: IndexError

An **IndexError** exception indicates use of an index for a list/tuple element that doesn't exist.

Practical: looping through a data source and building up containers

The "summary algorithm" is very similar to building a float sum from a file source. We loop; select; add.

list: build a list of states

```
state_list = []                      # initialize an empty list
for line in open('student_db.txt'):

    elements = line.split(':')
    state_list.append(elements[3])     # add the state for this row to state_list

chosen_state = raw_input('enter a state ID:  ')
state_freq = state_list.count(chosen_state)
print '{} occurs {} times'.format(chosen_state, state_freq)
```

set: build a set of unique states

```
state_set = set()                      # initialize an empty set
for line in open('student_db.txt'):

    elements = line.split(':')
    state_set.append(elements[3])     # add the state for this row to state_list

chosen_state = raw_input('enter a state ID:  ')

if chosen_state in state_set:
   print 'that is a valid state'
else:
    print 'that is not a valid state'
```

Practical: checking for membership

We use **in** to compare two collections.

In this example, we have a **list** of ids and a **set** of valid ids. With looping and **in** we can build a list of valid and invalid ids.

```
student_states = ['CA', 'NJ', 'VT', 'ME', 'RI', 'CO', 'NY']
ne_states = set(['ME', 'VT', 'NH', 'MA', 'RI', 'CT'])

ne_student_states = []
for state in student_states:
    if state in ne_states:
        ne_student_states.append(state)

print 'students in our school are from these New England states: ', ne_student_states
```

This kind of analysis can also be done purely with **sets** and we'll discuss these methods later in the course.

Practical: treating a file as a list

Data files can be rendered as lists of lines, and slicing can manipulate them holistically rather than by using a counter.

In this example, we want to skip the 'header' line of the **student_db.txt** file. Rather than count the lines and skip line 1, we simply treat the entire file as a list and slice the list as desired:

```
fh = open('../python_data/student_db.txt')
file_lines_list = fh.readlines()            # a list of lines in the file
print file_lines_list
      # [ "id:address:city:state:zip",
      #   "jk43:23 Marfield Lane:Plainview:NY:10023",
      #   "ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027",
      #   "jab44:23 Rivington Street, Apt. 3R:New York:NY:10002"  ]

wanted_lines = file_lines_list[1:]          # take all but 1st element (i.e., 1st line)
for line in wanted_lines:
  print line.rstrip()                       # jk43:23 Marfield Lane:Plainview:NY:10023
                                            # ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027
                                            # jab44:23 Rivington Street, Apt. 3R:New York:NY:10002
```

Sidebar: removing a container element

We rarely need to remove elements from a container, but here is how we do it.

```
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

Introduction: Dictionaries for Paired Data

dicts pair unique keys with associated values.

Much of the data we use tends to be *paired*:

- **companies** paired with **annual revenue** for each
- **employees** with **contact information** for each
- **students** with **grade point averages**
- **dates** with the **high temperature** for each
- **web pages** with the **number of times** each was accessed To store paired data, we use a Python container called a *dict*, or dictionary. The dict contains *keys* paired with *values*.

```
customer_balances = { 'jk125': 493.95,      # spacing for clarity
                      'xx3':   122.03,
                      'jp9':   23238.72 }

print customer_balances          # { 'jk125': 493.95, 'xx3': 122.03, 'jp9': 23238.72 }
print type(customer_balances)    # <type 'dict'>
```

dicts have some of the same features as other containers. When standard container operations are applied, the *keys* are used:

```
print len(customer_balances)        # 3 keys in dict

print 'xx3' in customer_balances    # True:  the key is there

for yyy in customer_balances:
    print yyy                       # prints each key in customer_balances

print customer_balances['jp9']      # 23238.72
```

Objectives for the Unit: Dictionaries

This *key/value pairs* container allows us to summarize data in powerful ways:

- Use a dict to store and report a value for each a unique collection of keys, such as date to price, contract to expiration, device to uptime, etc.
- Use a dict to take a "running sum" or "running count", such as summing up revenue by month (based on daily revenue data), counting the number of times a web page was visited, counting the number of error requests each month, etc.

Summary for Object: Dictionary (dict)

A dictionary (or *dict*) is an *unordered collection* of *unique key/value pairs* of objects.

The keys in a dict are *unordered* and *unique*. In this way it is like a **set**.

A dict key can be used to obtain the assocaiated *value* ("addressible by key"). In this way it is like a **list** or **tuple** (which use an *integer index* to obtain a value).

initialize a dict

```
mydict = {}                         # empty dict

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

add a key/value pair to a dict

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

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

read a value based on a key

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

xxx = mydict['c']               # value for 'c' is 3
```

Note in the standard container features below, only the *keys* are used:

loop through a dict and read keys and values

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

for key in mydict:
    print key          # prints 'a', then 'c', then 'b', then 'd'
```

check for key membership

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

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

sort a dict's keys

```
mydict = {'xenophon': 10, 'abercrombie': 5, 'denusia': 3}

mykeys = sorted(mydict)         # ['abercrombie', 'denusia', 'xenophon']
```

retrieve list of keys or list of values

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

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

zzz = mydict.values()                # [1, 3, 2]
```

Summary dict Method: get()

The **dict get()** method returns a value based on a key. An optional 2nd argument provides a *default* value if the key is missing.

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

xx = mydict.get('a', 0)            # 1 (key exists so paired value is returned)

yy = mydict.get('zzz', 0)          # 0 (key does not exist so default value is returned)
```

The default value is your choice.

This method is sometimes used as an alternative to testing for a key in a dict before reading it -- avoiding the **KeyError** exception that occurs when trying to read a nonexistent key.

Summary dict Methods: keys() and values()

keys() returns a list of keys; *values()* returns a list of values.

These methods are used for advanced manipulation of a **dict**.

dict keys() method returns a list of keys

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

yyy = mydict.keys()                  # ['a', 'c', 'b']
```

We might want a list of a dict's keys for advanced manipulation of a dictionary. We can retrieve the list for sorting, for testing for membership, for checking length, BUT these can also accomplished by using the dictionary directly: a dict used in a "listy" context always works with its keys.

dict values() method returns a list of values in the dict

```
zzz = mydict.values()                # [1, 3, 2]
```

This method, like, **keys()** is also less often used -- to test for membership or frequency among values, for example.

Summary Exception: KeyError

The **KeyError** exception indicates that the requested key does not exist in the dictionary.

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

xx = mydict['a']                     # 1

yy = mydict['NARNAR']                # KeyError
```

The above code results in this exception:

```
Traceback (most recent call last):
  File "./keyerrortest.py", line 7, in
    yy = mydict['NARNAR']
KeyError: 'NARNAR'
```

As you can see, the line of code and the key involved in the error are displayed. If you get this error the debugging procedure should be to check to see first whether the key seems to be in the dict or not; if it is, you may want to check the type of the object, since string **'1'** is not the same as int **1**. String case also matters when Python is looking for a key -- the object value must match the dict's key exactly.

One way to handle an error like this is to test the dict ahead of time using the **in** operator:

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

if 'NARNAR' not in mydict:
    yy = 0
else:
    yy = mydict['NARNAR']    # else block only executed
                             # if key NARNAR exists in the dict
```

Another approach is the use the **dict get()** method:

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

yy = mydict.get('a', 0)            # 1

zz = mydict.get('zzz', 0)          # 0
```

Summary Task: read and write to a dict

Subscript syntax is used to add or read key/value pairs. The dict's *key* is the key!

Note well: *the syntax is the same* for setting a key/value pair *or* getting a value based on a key.

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

Getting a value from a dict based on a key

```
val  = mydict['a']       # get a value using a key:  1
val2 = mydict['b']       # get a value using a key:  2
```

Summary Task: Looping through a Dict with *for*

Looping through a dict means looping through its *keys*.

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

for key in mydict:
    print key        # prints 'a', then 'c', then 'b', then 'd'
```

As with all standard container features (e.g. **in**, **for**, **sorted()**) the **in** operator test the **keys** of the dict.

Of course having a key means we can get the value -- here we loop through and print each key *and* value in the dict:

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

for key in mydict:
    print "key: {}; value: {}".format(key, mydict[key])

    ### key: a; value 1
    ### key: c; value 3
    ### key: b; value 2
    ### key: d; value 4
```

Summary Task: Checking for Membership in a Dict with *in*

Checking membership in a dict means checking for the presence of a *key*.

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

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

As with all standard container features (e.g. **in**, **for**, **sorted()**) the **in** operator test the **keys** of the dict.

