All Slides on One Page

Python 2

home (../handouts.html)

Objects, Object Types, Operators and Functions

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 author of the code. All other names are built-in to Python.

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

Basic algebraic math operators work as expected; exponientation and modulus operators provide additional functionality

Simple math operations

```
var = 5
var2 = 15.3

var3 = var + var2    # 20.3

var4 = var2 - 0.3    # 15.0

var5 = var4 / 3      # 5.0
```

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
```

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**.

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**

Operators with Strings: + and *

The plus operator (+) with two strings returns a concatenated string; the times operator (*) *repeats a string* to produce a new string.

+ Operator with Two Strings: string concatenation

```
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 aa

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.)

Functions: len(), round(), type()

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
```

round() 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
```

type() 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.

"Behavioral" Functions: raw_input() and exit()

These functions cause the program to pause or exit.

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.

exit() 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 soon.

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                     # because of exit() above, this code will not be reached
```

"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.

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

These terms don't need to be memorized right away but you will eventually learn all of them. Using them may help focus your conceptualization of the material.

- 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

[This is the last slide in this section.]

Object Attributes and Methods

Object Attributes

Every object has attributes, and each attribute points to another object (string, method, function/method, etc.)

An object's attributes are accessed using *object.attribute* syntax:

```
var = 'hello'

print var.upper       # <method 'upper' of 'str' objects>
```

Python indicates that attribute **upper** is a *method* of the string **var**.

Every method is an attribute, but not all attributes are methods.

We can see a list of attributes accessible to an object using the **dir()** function.

```
var = 'hello'

print dir(var)
```

This prints:

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
 '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Object Methods

Methods are *custom functions* that can be 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() and format()

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
```

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 -- 34 years old.


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 "that line had a period at the end"
```

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

```
cc = raw_input('yes? ')
if cc.startswith('y') or cc.startswith('Y'):
    print 'thanks!'
else:
    print "ok, I guess not."
```

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 'in' 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")    # 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:  4.0


print len(letters) / letters.count("a")   # the above lines combined, prints 4.0


print float(letters.count("a")) / len(letters) * 100    # 25.0

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

Conditionals and Exceptions

Conditionals control program flow. Exceptions produce error conditions that allow program flow control.

The **if**, **elif** and **else** statements determine which parts of the program will be executed.

Exceptions indicate that an error condition has occurred. They can be **trapped**, allowing for program flow control when an error occurs.

```
while True:
    zz = raw_input('type an integer and I will tell you its sign:  ')
    try:
        zyz = int(zz)
    except TypeError:
        print 'error:  please enter an integer'
        continue
    break

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."
```

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.

'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.

'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*:

```
var = 5
var2 = 10
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 (infinite 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'
```

This *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 10 (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!

[This is the last slide in this section.]

Exceptions: SyntaxError, NameError, TypeError

An **Exception** is *raised* when you ask Python to do something that it can't understand, or that breaks one of its rules. When you encounter an exception for the first time, make sure to understand what Python is trying to tell you. Understanding each exception message will help you understand how to "think like Python" and avoid the problem in future.

SyntaxError: raised 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.

NameError: 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.

TypeError: 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.

Practical: Parsing tabular data

Table Data: Rows and Fields

Tables consist of *records* (rows) and *fields* (column values). Relational databases, CSV files and Excel files can all be processed using the same

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.

Summary Algorithm

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:

Summing a Column of Values

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.
Counting Lines, Words, Characters *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

file object: 3 ways to read strings from a file: **for** looping, **read()** and **readlines**

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']
```

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()**).

Summary: String Object

string object: 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'
```

rstrip() 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 Item Subscripting

subscripts select an item from a list

A list is a sequence of objects of any type. The objects are called *items*.

```
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\
```

Summary Algorithm

The summary algorithm allows for parsing of tabular structures.

```
# setting a summary variable
mysum = 0.0


# reading table records from a file as an iteration of string objects
fh = open('../python_data/FF_abbreviated.txt')   # file object allows looping through a
                                                 # series of strings
for bb in fh:

    print(type(bb))            # 'str'


    # when reading a file line-by-line, we should strip off the newline.
    xx = bb.rstrip()            # remove "whitespace" from end of the line
                                # and return a new string with the string removed


    # a string can be sliced by position
    id =  xx[0:4]               # '1926' (indices 0-4 of string xx)


    # the string split() method returns a list of strings,
    # each string a field in a single record (row or line from the table).
    yy = xx.split()

    print yy                    # ['19260701', '0.09', '0.22', '0.30', '0.009']



    # list subscripting:  each table record (row or line from the table) when rendered as a
    # list of strings (from split()) is addressable by index

    kk = yy[0]                  # '19260701' (first element:  index starts at 0)
    jj = yy[-1]                 # '0.009'    (last element:  negative index counts from end)


    # adding value from this row to summary variable
    mysum = mysum + float(jj)


# reporting value of sum
print mysum
```

Summary Algorithm: Steps in Detail

These standard steps are used to parse through a file in order to summarize some portion of its data.

Reading table *Records* from a file as an iteration of string objects

```
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 Each Line

When reading a file line-by-line, we should strip off the newline.

```
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
```

Slicing each line

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'
```

Parsing table *fields* from each file line string 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']
```

Selecting table fields 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'
```

Slicing a portion of a string 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'
```

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.