Summary Task: Dict Size with len()

len() counts the *pairs* in a dict.

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

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

Summary Task: Obtaining List of Keys, List of Values, List of key/value Items

keys(), **values()** return a list of objects; **items()** returns a list of *2-element tuples*.

keys(): return a list of keys in the dict

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

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

Of course once we have the keys, we can loop through and get the value for each key.

values(): return a list of values in the dict

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

The values cannot be used to get the keys - it's a one-way lookup from the keys. However, we might want to check for membership in the values, or sort the values, or some other less-used approach.

The dict items() method: pairs as a list of 2-element tuples

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

There are three elements here: three tuples. Each tuple contains two elements: a key and value pair from the dict. There are a number of reasons we might wish to use a structure like this -- for example, to sort the dictionary and store it in sorted form. As you know, a dictionary's keys are unordered, but a lists are not. A list of tuples can also be manipulated in other ways pertaining to a list. It is a convenient structure that is preferred by some developers as an alternative to working with the keys.

Review Summary Task: Sorting a Container with sorted()

With a list, tuple or set, **sorted()** returns a list of sorted elements

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

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

Remember that no matter what container is passed to **sorted()**, the function returns list -- even a string!

Summary Task: Reversing a Sort

reverse=True, a special arg to **sorted()**, reverses the order of a sort.

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

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

reverse is called a *keyword argument* -- it is no different from a regular *positional* arguments we have seen before; it is simply notated differently.

Summary Task: Sorting a Dict (sorting its *Keys*)

sorted() returns a sorted list of a dict's *keys*

```
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])
```

Practical: Build Up a Dict from Two Fields in a File

As with all containers, we loop through a data source, select and add to a dict.

```
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
```

Practical: Build a "Counting" or "Summing" Dictionary

We can use a dict's keys and associated values to create an aggregation (correlating a sum or count to each of a collection of keys).

```
state_count = dict()                                # initialize an empty dict
for line in open('/Users/dblaikie/Desktop/python_data/student_db.txt'):
    items = line.split(':')
    state = items[3]
    if state not in state_count:
        state_count[state] = 0
    state_count[state] = state_count[state] + 1

print "here is the count of states from the students.txt file:  "
for state in state_count :
    print "{}:  {} occurrences".format(state, state_count[state])

print "here is the count for 'NY':  "
print state_count['NY']                             # 4
```

Practical: Build and Read a Dict of Lists

A list can be added to a dict just like any other object.

Here we're keying a dict to student id, and setting a list of the remaining elements as its value:

```
ids_data = dict()                    # initialize an empty dict
for line in open('student_db.txt'):
    item_list = line.split(':') # note "multi-target assignment" of 5 elements
    id = item_list[0]
    data = item_list[1:]
    ids_data[id] = data              # key id is paired to student's state

print "here is the data for id 'jb29':  ", ids_names['jb29']    #
```

Practical: Working with dict items()

dict items() produces a 2-element tuple;

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

Boolean (True/False) Values

Introduction: Booleans (True/False) Values

Every object can be converted to boolean (True/False): an object is **True** if it is "non-zero".

```
counter = 5
if counter:                             # True
    print 'this int is not zero'
else:
    print 'this int is zero'


mystr = ''                              # empty string

if mystr:                               # False
    print 'this string has characters'
else:
    print 'this string is empty'


var = ['a', 'b', 'c']
if var:                                 # True
    print 'this container has elements'
else:
    print 'this container is empty'
```

Objectives for the Unit: Booleans

This *key/value pairs* container allows us to summarize data in powerful ways:

- Read any object in *boolean context* (i.e., with **if** or **while**) to see if it is empty

Summary for Object: Boolean

A *boolean* object can be True or False. All objects can be converted to boolean, meaning that all objects can be seen as True or False in a *boolean* context.

```
print type(True)              # <type 'bool'>
print type(False)             # <type 'bool'>
```

bool() converts objects to boolean.

```
print bool(['a', 'b', 'c'])    # True
print bool([])                # False
```

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 va


print bool(5)                 # True
print bool(0)                 # False
```

if and **while** induce the boolean conversion -- an object is evaluated in *boolean context*

```
mylist = [0, 0]

if mylist:                          # not empty, so True in boolean context
    print 'that list is not empty'


yourlist = [1, 2, 3]

while yourlist:            # True as long as yourlist has elements
    x = yourlist.pop()     # remove element from the end of yourlist
    print x
```

Boolean quiz

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
```

Complex Sorting

Introduction: Complex Sorting

We often sort a sequence of values by some other criteria.

Some values simply represent themselves - for example, a list of floats or a set of strings. Most other data that we work with, however, represent more than just the value itself.

For example if we are working with a list of filenames, we might want to consider the files by their names, or by their sizes (which is the biggest?), or by their last modification times (which is the latest?).

Or if we are working with a set of corporation names, we might be interested in the market cap, the revenue from last year, the change in stock price over the last quarter, etc. All of these are interesting attributes of the company, and we are often interested in ranking them according to one of these criteria (which company has the greatest market cap? which had the greatest revenue?)

So if we are sorting a list of items in a container, there are often multiple criteria by which we might want to sort.

The default sort we have begun to see with **sorted()** -- i.e., to sort strings by their alphabetic value or numbers by their numeric value -- must be rerouted so it uses *alternate criteria*.

This unit is about the mechanism by which we can accomplish this alternate sorting.

Objectives for the Unit: Complex Sorting

- Define custom **functions**: named blocks of Python code.
- Review sequence sorting using **sorted()**
- Review dictionary sorting (**sorted()** returns a list of keys)
- Use an *item criteria function* to indicate how items should be sorted
- Use built-in functions to indicate how items should be sorted

Summary Statement: Functions (user-defined)

A *user-defined function* is simply a named code block that can be called and executed any number of times.

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

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

Function argument(s)

A function's *arguments* are renamed in the function definition, and the function refers to them by these names.

```
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!'
```

(The argument objects are *copied* to the argument names -- they are the same objects.)

Function return value

A function's *return value* is passed back from the function with the **return** statement.

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

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

Summary Function (Review): sorted()

sorted() takes a sequence argument and returns a sorted list. The sequence items are sorted according to their respective types.

sorted() with numbers

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

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

sorted() with strings

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

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

Summary Task (Review): sorting a dictionary's keys

Sorting a **dict** means sorting the *keys* -- **sorted()** returns a list of sorted keys.

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}

sorted_keys = sorted(bowling_scores)

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

Indeed, any "listy" sort of operation on a **dict** assumes the keys: **for** looping, subscripting, **sorted()**; even **sum()**, **max()** and **min()**.

Summary Task (Review): sorting a dictionary's keys by its values

The dict **get()** method returns a value based on a key -- perfect for sorting keys by values.

```
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']


for player in sorted_keys:
        print "{} scored {}".format(player, bowling_scores[player])

        ##  zeb scored 98
        ##  jeb scored 123
        ##  janice scored 184
        ##  mike scored 202
```

Summary Feature: Custom sort using an *item criteria* function

An **item criteria** function returns to python the *value by which* a given element should be sorted.

Here is the same dict sorted by value in the same way as previously, through a custom *item criteria* function.

```
def by_value(dict_key):                      # a key to be sorted (for example, 'mike'
    dict_value = bowling_scores[dict_key]    # retrieving the value based on 'mike':  202
    return dict_value                        # returning the value 202

bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
sorted_keys = sorted(bowling_scores, key=by_value)

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

The dict's keys are sorted by value because of the **by_value()** function:

1. **sorted()** sees **by_value** referenced in the function call.
2. **sorted()** calls the **by_value()** *four times*: once with each key in the dict.
3. **by_value()** is called with **'jeb'** (which returns **123**), **'zeb'** (which returns **98**), **'mike'** (which returns **202**), and **'janice'** (which returns **184**).
4. The *return value* of the function is the *value by which* the key will be sorted

Therefore because of the return value of the function, **jeb** will be sorted by **123**, **zeb** by **98**, etc.

Summary Task: sort a numeric string by its numeric value

Numeric strings (as we might receive from a file) sort alphabetically:

```
numbers_from_file = ['1', '10', '3', '20', '110', '1000' ]
sorted_numbers = sorted(numbers_from_file)

print sorted_numbers    # ['1', '1000', '110', '20', '3'] (alphabetic sort)
```

To sort numerically, the item criteria function can convert to **int** or **float**.