Arguments to a program with argparse

Formal program argument validation is provided by **argparse** module.

Besides validating your arguments and making them available as object attributes, the module will respond to the --help flag by summarizing available arguments as well as other help text that you can define.

Here's a simple example:

```
import argparse

parser = argparse.ArgumentParser()

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

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

# require an argument, also require a type
parser.add_argument('-a', '--anotheropt', action='store', required=True, type=int)

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

print args.mybooleanopt
print args.myvalueopt
```

Options are available to make some args required, to require a certain type of one of a set of valid values, and much more. See the docs here (https://docs.python.org/2.7/library/argparse.html#module-argparse).

range() and enumerate()

The **range()** function produces a *new list of consecutive integers* which can be used for counting.

```
intlist = range(10)

for val in intlist:
    print val,            # comma:  keep print output on one line
                          # 0 1 2 3 4 5 6 7 8 9

for val in range(5,10):
    print val,            # 5 6 7 8 9

print sum(range(100))     # sum up the values from 0 to 99
```

enumerate() takes any *iterable* (thing that can be looped over) and "marries" it to a range of integers, so you can keep a simultaneous count of something

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

for count, element in enumerate(mylist):
    print "element {}:  {}".format(count, element)
```

This can be handy for example with a filehandle. Since we can loop over a file, we can also pass it to **enumerate()**, which would render a line number with each line:

```
fh = open('file.txt')

for count, line in enumerate(fh):
    print "line {}:  {}".format(count, line.rstrip())

    ## (stripping the line above for clean-looking output)
```

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 38.2
print 'average is {}  '.format(sum(var) / len(var)) # average is 7.640000000000001

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[int(len(var) / 2)])   # median is 6.9
```

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

myset = {'a', 9999, 4.3}        # initialize a set with elements

myset = set(['a', 9999, 4.3])   # legacy approach:  past a list to set()
```

Add to a Set

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

myset.add(4.3)                  # note well method call not assigned
myset.add('a')

print myset                    # {'a', 4.3}     (order is not necessarily maintained)
```

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 myset:                    # test is True
    print "'c' is in myset"         # this will be printed
```

Loop through a Set

```
myset = set(['b', 'a', 'c'])
for el in myset:
    print el                        # may be printed in seeming '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 = {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 = {'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?
```

Strings can be seen as sequences of characters.

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.

```
var = 5

newvar = var.upper()

## Traceback (most recent call last):
##   File "", line 1, in
## AttributeError: 'int' object has no attribute 'upper'
```

We are trying to call the **upper()** method on an integer.

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('../python_data/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)     # count # of occurrences of this string in list of strings
print '{} occurs {} times'.format(chosen_state, state_freq)
```

The list **count()** method counts the number of times an item value (in this case, a string "state" value) appears in the list of state string values.

set: build a set of unique states

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

    elements = line.split(':')
    state_set.add(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
```

Practical: Parsing CSV or Database tabular 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()
```

Relational Database Queries and Result Sets

Database queries are parsed in the same manner as text files -- through 'for' looping.

Note that looping through file rows is the same as

```
import sqlite3
conn = sqlite3.connect('../python_data/sqlite3_trades.db')

curs = conn.cursor()

# Create table
curs.execute("""SELECT date, trans_type, qty, price FROM stocks WHERE symbol = 'AAPL'""")

for row in curs:
    print row       # 'row' is a tuple:  ('2016-03-09', 'BUY', '30', '135.9')

# Close the connection as a matter of habit
conn.close()
```

Since databases (and Excel files) are iterated through and parsed using the same approach as text files, this section's focus on parsing file lines prepares us to work with databases (and Excel files) as well.

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.google.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: urllib.urlencode()

When including parameters in our requests, we must *encode* them into our request URL. The **urlencode()** method does this nicely:

```
import urllib2
import urllib
params = urllib.urlencode({'choice1': 'spam and eggs', 'choice2': 'spam, spam, bacon and spam'})
print "encoded query string: ", params
f = urllib2.urlopen("http://www.google.com?{}".format(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>
```

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:  'NARNAR'
```

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: Dictionary Size with len()

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

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

print len(mydict)              # 3 (number of keys in dict)
```

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 = sorted(keys)
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 list of 2-item tuples. **dict()** can convert back to a dictionary.

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


newdict = dict(these_items)

print newdict                        # {'a': 1, 'b': 2, 'c': 3}
```

2-item tuples can be sorted and sliced, so they are a handy alternate structure.

zip() zips up paralell lists into tuples

```
list1 = ['a', 'b', 'c', 'd']
list2 = [ 1,   2,   3,   4 ]

tupes = zip(list1, list2)

newdict = dict(tupes)                # [('a', 1), ('c', 3), ('b', 2), ('d', 4)]
```

Occasionally we are faced with lists that relate to each other one a 1-to-1 basis... or, we sometimes even shape our data into this form. Paralell lists like these can be zipped into multi-item tuples.

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'
```

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 value is a **True** value

```
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
```

any() all() and in

any() with a sequence checks to see if any elements are True; **all()** asks whether they are all True; **in** can be used to see if a value exists in a sequence.