```
def by_numeric_value(this_string):
    return int(this_string)

numbers_from_file = ['1', '10', '3', '20', '110', '1000' ]
sorted_numbers = sorted(numbers_from_file, key=by_numeric_value)

print sorted_numbers    # ['1', '3', '10', '20', '110', '1000']
```

Note that the values returned do not change; they are simply sorted by their integer equivalent.

Summary Task: sort a string by its case-insensitive value

Python string sorting sorts uppercase before lowercase:

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

To sort "insensitively", the item criteria function can lowercase each string.

```
def by_lowercase(my_string):
    return my_string.lower()

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

Summary Task: sort a string by a portion of the string

To sort a string by a portion of the string (for example, the last name in these 2-word names), we can split or slice the string and return the portion.

```
full_names = ['Jeff Wilson', 'Abe Zimmerman', 'Zoe Apple', 'Will Jefferson']

def by_last_name(fullname):
    fname, lname = fullname.split()
    return lname

sfn = sorted(full_names, key=by_last_name)

print sfn                                       #  ['Zoe Apple',
                                                #   'Will Jefferson',
                                                #   'Jeff Wilson',
                                                #   'Abe Zimmerman']
```

Summary Task: sort a file line by a field within the line

To sort a string of fields (for example, a CSV line) by a field within the line, we can **split()** and return a field from the split.

```
def by_third_field(this_line):
    els = this_line.split(',')
    return els[2]

lines = open('../python_data/students.txt')
sorted_lines = sorted(lines, key=by_third_field)
print sorted_lines


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

Summary Task: custom sort using built-in functions

Built-in functions can be used to indicate item sort criteria in the same way as custom functions -- by telling Python to pass an element and sort by the return value.

len() returns string length - so it can be used to sort strings by length

```
mystrs = ['angie', 'zachary', 'zeb', 'annabelle']

print sorted(mystrs, key=len)       # ['zeb', 'angie', 'zachary', 'annabelle']
```

Using a builtin function

os.path.getsize() returns the byte size of any file based on its name (in this example, in the *present working directory*):

```
import os

print os.path.getsize('test.txt')    # return 53, the byte size of test.txt
```

To sort files by their sizes, we can simply pass this function to **sorted()**

```
import os

files = ['test.txt', 'myfile.txt', 'data.csv', 'bigfile.xlsx']   # some files in my current dir

size_files = sorted(files, key=os.path.getsize)                  # pass each file to getsize()

for this_file in size_files:
        print "{}:  {} bytes".format(this_file, os.path.getsize(this_file))
```

(Please note that this will only work if your terminal's *present working directory* is the same as the files being sorted. Otherwise, you would have to prepend the path -- see **File I/O**, later in this course.)

Using methods

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

Using methods called on existing objects

```
companydict = {'IBM': 18.68, 'Apple': 50.56, 'Google': 21.3}

revc = sorted(companydict, key=companydict.get)       # ['IBM', 'Google', 'Apple']
```

Sidebar: cascading sort

Sort a list by multiple criteria by having your sort function return a 2-element tuple.

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

names = ['Zeb Will', 'Deb Will', 'Joe Max', 'Ada Max']

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

Reading Multidimensional Containers

Introduction: Reading Multidimensional Containers

Data can be expressed in complex ways using *nested* containers.

Real-world data is often more complex in structure than a simple sequence (i.e., a list) or a collection of pairs (i.e. a dictionary).

- A student database of unique student ids, with several address and billing fields associated with each id
- A listing of businesses with market cap, revenue, etc. associated with each business
- A list of devices on a network, each with attributes associated with each (id, firmware version, uptime, latency)
- A log listing of events on a web server, with attributes of the event (time, requesting ip, response code) for each
- A more complex nested structure as might be expressed in an XML or HTML file Complex data can be structured in Python through the use of *multidimensional containers*, which are simply containers that contain other containers (lists of lists, lists of dicts, dict of dicts, etc.) in structures of *arbitrary complexity*. Most of the time we are not called upon to handle structures of greater than 2 dimensions (lists of lists, etc.) although some config and data transmitted between systems (such as API responses) can go deeper. In this unit we'll look at the standard 2-dimensional containers we are more likely to encounter or want to build in our programs.

Objectives for the Unit: Reading Multidimensional Containers

- Identify and visually/mentally parse 2-dimensional structures: *lists of lists*, *lists of dicts*, *dicts of dicts* and *dicts of lists*
- Read any value within a multidimensional structure through chained subscripts
- Loop through and print selected values within a multidimensional structure

Summary Structure: List of Lists

A list of lists provides a "matrix" structure similar to an Excel spreadsheet.

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]
```

Probably used more infrequently, a list of lists allows us to access values through list methods (looping and indexed subscripts).

The "outer" list has 3 items -- each items is a list, and each list represents a row of data.

Each row list has 4 items, which represent the row data from the Fama-French file: the date, the Mkt-RF, SMB, HML and RF values.

Looping through this structure would be very similar to looping through a delimited file, which after all is an iteration of lines that can be split into fields.

```
for rowlist in value_table:
    print "the MktRF for {} is {}".format(rowlist[0], rowlist[1])
```

Summary Structure: List of Dicts

A list of dicts structures tabular rows into field-keyed dictionaries.

```
value_table = [
                { 'date': '19260701', 'MktRF': 0.09, 'SMB': -0.22, 'HML': -0.30, 'RF': 0.009 },
                { 'date': '19260702', 'MktRF': 0.44, 'SMB': -0.35, 'HML': -0.08, 'RF': 0.009 },
                { 'date': '19260706', 'MktRF': 0.17, 'SMB': 0.26,  'HML': -0.37, 'RF': 0.009 }
              ]
```

The "outer" list contains 3 items, each being a dictionary with identical keys. The keys in each dict correspond to field / column labels from the table, so it's easy to identify and access a given value within a row dict.

A structure like this might look elaborate, but is very easy to build from a data source. The convenience of named subscripts (as contrasted with the numbered subscripts of a list of lists) lets us loop through each row and name the fields we wish to access:

```
for rowdict in value_table:
    print "the MktRF for {} is {}".format(rowdict['date'], rowdict['MktRF'])
```

Summary Structure: Dict of Lists

A dict of lists allows association of a sequence of values with unique keys.

```
value_table = { '1926': [  0.09,  0.44,  0.17, -0.15, -0.06, -0.55,  0.61,  0.05, 0.51 ],
                '1927': [ -0.97,  0.30,  0.13, -0.18,  0.31,  0.39,  0.14, -0.27, 0.05 ],
                '1928': [  0.43, -0.14, -0.71,  0.61,  0.13, -0.88, -0.85,  0.12, 0.48 ]  }
```

The "outer" dict contains 3 string keys, each associated with a list of float values -- in this case, the MktRF values from each of the trading days for each year (only the first 9 are included here for clarity).

With a structure like this, we can perform calculations like those we have done on this data for a given year, namely to identify the **max()**, **min()**, **sum()**, average, etc. for a given year

```
for year in value_table:
    print 'for year {}:  len {}, sum {}, avg {}'.format(year,
                                                       len(value_table[year]),
                                                       sum(value_table[year]),
                                                       (sum(value_table[year]) / len(value_table[year])))
```

Summary Structure: Dict of Dicts

In a dict of dicts, each unique key points to another dict with keys and values.

```
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 },
}
```

The "outer" dict contains string keys, each of which is associated with a dictionary -- each "inner" dictionary is a convenient key/value access to the fields of the table, as we had with a list of dicts.

Again, this structure may seem complex (perhaps even needlessly so?). However, a structure like this is extremely easy to build and is then very convenient to query. For example, the 'HML' value for July 2, 1926 is accessed in a very visual way:

```
print date_values['19260702']['HML']        # -0.08
```

Summary Structure: arbitrary dimensions

Containers can nest in "irregular" configurations, to accomodate more complex orderings of data.

See if you can identify the object type and elements of each of the containers represented below:

```
conf = [
  {
    "domain": "www.example1.com",
    "database": {
      "host": "localhost1",
      "port": 27017
    },
    "plugins": [
       "plugin1",
       "eslint-plugin-plugin1",
       "plugin2",
       "plugin3"
    ]
  },   # (additional dicts would follow this one in the list)
]
```

Above we have a list with one item! The item is a dictionary with 3 keys. The "domain" key is associated with a string value. The "database" key is associated with another dictionary of string keys and values. The "plugins" key is associated with a list of strings.

Presumably this "outer" list of dicts would have more than one item, and would be followed by additional dictionaries with the same keys and structure as this one.

Summary Task: retrieving an "inner" element value

Nested subscripts are the usual way to travel "into" a nested structure to obtain a value.

A list of lists

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]

print "SMB for 7/3/26 is {}".format(value_table[2][2])
```

A dict of dicts

```
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 },
}

MktRF_thisday = date_values['19260701']['MktRF']   # value is 0.09

print date_values['19260701']['SMB']               # -0.22
print date_values['19260701']['HML']               # -0.3
```

Summary Task: looping through a complex structure

Looping through a nested structure often requires an "inner" loop within an "outer" loop.

looping through a list of lists

```
value_table =       [
                       [ '19260701', 0.09, -0.22, -0.30, 0.009 ],
                       [ '19260702', 0.44, -0.35, -0.08, 0.009 ],
                       [ '19260703', 0.17, 0.26,  -0.37, 0.009 ]

                    ]

for row in value_table:
    print "MktRF for {} is {}".format(row[0], row[1])
```

looping through a dict of dicts

```
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 },
}

for this_date in date_values:
    print "MktRF for {} is {}".format(this_date, date_values[this_date]['MktRF'])
```

Multidimensional structures - building

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 values
  outer_dict[date] = values                              # so values is a list, assigned as value to key date
                                                         # thus we have built a dictionary of lists (each key p
```

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 'inn
  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 value to
```

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.

Summary Function: pprint

pprint() prints a complex structure in readable format.

```
import pprint

dvs = {'19260701': {'HML': -0.3, 'RF': 0.009, 'MktRF': 0.09, 'SMB': -0.22}, '19260702': {'HML': -0.08, 'RF': 0.(

pprint.pprint(dvs)

       ### {'19260701': {'HML': -0.3, 'MktRF': 0.09, 'RF': 0.009, 'SMB': -0.22},
       ###  '19260702': {'HML': -0.08, 'MktRF': 0.44, 'RF': 0.009, 'SMB': -0.35}}
```

Set Operations, List Comprehensions, Lambdas and Sorting Multidimensional Structures

Advanced Container Processing

This week we will complete our tour of the core Python data processing features.

So far we have explored the reading and parsing of data; the loading of data into built-in structures; and the aggregation and sorting of these structures.

This session explores advanced tools for container processing.

set operations

```
a = set(['a', 'b', 'c'])
b = set(['b', 'c', 'd'])
print a.difference(b)      # set(['a'])
print a.union(b)           # set(['a', 'b', 'c', 'd'])
print a.intersection(b)    # set(['b', 'c'])
```

list comprehensions

```
a = ['hello', 'there', 'harry']
print [ var.upper() for var in a if var.startswith('h') ]
                           # ['HELLO', 'HARRY']
```

lambda functions

```
names = ['Joe Wilson', 'Pete Johnson', 'Mary Rowe']
sorted_names = sorted(names, key=lambda x: x.split()[1])
print sorted_names             # ['Pete Johnson', 'Mary Rowe', 'Joe Wilson']
```

ternary assignment

```
rev_sort = True if user_input == 'highest' else False

pos_val = x if x >= 0 else x * -1
```

conditional assignment

```
val = this or that       # 'this' if this is True else 'that'
val = this and that      # 'this' if this is False else 'that'
```

Container processing: Set Comparisons

We have used the **set** to create a unique collection of objects. The **set** also allows comparisons of sets of objects. Methods like **set.union** (complete member list of two or more sets), **set.difference** (elements found in this set not found in another set) and **set.intersection** (elements common to both sets) are fast and simple to use.

```
set_a = set([1, 2, 3, 4])
set_b = set([3, 4, 5, 6])

print set_a.union(set_b)          # set([1, 2, 3, 4, 5, 6])  (set_a + set_b)
print set_a.difference(set_b)     # set([1, 2])              (set_a - set_b)
print set_a.intersection(set_b)   # set([3, 4])  (what is common between them?)
```

List comprehensions: filtering a container's elements

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 MktRF 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 ])
```

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".

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

File, Directory and External Program I/O

Introduction: File, Directory and External Program I/O; Exceptions

Our programs don't always work in isolation -- they can read directories, write to files and run external programs.

This unit is about the "outside world" of your computer's operating system -- its files and directories and other programs running on it, the *STDOUT* and *STDIN* data streams that can pass data between them, as well as the *command line*, which is the prompt from which we have been running our Python programs.

The data we have been parsing has been accessed by us from a specific location, but often we are called upon to marshal data from many locations in our filesystem. We may also need to search for these files.

We also sometimes need to be able to read data produced by programs that reside on our filesystem. Our python scripts can run Unix or Windows utilities, installed programs, and even other python programs from within our running Python script, and capture the output.

Objectives for the Unit: File, Directory and External Program I/O

- take input at the command line with *sys.argv*
- write to and append to files with the **file** object
- list files in a directory with *os.listdir()*
- learn about file metatdata using *os.path.isfile()* and *os.path.isdir()*, *os.path.getsize()*
- traverse a tree of directories and files with *os.walk()*
- interact with files and other programs at the command line with *STDIN* ("standard in") and *STDOUT* ("standard out")
- launch external programs with *subprocess*

Summary structure: sys.argv

sys.argv is a **list** that holds strings passed at the command line

sys.argv example

a python script *myscript.py*

```
import sys                          # import the 'system' library

print 'first arg: ' + sys.argv[1]   # print first command line arg
print 'second arg: ' + sys.argv[2]  # print second command line arg
```

running the script from the command line

```
$ python myscript.py hello there
first arg: hello
second arg: there
```

sys.argv is a list that is *automatically provided by the **sys** module*. It contains any *string arguments to the program* that were entered at the command line by the user.

If the user does not type arguments at the command line, then they will not be added to the **sys.argv** list.

sys.argv[0]

sys.argv[0] always contains the name of the program itself
Even if no arguments are passed at the command line, **sys.argv** always holds one value: a string containing the program name (or more precisely, the pathname used to invoke the script).

example runs

a python script *myscript2.py*

```
import sys                           # import the 'system' library

print sys.argv
```

running the script from the command line (passing 3 arguments)

```
$ python myscript2.py hello there budgie
['myscript2.py', 'hello', 'there', 'budgie']
```

running the script from the command line (passing no arguments)

```
$ python myscript2.py
['myscript2.py']
```

Summary Exception: IndexError with sys.argv (when user passes no argument)

An **IndexError** occurs when we ask for a list index that doesn't exist. If we try to read **sys.argv**, Python can raise this error if the arg is not passed by the user.

a python script *addtwo.py*

```
import sys                          # import the 'system' library

firstint = int(sys.argv[1])
secondint = int(sys.argv[2])

mysum = firstint + secondint

print 'the sum of the two values is {}'.format(mysum)
```

running the script from the command line (passing 2 arguments)

```
$ python addtwo.py 5 10
the sum of the two values is 15
```

exception! running the script from the command line (passing no arguments)

```
$ python addtwo.py
Traceback (most recent call last):
  File "addtwo.py", line 3, in
firstint = int(sys.argv[1])
IndexError: list index out of range
```

The above error occurred because the program asks for items at subscripts **sys.argv[1]** and **sys.argv[2]**, but because no elements existed at those indices, Python raised an **IndexError** exception.

How to handle this exception? Test the **len()** of **sys.argv**, or trap the exception (see "Exceptions", coming up).

Summary task: writing and appending to files using the file object

Files can be opened for writing or appending; we use the **file** object and the **file write()** method.

```
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']
```

Note that we are explicitly adding newlines to the end of each line. The **write()** method doesn't do this for us.

Summary task: redirecting the STDOUT data stream

STDOUT is the 'output pipe' from our program (usually the screen, but it can be redirected to a file or other program).

hello.py: print a greeting

```
#!/usr/bin/env ptython

print 'hello, world!'
```

redirecting STDOUT to a file at the command line:

```
mycomputer$  python hello.py
hello, world!                              # default:  to the screen
mycomputer$  python hello.py > newfile.txt   # > redirect to a file (not the screen)


mycomputer$  cat newfile.txt                # cat spits out a file's contents
hello, world!

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

STDOUT is something we have been using all along. Any **print** statement sends string data to **STDOUT**. It is simply the conduit that allows us to send data out of our program.

In the above example, we first run the program (which prints a greeting). We see that this prints to the screen. Next, we use the **>** *redirection operator* (which is an operating system command, not a Python feature) to redirect **STDIN** to a file instead of the screen. This is why we don't see the output to the screen - it has been redirected.

redirecting STDOUT to the input of another program at the command line:

```
mycomputer$  python hello.py
hello, world!                       # default:  to the screen

mycomputer$  python hello.py | wc   # "piped" redirect to the wc utility

       1       2      14            # the output of wc
```

In the above example, we are *piping* the output of **hello.py** to another program: the **wc** utility, which counts lines, words and letters. **wc** can work with a filename, but it can also work with **STDIN**. In this case we see that **hello, world!** has **1** line, **2** words and **14** 14 characters.

Summary task: writing to STDOUT in different ways

We can use **print**, **print** with a comma, or **sys.stdout.write()** to write to **STDOUT**

print: print a newline after each statement

```
print 'hello, world!'
print 'how are you?'