any(): are any of these True?

```
mylist = [0, 0, 5, 0]

if any(mylist):
    print 'at least one item is True'
```

all(): are all of these True?

```
mylist = [5, 5, 'hello', 5.0]

if all(mylist):
    print 'all items are True'
```

in with a list: is this value anywhere within this list?

```
mylist = ['highest', 'lowest']

user_choice = 'lowest'

if user_choice not in mylist:
    print 'please enter "higest" or "lowest"'
```

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']
```

You can use a method here in the same way you would use a function, except that you won't be specifying the specific object as you would normally with a method. To refer to a method "in the abstract", you can say str.upper or str.lower. However, make sure not to actually call the method (which is done with the parentheses). Instead, you simply refer to the method, i.e., mention the method without using the parentheses.)

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

Perhaps we want to be more selective - build the inner list inside the loop:

```
outer_dict = {}                                         # new dict
for line in open('F-F.txt').read().splitlines()[4:-3]:
  inner_list = []                                        # a new, empty 'inner' list
  columns = line.split()                                 # split the line into a list of string values
  date = columns[0]                                      # the first value is the date
  inner_list.append(columns[1])                          # add the 1st numeric value after the date to the
  inner_list.append(columns[4])                          # add the 4th numeric value to the 'inner' list
  outer_dict[date] = inner_list                          # now assign this newly built 'inner' list as valu
```

Inside the loop we are creating a temporary, empty list; building it up; and then finally associating it with a key in the "outer" dictionary. The work inside the loop can be seen as "the life of an inner structure" - it is built and then added to the inner structure - and then the loop moves ahead one line in the file and does the work again with a new inner structure.

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'

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).

Usage: string

The "Usage:" string shows how a program should be used from the command line. It is normally displayed when the user attempts to use the program incorrectly, or upon request.

When the user puts in improper arguments (required args that are missing, invalid argument values, or unknown arguments) the program responds by printing a usage: message and exiting. It may also do this if the user asks for help with a **-h** argument to the program.

Usage strings can be in any format and style that is clear and straightforward to the user. Here are two exmaples:

This is a usage message that I improvised:

```
usage:
populate_dev_tables.py   read_link=mysql_fiba_dev:tables=table1,table2:rows=1000:days=30:omit_last_day=Fals
populate_dev_tables.py   tables=account_goals,member:days=1 tables=account,country_region_mapping,exception
populate_dev_tables.py   recipe=account_tables

"read_link" is optional, default "mysql_managed_fiba";
"rows" and "days" args are optional and exclusive - default is %s rows
rows can be 'all'
omit_last_day=True leaves yesterday date off
```

This is a sample usage message provided by the **argparse** module, as shown in the **argparse** docs:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

Again, the format is largely discretionary. If you are looking for guidance you might want to look closely at the **argparse** example above.

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: writing to STDERR

The STDERR output stream is similar to STDIN, but programs recognize it as containing error messages.

```
import sys

args = sys.argv[1:]

if not args:
    sys.stderr.write('must supply argument(s)')
    exit(1)
```

sys.stderr writes to the STDERR output stream, which signals that it contains error messages. The operating system or user can direct this stream to an error file or use it to display error messages.

exit(1) indicates that the program exited with error. The OS and other progams can use this value to choose a different behavior if the program did not exit() as expected or desired (i.e., in error).

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
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
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
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 task: read file size with os.path.getmtime()

os.path.getmtime() takes a filename and returns the time the file was modified as a float in *epoch seconds*

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

for item in os.listdir(mydirectory):
    item_path = os.path.join(mydirectory, item)
    item_size = os.path.getmtime(item_path)
    print "{}:  modified {} seconds since 1970".format(item_path, item_size)
```

epoch seconds are a whole number count of seconds since January 1, 1970. Using a whole number to mark the absolute time since the epoch means that we can easily compare dates. Python provides a number of ways to convert epoch seconds to a readable format, most notable **time.ctime()**

```
import time

epoch_seconds = 1447778067.0

print time.ctime(epoch_seconds)      # Tue Nov 17 11:34:27 2015
```

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:  [Errno 2] 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, WindowsError, 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:                 # in Windows, a WindowsError
        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 **main()** 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                    # False

print d1 > d2                    # True
```

"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 -- 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.

```
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)
  def speak(self):
    print '%s:  Bark!  Bark!' % (self.name)

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
  def speak(self):
    print '%s:  Meow!' % (self.name)

for a in (Dog('Rover'), Dog('Fido'), Cat('Fluffy'), Cat('Precious'), Dog('Rex'), Cat('Kittypie')):
  a.speak()

                  # Rover:  Bark!  Bark!
                  # Fido:  Bark!  Bark!
                  # Fluffy:  Meow!
                  # Precious:  Meow!
                  # Rex:  Bark!  Bark!
                  # Kittypie:  Meow!
```

Static Methods and Class Methods

A *class method* can be called through the instance or the class, and passes the **class** as the first argument. We use these methods to do class-wide work, such as counting instances or maintaining a table of variables available to all instances.

A *static method* can be called through the instance or the class, but knows nothing about either. In this way it is like a regular function -- it takes no implicit argument. We can think of these as 'helper' functions that just do some utility work and don't need to involve either class or instance.