                          # hello, world!
                          # how are you?
```

print with a comma: print a space after each statement

```
print 'hello, world!',
print 'how are you?'

                        # hello, world! how are you?
```

sys.stdout.write(): print nothing after each statement

```
import sys

sys.stdout.write('hello, world!')
sys.stdout.write('how are you?')

                        # hello, world!how are you?
```

Summary task: reading and redirecting the STDIN data stream

STDIN is the 'input pipe' to our program (usually the keyboard, but can be redirected to read from a file or other program).

```
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
```

The **stdin** datastream can also come from another program and *piped* into the program:

```
mycomputer$ ls -l | python readfile.py     # unix

mycomputer$ 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.

Summary task: read directories with os.listdir()

os.listdir() can read any directory, but the filename must be appended to the directory path in order for Python to find it.

```
import os                                       # os ('operating system') module talks to the os (for file acc
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):

    item_path = os.path.join(mydirectory, item)

    print item_path                             # /Users/dblaikie/photos/
                                                # /Users/dblaikie/backups/
                                                # /Users/dblaikie/college_letter.docx
                                                # /Users/dblaikie/notes.txt
                                                # /Users/dblaikie/finances.xlsx
```

Here we see all the files in my home direcctory on my mac (**/Users/dblaikie**). We must use **os.path.join()** to join the path to the file to see the whole path to the file.

os.path.join() is designed to take any two or more strings and insert a directory slash between them. It is preferred over regular string joining or concatenation because it is aware of the operating system type and inserts the correct slash (forward slash or backslash) for the operating system.

Summary task: read directory listing type with os.path.isfile() and os.path.isdir()

os.path.isdir() and **os.path.isfile()** return **True** or **False** depending on whether a listing is a file or directory.

```
import os                                          # os ('operating system') module talks to the os (for file acc
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):

    item_path = os.path.join(mydirectory, item)

    if os.path.isdir(item_path):
        print "{}:  {}".format(item, 'directory')
    elif os.path.isfile(item_path):
        print "{}:  {}".format(item 'file')
                                                   # photos:  directory
                                                   # backups:  directory
                                                   # college_letter.docx:  file
                                                   # notes.txt:  file
                                                   # finances.xlsx:  file
```

Summary task: read file size with os.path.getsize()

os.path.getsize() takes a filename and returns the size of the file in bytes

```
import os                                          # os ('operating system') module talks to the os (for file acc
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):
    item_path = os.path.join(mydirectory, item)
    item_size = os.path.getsize(item_path)
    print "{}:  {} bytes".format(item_path, item_size)
```

Keep in mind that Python won't be able to find a file unless its path is prepended. This is why **os.path.join()** is so important.

Summary exception: OSError with os.listdir() (and a bad directory)

Python will raise an **OSError** exception if we try to read a directory or file that doesn't exist, or we don't have permissions to read.

```
import os

user_file = raw_input('please enter a filename:  ')    # user enters a file that doesn't exist

file_size = os.listdir(os.path.getsize(user_file))


Traceback (most recent call last):
    File "getsize.py", line 5, in
OSError:  No such file or directory:  'mispeld.txt'
```

How to handle this exception? Test to see if the file exists first, or trap the exception (see "Exceptions", coming up).

Summary module: launch an external program with *subprocess*

The **subprocess** module allows us to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

subprocess.call(): execute a command and output to STDOUT

The first argument is a list containing the command and any arguments. Here's a Unix example:

```
import subprocess

# output to STDOUT (usually the screen)
subprocess.call(['ls', '-l'])                 # Unix-specific
```

For Windows, we can instead call the **dir** Windows utility:

```
import subprocess

# output to STDOUT (usually the screen)
subprocess.call(['dir', '.'], shell=True)
```

Each of the arguments **stdout=**, **stderr=** and **stdin=** can point to an open filehandle for reading or writing. **stderr=STDOUT** will redirect stderr output to stdout.

```
# output to an open filehandle
subprocess.call(['ls', '-l'], stdout=open('outfile.txt', 'w'), stderr=STDOUT)    # Unix-specific

# read from a file
subprocess.call(['wc'], stdin=open('readfile.txt'))                              # Unix-specific
```

shell=True will execute directly through the shell. In this case the entire shell command including arguments must be passed in a single argument. Using this flag and executing the command through the shell means that shell expansions (like * and any other shell behaviors can be accessed.

```
subprocess.call(['ls *'], shell=True)           # Unix-specific; use dir
```

However, never use **shell=True** if the command is coming from an untrusted source (like web input) as shell access means that arbitrary commands may be executed.

subprocess.check_output(): execute a command and return the output to a string

```
var = subprocess.check_output(["echo", "Hello World!"])  # Unix-specific
print var                                                # Hello World!  (echo just echoes back a string)

out = subprocess.check_output(['wc', 'test.txt'])        # Unix-specific
print out           #       43     112     845 test.py   (this is wc output)
```

Windows:

```
var = subprocess.check_output(["dir", "."], shell=True)
print var          # prints file listing for the current directory
```

Many of the optional arguments are the same; **check_output()** simply returns the output rather than sending it to **STDOUT**.

Forking child processes with *multiprocessing*

forking allows a running program to execute multiple copies of itself. It is used in situations in which a single script process would take too long to complete. This sometimes happens when a script either spends a lot of time processing data, or waits on external processes it must call numerous times.

For example, consider a script that must send out many mail messages. The call to **sendmail** or similar program sometimes takes a second or so to complete, and if that is multiplied by many hundreds of such requests, total execution time may be several hours. If the sending of mail can be split among multiple processes, it can happen much faster.

This script forks itself into three child processes. The "parent" process (the original script execution) and each of the "child" processes (which are spawned and immediately directed to the function **f**) identifies itself and its parent and child by *process id*; looking at the output, note the relationship between these numbers in each of the processes.

```
from multiprocessing import Process
import os
import time

def info(title):                    # function for a process to identify itself
    print title
    if hasattr(os, 'getppid'):  # only available on Unix
        print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(childnum):                    # function for child to execute
    info('*Child Process {}*'.format(childnum))
    print 'now taking on time consuming task...'
    print
    print
    time.sleep(3)

if __name__ == '__main__':
    info('*Parent Process*')
    print; print
    procs = []
    for num in range(3):
        p = Process(target=f, args=(num,))  # a new process object
                                            # target is function f
        p.start()                           # new process is spawned
        procs.append(p)                     # collecting list of Process objects
    for p in procs:
        p.join()                            # parent waits for child to return
    print 'parent concludes'
```

Looking closely at the output, note that all three processes executed and that the parent didn't continue until it had heard back from each of them

```
*Parent Process*
parent process: 77233
process id: 77958


*Child Process 0*
parent process: 77958
process id: 77959
now taking on time consuming task...


*Child Process 1*
parent process: 77958
process id: 77960
now taking on time consuming task...


*Child Process 2*
parent process: 77958
process id: 77961
now taking on time consuming task...


parent concludes         # parent has waited for the 3 processes to return
```

Sidebar -- summary function: traverse a directory tree with os.walk()

os.walk<() visits every directory in a directory tree so we can list files and folders.

```
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

Introduction: Exceptions

Exception handling is a flow control mechanism -- rerouting program flow when an error occurs.

When a program encounters an error it is referred to as an *exception*.

Errors are endemic in any programming language, but we can broadly classify errors into two categories:

- *unanticipated errors* (caused by programmer error or oversight)
- *anticipatable errors* (caused by incorrect user input, environmental errors such as permissions or missing files, networking or process errors such as database failures, etc.) *Trapping exceptions* means deciding what to do when an anticipatable error occurs. When we trap an error using a *try/except* block, we have the opportunity to have our program respond to the error by executing a block of code. In this way, exception handling is another flow control mechanism: **if** this error occurs, **do** something about it.

Objectives for the Unit: Exceptions

- identify exception types (IndexError, KeyError, IOError, etc.)
- use *try:* blocks to identify code where an exception is anticipatable
- use *except:* blocks to specify the anticipatable exception and to provide code to be run in the event of an exception
- trap multiple exception types anticipatable from a **try:** block, and chain **except:** blocks to execute different code blocks depending on which exception type was raised.

Summary: Exceptions signify an error condition

Exceptions are *raised* when an error condition is encountered; this condition can be handled.

In each of these *anticipateable* errors below, the user can easily enter a value that is invalid and would cause an error if not handled. We are not handling these errors, but we should:

ValueError: when the wrong value is used with a function or statement

```
uin = raw_input('please enter an integer:  ')

intval = int(uin)               # user enters 'hello'

print '{} doubled is {}'.format(uin, intval*2)
```

KeyError: when a dictionary key cannot be found. Here we ask the user for a key in the dict, but they could easily enter the wrong key:

```
mydict = {'1972': 3.08, '1973': 1.01, '1974': -1.09}

uin = raw_input('please enter a year: ')    # user enters 2116

print 'mktrf for {} is {}'.format(uin, mydict[uin])


Traceback (most recent call last):
   File "getavg.py", line 4, in
KeyError: '2116'
```

IndexError: when a list index can't be found. Here we ask the user to enter an argument at the command line, but they could easily skip entering the argument:

```
import sys

user_input = sys.argv[1]              # user enters no arg at command line


Traceback (most recent call last):
   File "getarg.py", line 3, in
IndexError:  1
```

OSError: when a file or directory can't be found. Here we ask the user to enter a filename

```
import os

user_file = raw_input('please enter a filename:  ')    # user enters a file that doesn't exist

file_size = os.listdir(os.path.getsize(user_file))


Traceback (most recent call last):
    File "getsize.py", line 5, in
OSError:  No such file or directory:  'mispeld.txt'
```

In each of these situations we are working with a process that may make an error (in this case, the 'process' is the user). We can then say that these errors are *anticipatable* and thus can be handled by our script.

Summary statements: *try* block and *except* block

The **try:** block contains statements from which a potential error condition is anticipated; the **except:** block identifies the anticipated exception and contains statements to be excecuted if the exception occurs.

How to avoid an anticipatable exception?

- wrap the lines where the error is anticipated in a **try:** block
- define statements to be executed if the error occurs

```
try:
    firstarg = sys.argv[1]
    secondarg = sys.argv[2]
except IndexError:
    exit('error:  two args required')
```

This code anticipates that the user may not pass arguments to the script. If two arguments are not passed, then **sys.argv[1]** or **sys.argv[2]** will fail with an **IndexError** exception.

Summary technique: trapping multiple exceptions

Multiple exceptions can be trapped using a **tuple** of exception types.

```
try:
    firstarg = sys.argv[1]
    secondarg = sys.argv[2]

    firstint = int(firstarg)
    secondint = int(secondarg)

except (IndexError, ValueError):
    exit('error:  two int args required')
```

In this case, whether an **IndexError** or a **ValueError** exception is raised, the **except:** block will be executed.

Summary technique: chaining except: blocks

The same **try:** block can be followed by multiple **except:** blocks.

```
try:
    firstint = int(sys.argv[1])
    secondint = int(sys.argv[2])
except IndexError:
    exit('error:  two args required')
except ValueError:
    exit('error:  args must be ints')
```

The exception raised will be matched against each type, and the first one found will excecute its block.

User-Defined Functions

Introduction: User-Defined Functions

A **user-defined function** is a *named code block* -- very simply, a block of Python code that we can call by name. These functions are used and behave very much like built-in functions, except that we define them in our own code.

```
def addthese(val1, val2):  # function definition; argument signature
    valsum = val1 + val2
    return valsum          # return value

x = addthese(5, 10)        # function call; 2 arguments passed;
                           # return value assigned to x
print x                    # 15
```

There are two primary reasons functions are useful: to *reduce code duplication* and to *organize our code*:

Reduce code duplication: a named block of code can be called numerous times in a program, which means the same series of statements can be executed repeatedly, without having to type them out multiple times in the code.

Organize code: large programs can be difficult to read, even with helpful comments. Dividing code into named blocks allows us to identify the major steps our code can take, and see at a glance what steps are being taken and the order in which they are taken.

We have learned about using simple functions for sorting; in this unit we will learn about:

1) different ways to define function arguments
2) the "scoping" of variables within functions
3) the four "naming" scopes within Python

Objectives for the Unit: User-Defined Functions

- define functions that take *arguments* and return *return values*
- define functions that take *positional* and *keyword* arguments
- define functions that can take an *arbitrary number* of arguments
- learn about the four *variable scopes* and how scopes interact

Review: functions are *named code blocks*

The block is executed every time the function is called.

```
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.

Review: function argument(s)

Any argument(s) passed to a function are *aliased* to variable names inside the function definition.

```
def print_hello(greeting, person):    # 2 strings aliased to objects
                                      # passed in the call

    full_greeting = "{}, {}!".format(greeting, person)
    print full_greeting

print_hello('Hello', 'World')         # pass 2 strings:  prints "Hello, World!"
print_hello('Bonjour', 'Python')      # pass 2 strings:  prints "Bonjour, Python!"
print_hello('squawk', 'parrot')       # pass 2 strings:  prints "squawk, parrot!"
```

Review: the *return* statement returns a value

Object(s) are returned from a function using the **return** statement.

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

msg = print_hello('Bonjour', 'parrot')   # full_greeting
                                         # aliased to msg

print msg                                # 'Bonjour, parrot!'
```

Summary argument types: *positional* and *keyword*

Your choice of type depends on whether they are required.

positional: args are required and in particular order

```
def sayname(firstname, lastname):
    print "Your name is {} {}".format(firstname, lastname)

sayname('Joe', 'Wilson')    # passed two arguments:  correct

sayname('Joe')              # TypeError:  sayname() takes exactly 2 arguments (1 given)
```

keyword: args are not required, can be in any order, and the function specifies a default value

```
def sayname(lastname, firstname="Citizen"):
    print "Your name is {} {}".format(firstname, lastname)


sayname('Wilson', firstname='Joe')  # Your name is Joe Wilson

sayname('Wilson')                   # Your name is Citizen Wilson
```

Variable name *scoping* inside functions

Variable names initialized inside a function are *local* to the function, and not available outside the function.

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

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

Note that although the object associated with **a** is returned and assigned to **var**, the *name* **a** is not available outside the function. Scoping is based on names.

global variables (i.e., ones defined outside a function) are available both inside and outside functions:

```
var = 'hello global'

def myfunc():
    print var

myfunc()              # hello global
```

The four variable scopes: (L)ocal, (E)nclosing, (G)lobal and (B)uiltin

Variable scopes "overlay" one another; a variable can be "hidden" by a same-named variable in a "higher" scope.

From top to bottom:

- **Local**: local to (defined in) a function
- **Enclosing**: local to a function that may have other functions in it
- **Global**: available anywhere in the script (also called *file scope*)
- **Built-in**: a built-in name (usually a function like **len()** or **str()**) A variable in a given scope can be "hidden" by a same-named variable in a scope above it (see example below):

```
def myfunc():
    len = 'inside myfunc'  # local scope:  len is initialized in the function
    print len

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

len = 'in global scope'    # assigned in global scope:  a global variable
print len                  # global scope:  prints 'in global scope'

myfunc()                   # prints 'inside myfunc' (i.e. the function executes)

print len                  # prints 'in global scope' (the local len is gone, so we see the global

del len                    # 'deletes' the global len
print len                  # prints '<built-in function len>'
```

Summary exception: UnboundLocalError

An **UnboundLocalError** exception signifies a local variable that is "read" before it is defined.