```
class MyClass(object):

  def myfunc(self):
    print "myfunc:  arg is %s" % self

  @classmethod
  def myclassfunc(klass):      # we spell it differently because 'class' will confuse the interpreter
    print "myclassfunc:  arg is %s" % klass

  @staticmethod
  def mystaticfunc():
    print "mystaticfunc: (no arg)"

a = MyClass()

a.myfunc()             # myfunc:  arg is <__main__.MyClass instance at 0x6c210>

MyClass.myclassfunc()  # myclassfunc:  arg is __main__.MyClass
a.myclassfunc()        # [ same ]

a.mystaticfunc()       # mystaticfunc: (no arg)
```

Here is an example from *Learning Python*, which counts instances that are constructed:

```
class Spam:

  numInstances = 0

  def __init__(self):
    Spam.numInstances += 1

  @staticmethod
  def printNumInstances():
    print "instances created:  ", Spam.numInstances

s1 = Spam()
s2 = Spam()
s3 = Spam()

Spam.printNumInstances()        # instances created:  3
s3.printNumInstances()          # instances created:  3
```

Python Data Model

Python's Data Model: Overview

The Data Model specifies how objects, attributes, methods, etc. function and interact in the processing of data.

The Python Language Reference (https://docs.python.org/2/reference/index.html) provides a clear introduction to Python's lexical analyzer, data model, execution model, and various statement types.

This session covers the basics of Python's data model. Mastery of these concepts allows you to create objects that behave (i.e., using the same interface -- operators, looping, subscripting, etc.) like standard Python objects, as well as in becoming conversant on StackOverflow and other discussion sites.

Special / "Private" / "Magic" Attributes

All objects contain "private" attributes that may be methods that are indirectly called, or internal "meta" information for the object.

The **__dict__** attribute shows any attributes stored in the object.

```
>>> list.__dict__.keys()
['__getslice__', '__getattribute__', 'pop', 'remove', '__rmul__', '__lt__', '__sizeof__',
 '__init__', 'count', 'index', '__delslice__', '__new__', '__contains__', 'append',
 '__doc__', '__len__', '__mul__', 'sort', '__ne__', '__getitem__', 'insert',
 '__setitem__', '__add__', '__gt__', '__eq__', 'reverse', 'extend', '__delitem__',
 '__reversed__', '__imul__', '__setslice__', '__iter__', '__iadd__', '__le__', '__repr__',
 '__hash__', '__ge__']
```

The **dir()** function will show the object's available attributes, including those available through inheritance.

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

In this case, dir(list) includes attributes not found in list.__dict__. What class(es) does list inherit from? We can use **__bases__** to see:

```
>>> list.__bases__
(object,)
```

This is a tuple of classes from which **list** inherits - in this case, just the super object **object**.

```
>>> object.__dict__.keys()
['__setattr__', '__reduce_ex__', '__new__', '__reduce__', '__str__', '__format__',
 '__getattribute__', '__class__', '__delattr__', '__subclasshook__', '__repr__',
 '__hash__', '__sizeof__', '__doc__', '__init__']
```

Of course this means that any object that inherits from **object** will have the above attributes. Most if not all built-in objects inherit from **object**. (In Python 3, all classes inherit from **object**.)

(Note that of course the term "private" in this context does not refer to unreachable data as would be used in C++ or Java.)

Object Inspection And Modification Built-in Functions

Object Inspection

isinstance()	Checks to see if this object is an instance of a class (or parent class)
issubclass()	Checks to see if this class is a subclass of another
callable()	Checks to see if this object is callable
hasattr()	Checks to see if this object has an attribute of this name

Object Attribute Modification

setattr()	sets an attribute in an object (using a string name)
getattr()	retrieves an attribute from an object (using a string name)
delattr()	deletes an attribute from an object (using a string name)

Special Attributes: "operator overloading"

Some special attributes are methods, usually called implictly as the result of function calls, the use of operators, subscripting or slicing, etc.

We can replace any operator and many functions with the corresponding "magic" methods to achieve the same result:

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

print var + var2         # helloworld
print var.__add__(var2)  # helloworld

print len(var)           # 5
print var.__len__()      # 5

if 'll' in var:
    print 'yes'

if var.__contains__('ll'):
    print 'yes'
```

Here is an example of a new class, **Number**, that reproduces the behavior of a number in that you can add, subtract, multiply, divide them with other numbers.

```
class Number(object):
  def __init__(self, start):
    self.data = start
  def __sub__(self, other):
    return Number(self.data - other)
  def __add__(self, other):
    return Number(self.data + other)
  def __mul__(self, other):
    return Number(self.data * other)
  def __div__(self, other):
    return Number(self.data / float(other))
  def __repr__(self):
    print "Number value: ",
    return str(self.data)

X = Number(5)
X = X - 2
print X              # Number value: 3
```

Of course this means that existing built-in objects make use of these methods -- you can find them listed from the object's **dir()** listing.

Special Attributes: Reimplementing __repr__ and __str__

__**str**__ is invoked when we print an object or convert it with **str()**; __**repr**__ is used when __**str**__ is not available, or when we view an object at the Python interpreter prompt.

```
class Number(object):
  def __init__(self, start):
    self.data = start
  def __str__(self):
    return str(self.data)
  def __repr__(self):
    return 'Number(%s)' % self.data

X = Number(5)
print X          # 5  (uses __str__ -- without repr or str, would be <__main__.Y object at 0x105d61190>
```

__**str**__ is intended to display a human-readable version of the object; __**repr**__ is supposed to show a more "machine-faithful" representation.

Special attributes available in class design

Here is a short listing of attributes available in many of our standard objects.

You view see many of these methods as part of the attribute dictionary through **dir()**.

There is also a more exhaustive exhaustive list (http://www.rafekettler.com/magicmethods.html#reflection) with explanations provided by Rafe Kettler.

object construction and destruction:

__init__	object constructor
__del__	del x (invoked when reference count goes to 0)
__new__	special 'metaclass' constructor

object rendering:

__repr__	"under the hood" representation of object (in Python interpreter)
__str__	string representation (i.e., when printed or with str()

object comparisons:

__lt__	<
__le__	<=
__eq__	==
__ne__	!=
__gt__	>
__ge__	>=
__nonzero__	(bool(), i.e. when used in a boolean test)

calling object as a function:

__call__	when object is "called" (i.e., with **()**)

container operations:

__len__	handles len() function
__getitem__	subscript access (i.e. mylist[0] or mydict['mykey'])

__missing__	handles missing keys
__setitem__	handles dict[key] = value
__delitem__	handles del dict[key]
__iter__	handles looping
__reversed__	handles reverse() function
__contains__	handles 'in' operator
__getslice__	handles slice access
__setslice__	handles slice assignment
__delslice__	handles slice deletion

attribute access (discussed in upcoming session):

__getattr__	object.attr read: attribute may not exist
__getattribute__	object.attr read: attribute that already exists
__setattr__	object.attr write
__delattr__	object.attr deletion (i.e., del this.that)

'descriptor' class methods (discussed in upcoming session)

__get__	when an attribute w/descriptor is read
__set__	when an attribute w/descriptor is written
__delete__	when an attribute w/descriptor is deleted with **del**

numeric types:

__add__	addition with +
__sub__	subtraction with -
__mul__	multiplication with *
__div__	division with \/
__floordiv__	"floor division", i.e. with //
__mod__	modulus

"Introspection" Special Attributes

The name, module, file, arguments, documentation, and other "meta" information for an object can be found in special attributes.

Below is a partial listing of special attributes; available attributes are discussed in more detail on the data model documentation page (https://docs.python.org/2/reference/datamodel.html).

user-defined functions

__doc__	doc string
__name__	this function's name
__module__	module in which this func is defined
__defaults__	default arguments

__code__	the "compiled function body" of bytecode of this function. Code objects can be inspected with the inspect (https://docs.python.org/2.7/library/inspect.html) module and "disassembled" with the dis (https://docs.python.org/2.7/library/dis.html) module.
__globals__	global variables available from this function
__dict__	attributes set in this function object by the user

user-defined methods

im_class	class for this method
__self__	instance object
__module__	name of the module

modules

__dict__	globals in this module
__name__	name of this module
__doc__	docstring
__file__	file this module is defined in

classes

__name__	class name
__module__	module defined in
__bases__	classes this class inherits from
__doc__	docstring

class instances (objects)

im_class	class
im_self	this instance

Variable Naming Conventions

Underscores are used to designate variables as "private" or "special".

lower-case separated by underscores	**my_nice_var**	"public", intended to be exposed to users of the module and/or class
underscore before the name	**_my_private_var**	"non-public", *not* intended for importers to access (additionally, "from modulename import *" doesn't import these names)
double-underscore before the name	**__dont_inherit**	"private"; its name is "mangled", available only as _classname__dont_inherit
double-underscores before and after the name	**__magic_me__**	"magic" attribute or method, specific to Python's internal workings
single underscore after the name	**file_**	used to avoid overwriting built-in names (such as the **file()** function)

```
class GetSet(object):

    instance_count = 0

    __mangled_name = 'no privacy!'

    def __init__(self,value):
        self._attrval = value
        instance_count += 1

    def getvar(self):
        print('getting the "var" attribute')
        return self._attrval

    def setvar(self, value):
        print('setting the "var" attribute')
        self._attrval = value

cc = GetSet(5)
cc.var = 10
print cc.var
print cc.instance_count

print cc._attrval                # "private", but available:  10
print cc.__mangled_name          # "private", apparently not available...
print cc._GetSet__mangled_name   # ...and yet, accessible through "mangled" name

cc.__newmagic__ = 10             # MAGICS ARE RESERVED BY PYTHON -- DON'T DO THIS
```

Subclassing Builtin Objects

Inheriting from a class (the *base or parent class*) makes all methods and attributes available to the inheriting class (the *child class*).

```
class NewList(list):     # an empty class - does nothing but inherit from list
    pass

x = NewList([1, 2, 3, 'a', 'b'])
x.append('HEEYY')

print x[0]   # 1
print x[-1]  # 'HEEYY'
```

Overriding Base Class Methods

This class automatically returns a default value if a key can't be found -- it traps and works around the **KeyError** that would normally result.

```
class DefaultDict(dict):

    def __init__(self, default=None):
        dict.__init__(self)
        self.default = default

    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.default
    def get(self, key, userdefault):
        if not userdefault:
            userdefault = self.default
        return dict.get(self, key, userdefault)

xx = DefaultDict()

xx['c'] = 5

print xx['c']          # 5
print xx['a']          # None
```

Since the other **dict** methods related to dict operations (__**setitem**__, extend(), keys(), etc.) are present in the **dict** class, any calls to them also work because of inheritance.