```
x = 99
def selector():
    x = x + 1       # "read" the value of x; then assign to x
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
```

Remember that a *local* variable is one that is initialized or assigned inside a function. In the above example, **x** is a local variable. So Python sees **x** not as the global variable (with value **99**) but as a local variable. However, in the process of initializing **x** Python attempts to *read* **x**, and realizes that is hasn't been initialized yet -- the code has attempted to *reference* (i.e., read the value of) **x** before it has been assigned.

Since we want Python to treat **x** as the global **x**, we need to tell it to do so. We can do this with the **global** keyword:

```
x = 99
def selector():
    global x
    x = x + 1
selector()
print x             # 100
```

Modules

Introduction: Modules

Modules are files that contain reusable Python code: we often refer to them as "libraries" because they contain code that can be used in other scripts.

It is possible to *import* such library code directly into our programs through the **import** statement -- this simply means that the functions in the module are made available to our program.

Modules consist principally of functions that do useful things and are grouped together by subject. Here are some examples:

- The **sys** module has functions that let us work with python's interpreter and how it interacts with the operating system
- The **os** module has functions that let us work with the operating system's files, folders and other processes
- The **datetime** module has functions that let us easily calculate date into the future or past, or compare two dates
- The **urllib2** module has functions that let us easily make HTTP requests over the internet So when we **import** a module in our program, we're simply making *other Python code* (in the form of functions) available to our own programs. In a sense we're creating an assemblage of Python code -- some written by us, some by other people -- and putting it together into a single program. The imported code doesn't literally become part of our script, but it is part of our program in the sense that our script can call it and use it. We can also define *our own modules* -- collections of Python functions and/or other variables that we would like to make available to our other Python programs. We can even prepare modules designed for others to use, if we feel they might be useful. In this way we can collaborate with other members of our team, or even the world, by using code written by others and by providing code for others to use.

Objectives for the Unit: Modules

- save a file of Python code and import it into and use it in another Python program
- import individual functions from a module into our Python program
- access a module's functions as its attributes
- manipulate the module's *search path*
- write Python code that can act both as a module and a script
- raise exceptions in our module code, to be handled by the calling code
- design modules with an eye toward reuse by others
- install modules with **pip** or **easy_install**

Summary Statement: import *modulename*

Using **import**, we can import an entire Python module into our own code.

messages.py: a Python *module* that prints messages

```
import sys

def print_warning(msg):
    """write a message to STDOUT"""
    sys.stdout.write('warning:  {}\n'.format(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')
```

test.py: a Python *script* that imports **messages.py**

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

import messages

print "test program running..."

messages.log_message('this is an important message')
messages.print_warning("I think we're in trouble.")
```

The *global variables* in the module become *attributes* of the module. The module's variables are accessible through the name of the module, as its attributes.

Summary statement: import *modulename* as *convenientname*

A module can be renamed at the point of import.

```
import pandas as pd
import datetime as dt

users = pd.read_table('myfile.data', sep=',', header=None)

print "yesterday's date:  {}".format(dt.date.today() - dt.timedelta(days=1))
```

Summary statement: from *modulename* import *variablename*

Individual variables can be imported by name from a module.

```
#!/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.")
```

Summary: module search path

Python must be told where to find our own custom modules.

Python's standard module directories

When it encounters an **import**, Python searches for the module in a selected list of standard module directories. It does not search through the entire filesystem for modules. Modules like **sys** and **os** are located in one of these standard directories.

Our own custom module directories

Modules that we create should be placed in one or more directories that we designate for this purpose. In order to let

Python know about our own module directories, we have a couple of options:

PYTHONPATH environment variable

The standard approach to adding our own module directories to the list of those that Python searches is to create or modify the PYTHONPATH environment variable. This colon-separated list of paths indicates any paths to search *in addition* to the ones Python normally searches.

In your Unix .bash_profile file in your home directory, you would place the following command:

```
export PYTHONPATH=$PYTHONPATH:/path/to/my/pylib:/path/to/my/other/pylib
```

...where **/path/to/my/pylib** and **/path/to/my/other/pylib** are paths

In Winows, you can set the **PYTHONPATH** environment variable through the Windows GUI.

manipulating *sys.path*

```
import sys

print sys.path

    # ['', '/Users/dblaikie/lib', '//anaconda/lib/python2.7', '//anaconda/lib/python2.7/plat-darwin',
    #  '//anaconda/lib/python2.7/plat-mac', '//anaconda/lib/python2.7/plat-mac/lib-scriptpackages',
    #  '//anaconda/lib/python2.7/lib-tk', '//anaconda/lib/python2.7/lib-old',
    #  '//anaconda/lib/python2.7/lib-dynload', '//anaconda/lib/python2.7/site-packages',
    #  '//anaconda/lib/python2.7/site-packages/PIL',
    #  '//anaconda/lib/python2.7/site-packages/Sphinx-1.3.1-py2.7.egg',
    #  '//anaconda/lib/python2.7/site-packages/aeosa',
    #  '//anaconda/lib/python2.7/site-packages/setuptools-19.1.1-py2.7.egg']

sys.path.append('/path/to/my/pylib')
```

Once a python script is running, Python makes the PYTHONPATH search path available in a list called **sys.path**. Since it is a list, it can be manipulated; you are free to add whatever paths you wish

Summary: coding all Python scripts as modules

All Python scripts should be coded as modules: able to be both imported and executed.

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


""" helloworld.py:  print the "Hello! message """

def print_hello(arg):
    print "Hello, {}!".format(arg)

def main():
    print_hello('World')

if __name__ == '__main__':       # value is '__main__' if this script is executed
                                 # value is 'helloworld' if this script is imported
    main()
```

If we run the above script, we'll see "Hello, World!". But if we *import* the above script, we won't see anything. Why? Because the **if** statement above will be true *only if the script is executed*.

This is important behavior, because there are some scripts that we may want to run directly, but also allow others to import (in order to use the script's functions).

Whether we intend to import a script or not, it is considered a "best practice" to build **all of our programs** in this way -- with a "main body" of statements collected under function **main()** and the call to ain() inside the **if __name__ == '__main__'** gate.

Summary: raising exceptions

Causing an exception to be raised is the principal way a module signals an error to the importing script.

A file called **mylib.py**

```
def get_yearsum(user_year):

    user_year = int(user_year)
    if user_year < 1929 or user_year > 2013:
        raise ValueError('year {} out of range'.format(user_year))

    # calculate value for the year

    return 5.9             # returning a sample value (for testing purposes only)
```

An exception raised by us is indistinguishable from one raised by Python, and we can raise any exception type we wish.

This allows the user of our function to handle the error if needed (rather than have the script fail):

```
import mylib

while True:

    year = raw_input('please enter a year:  ')

    try:
        mysum = mylib.get_yearsum(year)
        break
    except ValueError:
        print 'invalid year:  try again'

print 'mysum is', mysum
```

Summary: installing modules

Third-party modules must be downloaded and installed into our Python distribution.

Unix

```
$ sudo pip search pandas             # searches for pandas in the PyPI repository
$ sudo pip install pandas            # installs pandas
```

Installation on Unix requires something called *root permissions*, which are permissions that the Unix system administrator uses to make changes to the system. The below commands include **sudo**, which is a way to temporarily be granted root permissions.

Windows

```
C:\Windows > pip search pandas       # searches for pandas in the PyPI repository
C:\Windows > pip install pandas      # installs pandas
```

PyPI: the Python Package Index

The Python Package Index at https://pypi.python.org/pypi (https://pypi.python.org/pypi) is a repository of software for the Python programming language. There are more than 70,000 projects uploaded there, from serious modules used by millions of developers to half-baked ideas that someone decided to share prematurely.

Usually, we encounter modules in the field -- shared through blog posts and articles, word of mouth and even other Python code. But the PPI can be used directly to try to find modules that support a particular purpose.

Summary: the Python *standard distribution* of modules

Modules included with Python are installed when Python is installed -- they are always available.

Python provides hundreds of supplementary modules to perform myriad tasks. The modules do not need to be installed because they come bundled in the Python distribution, that is they are installed at the time that Python itself is installed.

The documentation for the standard library is part of the official Python docs (https://docs.python.org/2/library/index.html).

- various string-related services
- specialized containers (type-specific lists and dicts, pseudohashes, etc.)
- math calculations and number generation
- file and directory manipulation
- persistence (saving data on disk)
- data compression and archiving (e.g., creating zip files)
- encryption
- networking and interprocess (program-to-program) communication
- internet tasks: web server, web client, email, file transfer, etc.
- XML and HTML parsing
- multimedia: audio and image file manipulation
- GUI (graphical user interface) development
- code testing
- etc.

From the Standard Distribution: the time module

time gives us simple access to the current time or any other time in terms of *epoch seconds*, or seconds since the *epoch* began, which was set by Unix developers at January 1, 1970 at midnight!

The module has a simple interface - it generally works with seconds (which are often a float in order to specify milliseconds) and returns a float to indicate the time -- this float value can be manipulated and then passed back into **time** to see the time altered and can be formatted into any display.

```
import time

epochsecs = time.time()       # 1450818009.925441 (current time in epoch seconds)

print time.ctime(epochsecs)   # 'Tue Apr 12 13:29:19 2016'

epochsecs = epochsecs + (60 * 60 *24)    # adding one day!

print time.ctime(epochsecs)   # 'Wed Apr 13 13:29:19 2016'

time.sleep(10)                # pause execution 10 seconds
```

datetime.date, datetime.datetime and datetime.timedelta

Python uses the 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).