WARNING! Avoiding method recursion

Note the bolded statements in **DefaultDict** above (as well as **MyList** below) -- are calling methods *in the parent* in order to avoid *infinite recursion*. If we were to call **DefaultDict.get()** from inside **DefaultDict.__getitem__()**, Python would again call **DefaultDict.__getitem__()** in response, and an infinite loop of calls would result. We call this *infinite recursion*

The same is true for **MyList.__getitem__()** and **MyList.__setitem__()** below.

```
    # from DefaultDict.__getitem__()
    dict.get(self, key, userdefault)       # why not self.get(key, userdefault)?

    # from MyList.__getitem__()
    return list.__getitem__(self, index)   # why not self[index]?

    # from MyList.__setitem__()                   # (from example below)
    list.__setitem__(self, index, value)   # why not self[index] = value?
```

Another example -- a custom list that indexes items starting at **0**:

```
class MyList(list):         # inherit from list
  def __getitem__(self, index):
    if index == 0:  raise IndexError
    if index > 0: index = index - 1
    return list.__getitem__(self, index)  # this method is called when we access
                                          # a value with subscript (x[1], etc.)
  def __setitem__(self, index, value):
    if index == 0:  raise IndexError
    if index > 0: index = index - 1
    list.__setitem__(self, index, value)

x = MyList(['a', 'b', 'c'])  # __init__() inherited from builtin list

print x                      # __repr__() inherited from builtin list

x.append('spam');            # append() inherited from builtin list

print x[1]                   # 'a' (MyList.__getitem__
                             #      customizes list superclass method)
                             # index should be 0 but it is 1!

print x[4]                   # 'spam' (index should be 3 but it is 4!)
```

So **MyList** acts like a list in most respects, but its index starts at 0 instead of 1 (at least where subscripting is concerned -- other list methods would have to be overridden to complete this 1-indexing behavior).

Iterator Protocol

The protocol specifies methods to be implemented to make our objects *iterable*.

"Iterable" simply means able to be looped over or otherwise treated as a sequence or collection. The **for** loop is the most obvious feature that iterates, however a great number of functions and other features perform iteration, including list comprehensions, max(), min(), sorted(), map(), filter(), etc., because each of these must consider every item in the collection.

We can make our own objects iterable by implementing **__iter__** and **next**, and by raising the **StopIteration** exception

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def next(self):                  # Python 3: def __next__(self)
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1


for c in Counter(3, 8):
    print c
```

Reading a file with 'with'

```
with open('myfile.txt') as fh:
    for line in fh:
        print line

## at this point (outside the with block), filehandle fh has been closed.
```

Implementing a 'with' context

```
class LinkResolve():
    """ this class implements __exit__() so is used with python 'with'.
        Exit of the 'with' block results in one of two behaviors:
        - with a string argument, creates a 'link' db connection
          and closes the connection at 'with' block exit
        - with a 'link' db connection argument, returns the
          link db connection and does not close it.  """

    def __init__(self, obj):
        if isinstance(obj, link.wrappers.dbwrappers.DBConnectionWrapper):
             self.link = obj
        elif isinstance(obj, basestring):
            self.link = lnk('dbs.' + obj)
        else:
            raise ValueError('dutils.LinkResolve():  LinkResolve() requires a '
                             'single string or link db connection argument')

    def __enter__(self):
        return self

    def __exit__(self, type, value, traceback):
        if not keepalive_all and not self.keepalive and not self.link.wrap_name.startswith('salesforce'):
            self.link.close()
```

Internal Types

Some implicit objects can provide information on code execution.

Traceback objects

Traceback objects become available during an exception. Here's an example of inspection of the exception type using **sys.exc_info()**

```
import sys, traceback
try:
    some_code_i_wrote()
except BaseException, e:
    error_type, error_string, error_tb =  sys.exc_info()
    if not error_type == SystemExit:
        print 'error type:    {}'.format(error_type)
        print 'error string:  {}'.format(error_string)
        print 'traceback:     {}'.format(''.join(traceback.format_exception(error_type, e, error_tb)))
```

Code objects

In CPython (the most common distribution), a code object is a piece of compiled bytecode. It is possible to query this object / examine its attributes in order to learn about bytecode execution. A detailed exploration of code objects can be found here (https://late.am/post/2012/03/26/exploring-python-code-objects.html).

Frame objects

A frame object represents an execution frame (a new frame is entered each time a function is called). They can be found in traceback objects (which trace frames during execution).

f_back	previous stack frame
f_code	code object executed in this frame
f_locals	local variable dictionary
f_globals	global variable dictionary
f_builtins	built-in variable dictionary

For example, this line placed within a function prints the function name, which can be useful for debugging -- here we're pulling a frame, grabbing the code object of that frame, and reading the attribute **co_name** to read it.

```
import sys

def myfunc():
    print 'entering {}()'.format(sys._getframe().f_code.co_name )
```

Calling this function, the **frame** object's function name is printed:

```
myfunc()        # entering myfunc()
```

Attribute Access in Classes and Instances

getattr() and setattr()

These built-in functions allow attribute access through string arguments.

```
class This(object):       # a simple class with one class variable
    a = 5

x = This()

print getattr(x, 'a')     # 5:  finds the 'a' attribute in the class

setattr(x, 'b', 10)       #     set x.b = 10 in the instance

print x.b                 # 10: retrieve x.b from the instance
```

Similar to the dict method **get()**, a default argument can be passed to **getattr()** to return a value if the attribute doesn't exist (otherwise, a missing attribute will raise an **AttributeError** exception).

```
class InstMake(object):       # create a featureless class so we can
                              # play with its instance
    pass

x = InstMake()

curr_val = getattr(x, 'intval', 0)   # no 'intval' attribute, so default 0
setattr(x, 'intval', 10)             # set 'intval' to 10

print x.__dict__                     # {'intval': 10}
```

We might want to use these functions as a dispatch utility: if our program is working with a string value that is the name of an attribute, we can use the string directly.

```
var = 'hElLo'
for methodname in ('upper', 'lower', 'title'):
    print getattr(var, methodname)()            # call each method through the attribute
                                                # HELLO
                                                # hello
                                                # Hello
```

Special methods __getattr__, __getattribute__ and __setattr__

These special methods are called when we access an attribute (setting or getting). We can implement them for broad control over our custom class' attributes.

```
class MyClass(object):

    classval = 5

    # when read attribute does not exist
    def __getattr__(self, name):
        default = 0
        print 'getattr: "{}" not found; setting default'.format(name)
        setattr(self, name, default)
        return default

    # when attribute is read
    def __getattribute__(self, name):
        print 'getattribute:  attempting to access "{}"'.format(name)
        return object.__getattribute__(self, name)

    # when attribute is assigned
    def __setattr__(self, name, value):
        print 'setattr:  setting "{}" to value "{}"'.format(name, value)
        self.__dict__[name] = value

x = MyClass()

x = MyClass()

x.a = 5             # setattr:  setting "a" to value "5"

print x.a           # getattribute:  attempting to access "__dict__"
                    # getattribute:  attempting to access "a"
                    # 5

print x.ccc         # getattribute:  attempting to access "ccc"
                    # getattr: "ccc" not found; setting default
                    # setattr:  setting "ccc" to value "0"
                    # getattribute:  attempting to access "__dict__"
                    # 0
```

__getattribute__: implicit call upon attribute read. Anytime we attempt to access an attribute, Python calls this method if it is implemented in the class.

__getattr__: implicit call for non-existent attribute. If an attribute does not exist, Python calls this method -- regardless of whether it called **__getattribute__**.

recursion alert: we must use alternate means of getting or setting attributes lest Python call these methods repeatedly:
__getattribute__(): use **object.__getattribute__(self, name)**
__setattr__(): use **self.__dict__[name] = value**

e.g., use of **self.attr = val** in **__setattr__** would cause the method to call itself.

@property: attribute control

This *decorator* allows behavior control when an individual attribute is accessed, through separate @property, @setter and @deleter methods.

```
class GetSet(object):

    def __init__(self,value):
        self.attrval = value

    @property
    def var(self):
        print 'thanks for calling me -- returning {}'.format(self.attrval)
        return self.attrval

    @var.setter
    def var(self, value):
        print "thanks for setting me -- setting 'var' to {}".format(value)
        self.attrval = value

    @var.deleter
    def var(self):
        print 'should I thank you for deleting me?'
        self.attrval = None

me = GetSet(5)

me.var = 1000    # setting the "var" attribute

print me.var     # thanks for calling me -- returning 1000
                 # 1000

del me.var       # should I thank you for deleting me?
print me.var     # thanks for calling me -- returning None
                 # None
```

Note that each decorated method is called **def var**. This would cause conflicts if it weren't for the decorators.

One caveat: since the interface for attribute access appears very simple, it can be misleading to attach computationally expensive operations to an attribute decorated with @property.

Implementing Decorators

The ability of a function to accept other functions as arguments, or to return functions as return values, are at the heart of the *@decorator* scheme.

Python **decorators** are functions that modify the behavior of other functions. They can be added to any function through the use of the **@** sign and the decorator name on the line above the function.

Here's a simple example adapted from the Jeff Knupp blog (https://jeffknupp.com/blog/2013/11/29/improve-your-python-decorators-explained/):

```
def currency(f):                              # decorator function
    def wrapper(*args, **kwargs):
        return '$' + str(f(*args, **kwargs))
    return wrapper

@currency
def price_with_tax(price, tax_rate_percentage):
    """Return the price with *tax_rate_percentage* applied.
    *tax_rate_percentage* is the tax rate expressed as a float, like
    "7.0" for a 7% tax rate."""

    return price * (1 + (tax_rate_percentage * .01))

print price_with_tax(50, .10)           # $50.05

# 'manual' version of the above
pwt = currency(price_with_tax)
print pwt(50, .10)                      # $$50.05
```

In this example, ***args** and ****kwargs** represent "unlimited positional arguments" and "unlimited keyword arguments". This is done to allow the flexibility to decorate any function (as it will match any function argument signature).

You may have reflected that **@classmethod** and **@staticmethod** are decorators. This means that there is a built-in object called **classmethod** and one called **staticmethod** and these are responsible for handling the arguments to any decorated function -- principally to allow calls that don't require an instance as first argument, which is the default behavior when methods are called on objects.