```
import datetime as dt            # load the datetime library
dt = dt.datetime.now()           # create a new date object set to now
dt = dt.date.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 to subtract one day, one hour, one minute and one second from the date.

```
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                                # 2011-07-14 09:40:15.678331

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")    # 2011-07-13 08:39:14.678331
```

Classes

Introduction: Classes

Classes allow us to create a *custom type of object* -- that is, an object with its own *behaviors* and its own ways of storing *data*.

Consider that each of the objects we've worked with previously has its own behavior, and stores data in its own way: dicts store pairs, sets store unique values, lists store sequential values, etc.

An object's *behaviors* can be seen in its methods, as well as how it responds to operations like subscript, operators, etc.

An object's *data* is simply the data contained in the object or that the object represents: a string's characters, a list's object sequence, etc.

Objectives for this Unit: Classes

- Understand what classes, objects and attributes are and why they are useful
- Create our own classes -- our own object types
- Set *attributes* in objects and read attributes from objects
- Define *methods* in classes that can be used by objects
- Define object *initializers* with **__init__()**
- Use *getter* and *setter* methods to enforce *encapsulation*
- Understand class *inheritance*
- Understand *polymorphism*

Class Example: the *date* and *timedelta* object types

First let's look at object types that demonstrate the convenience and range of behaviors of objects.

A *date* object can be set to any date and knows how to calculate dates into the future or past.
To change the date, we use a *timedelta* object, which can be set to an "interval" of days to be added to or subtracted from a date object.

```
from datetime import date, timedelta

dt = date(1926, 12, 30)         # create a new date object set to 12/30/1926
td = timedelta(days=3)          # create a new timedelta object:  3 day interval

dt = dt + timedelta(days=3)     # add the interval to the date object:  produces a new date object

print dt                        # '1927-01-02' (3 days after the original date)


dt2 = date.today()              # as of this writing:  set to 2016-08-01
dt2 = dt2 + timedelta(days=1)   # add 1 day to today's date

print dt2                       # '2016-08-02'

print type(dt)                  # <type 'datetime.datetime'>
print type(td)                  # <type 'datetime.timedelta'>
```

Class Example: the proposed *server* object type

Now let's imagine a useful object -- this proposed class will allow you to interact with a server programmatically. Each server object represents a server that you can ping, restart, copy files to and from, etc.

```
import time
from sysadmin import Server


s1 = Server('blaikieserv')

if s1.ping():
    print '{} is alive '.format(s1.hostname)

s1.restart()                        # restarts the server

s1.copyfile_up('myfile.txt')        # copies a file to the server
s1.copyfile_down('yourfile.txt')    # copies a file from the server

print s1.uptime()                   # blaikieserv has been alive for 2 seconds
```

A *class* block defines an object "factory" which produces *objects* (*instances*) of the class.

Method calls on the object refer to functions defined in the class.

```
class Greeting(object):
    """ greets the user """

    def greet(self):
        print 'hello, user!'


c = Greeting()

c.greet()                   # hello, user!

print type(c)               # <class '__main__.Greeting'>
```

Each class *object* or *instance* is of a type named after the class. In this way, *class* and *type* are almost synonymous.

Each object holds an *attribute dictionary*

Data is stored in each object through its attributes, which can be written and read just like dictionary keys and values.

```
class Something(object):
    """ just makes 'Something' objects """

obj1 = Something()
obj2 = Something()

obj1.var = 5             # set attribute 'var' to int 5
obj1.var2 = 'hello'      # set attribute 'var2' to str 'hello'

obj2.var = 1000          # set attribute 'var' to int 1000
obj2.var2 = [1, 2, 3, 4] # set attribute 'var2' to list [1, 2, 3, 4]


print obj1.var           # 5
print obj1.var2          # hello

print obj2.var           # 1000
print obj2.var2          # [1, 2, 3, 4]

obj2.var2.append(5)      # appending to the list stored to attribute var2

print obj2.var2          # [1, 2, 3, 4, 5]
```

In fact the attribute dictionary is a real dict, stored within a "magic" attribute of the object:

```
print obj1.__dict__      # {'var': 5, 'var2': 'hello'}

print obj2.__dict__      # {'var': 1000, 'var2': [1, 2, 3, 4, 5]}
```

The class also holds an attribute dictionary

Data can also be stored in a class through class attributes or through variables defined in the class.

```
class MyClass():
    """ The MyClass class holds some data """

    var = 10              # set a variable in the class (a class variable)


MyClass.var2 = 'hello'    # set an attribute directly in the class object

print MyClass.var         # 10      (attribute was set as variable in class block)
print MyClass.var2        # 'hello' (attribute was set as attribute in class object)

print MyClass.__dict__    # {'var': 10,
                          #  '__module__': '__main__',
                          #  '__doc__': ' The MyClass class holds some data ',
                          #  'var2': 'hello'}
```

The additional **__module__** and **__doc__** attributes are automatically added -- **__module__** indicates the active module (here, that the class is defined in the script being run); **__doc__** is a special string reserved for documentation on the class).

object.attribute lookup tries to read from object, then from class

If an attribute can't be found in an object, it is searched for in the class.

```
class MyClass(object):
  classval = 10         # class attribute

a = MyClass()
b = MyClass()

b.classval = 99         # instance attribute of same name

print a.classval        # 10 - still class attribute
print b.classval        # 99 - instance attribute

del b.classval          # delete instance attribute

print b.classval        # 10 -- now back to class attribute
```

Method calls pass the object as first (implicit) argument, called *self*

Object methods or *instance methods* allow us to work with the object's data.

```
class Do(object):
    def printme(self):
        print self      # <__main__.Do object at 0x1006de910>

x = Do()

print x                 # <__main__.Do object at 0x1006de910>
x.printme()
```

Note that **x** and **self** have the same hex code. This indicates that they are the very same object.

Instance methods / object methods and object attributes: changing object "state"

Since instance methods pass the object, and we can store values in object attributes, we can combine these to have a method modify an object's values.

```
class Sum(object):
    def add(self, val):
        if not hasattr(self, 'x'):
            self.x = 0
        self.x = self.x + val

myobj = Sum()
myobj.add(5)
myobj.add(10)

print myobj.x      # 15
```

Objects are often modified using *getter* and *setter* methods

These methods are used to read and write object attributes in a controlled way.

```
class Counter(object):
    def setval(self, val):     # arguments are:  the instance, and the value to be set
        if not isinstance(val, int):
            raise TypeError('arg must be a string')

        self.value = val        # set the value in the instance's attribute

    def getval(self):          # only one argument:  the instance
        return self.value       # return the instance attribute value

    def increment(self):
        self.value = self.value + 1

a = Counter()
b = Counter()

a.setval(10)       # although we pass one argument, the implied first argument is a itself

a.increment()
a.increment()

print a.getval()   # 12


b.setval('hello')  # TypeError
```

__init__() is *automagically* called when a new instance is created

The *initializer* of an object allows us to set the initial attribute values of the object.

```
class MyCounter(object):
  def __init__(self, initval):   # self is implied 1st argument (the instance)
    try:
      initval = int(initval)     # test initval to be an int,
    except ValueError:           # set to 0 if incorrect
      initval = 0
    self.value = initval         # initval was passed to the constructor

  def increment_val(self):
    self.value = self.value + 1

  def get_val(self):
    return self.value

a = MyCounter(0)
b = MyCounter(100)

a.increment_val()
a.increment_val()
a.increment_val()

b.increment_val()
b.increment_val()

print a.get_val()    # 3
print b.get_val()    # 102
```

Classes can be organized into an an *inheritance tree*

When a class inherits from another class, attribute lookups can pass to the *parent* class when accessed from the *child*.

```
class Animal(object):
  def __init__(self, name):
    self.name = name
  def eat(self, food):
    print '%s eats %s' % (self.name, food)

class Dog(Animal):
  def fetch(self, thing):
    print '%s goes after the %s!' % (self.name, thing)

class Cat(Animal):
  def swatstring(self):
    print '%s shreds the string!' % (self.name)
  def eat(self, food):
    if food in ['cat food', 'fish', 'chicken']:
      print '%s eats the %s' % (self.name, food)
    else:
      print '%s:  snif - snif - snif - nah...' % self.name

d = Dog('Rover')
c = Cat('Atilla')

d.eat('wood')                 # Rover eats wood.
c.eat('dog food')             # Atilla:  snif - snif - snif - nah...
```

Conceptually similar methods can be unified through *polymorphism*

Same-named methods in two different classes can share a conceptual similarity.