```
class DoThis(object):

    def show_instance(self):   # default 'instance' method:  instance passed
        print self             # <__main__.DoThis object at 0x1004df590>

    @classmethod
    def show_class(cls):       # decorated 'class' method:  class is passed
        print cls              # <class '__main__.DoThis'>

    @staticmethod
    def say_hello():           # decorated 'static' method:  no implicit argument
        print 'hello!'         # hello!

x = DoThis()
x.printme()                    # standard instance method:  object is impilcitly passed
x.show_class()                 # class method:  class is implicitly passed
x.say_hello()                  # static method:  no arg implicitly passed
```

The benefit here is that, rather than requiring the user to explicitly pass a function to a processing function, we can simply decorate each function to be processed and it will behave as advertised.

Attribute access: descriptors

A descriptor is an attribute that is linked to a separate class that defines **__get__()**, **__set__()** or **__delete__()**.

```
class RevealAccess(object):
    """ A data descriptor that sets and returns values and prints a message declaring access. """

    def __init__(self, initval=None):
        self.val = initval

    def __get__(self, obj, objtype):
        print 'Getting attribute from object', obj
        print '...and doing some related operation that should take place at this time'
        return self.val

    def __set__(self, obj, val):
        print 'Setting attribute from object', obj
        print '...and doing some related operation that should take place at this time'
        self.val = val


# the class we will work with directly
class MyClass(object):
    """ A simple class with a class variable as descriptor """
    def __init__(self):
        print 'initializing object ', self

    x = RevealAccess(initval=0)  # attach a descriptor to class attribute 'x'


mm = MyClass()                   # initializing object  <__main__.MyClass object at 0x10066f7d0>

mm.x = 5                         # Setting attribute from object <__main__.MyClass object at 0x1004de910>
                                 # ...and doing some related operation that should take place at this time

val = mm.x                       # Getting attribute from object <__main__.MyClass object at 0x1004de910>
                                 # ...and doing some related operation that should take place at this time

print 'retrieved value: ', val   # retrieved value:  5
```

You may observe that descriptors behave very much like the **@property** decorator. And it's no coincidence: **@property** is implemented using descriptors.

__slots__

This class variable causes object attributes to be stored in a specially designated space rather than in a dictionary (as is customary).

```
class MyClass(object):
    __slots__ = ['var', 'var2', 'var3']

a = MyClass()

a.var = 5
a.var2 = 10
a.var3 = 20
a.var4 = 40   # AttributeError:  'MyClass' object has no attribute 'var4'
```

All objects store attributes in a designated dictionary under the attribute __dict__. This takes up a fairly large amount of memory space for each object created.

__**slots**__, initialized as a list and as a class variable, causes Python *not* to create an object dictionary; instead, just enough memory needed for the attributes is allocated.

When many instances are being created a marked improvement in performance is possible. The hotel rating website **Oyster.com** reported a problem solution (http://tech.oyster.com/save-ram-with-python-slots/) in which they reduced their memory consumption by a third by using **__slots__**.

Please note however that slots should not be used to limit the creation of attributes. This kind of control is considered "un-pythonic" in the sense that privacy and control are mostly cooperative schemes -- a user of your code should understand the interface and not attempt to subvert it by establishing unexpected attributes in an object.

Functional Programming

Functional Programming: Overview

We may say that there are three commonly used styles (sometimes called *paradigms*) of program design: *imperative/procedural*, *object-oriented* and ***functional***.

Here we'll tackle a common problem (summing a sequence of integers, or an *arithmetic series*) using each style:

imperative or **procedural** involves a series of statements along with variables that change as a result. We call these variable values the program's *state*.

```
mysum = 0
for counter in range(11):
    mysum = mysum + counter

print mysum
```

object-oriented uses *object state* to produce outcome.

```
class Summer(object):
    def __init__(self):
        self.sum = 0
    def add(self, num):
        self.sum = self.sum + num

s = Summer()
for num in range(11):
    s.add(num)

print s.sum
```

functional combines *pure functions* to produce outcome. No *state change* is involved.

```
print sum(range(11))
```

A pure function is one that only handles input, output and its own variables -- it does not affect nor is it affected by global or other variables existing outside the function.

Because of this "air-tightness", functional programming can be tested more reliably than the other styles.

Some languages are designed around a single style or paradigm. But since Python is a "multi-paradigm" language, it is possible to use it to code in any of these styles.

To employ functional programming in our own programs, we need only seek to replace imperative code with functional code, combining pure functions in ways that replicate some of the patterns that we use to iterate, summarize, compute, etc.

After some practice coding in this style, you may recognize patterns for iteration, accumulation, etc. and more readily employ them in your programs, making them more predictable, testable and less prone to error.

Note: Python documentation provides a solid overview (https://docs.python.org/2/howto/functional.html) of functional programming in Python.

Mary Rose Cook provides a plain language introduction (https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming) to functional programming.

O'Reilly publishes a free e-book (http://www.oreilly.com/programming/free/functional-programming-python.csp) with a comprehensive review of functional programming by Python luminary David Mertz.

Review: lambdas

Lambda functions are simply inline functions -- they can be defined entirely within a single statement, within a container initialization, etc.

Lambdas are most often used inside functions like **sorted()**:

```
# sort a list of names by last name
names = [ 'Josh Peschko', 'Gabriel Feghali', 'Billy Woods', 'Arthur Fischer-Zernin' ]
sortednames = sorted(names, key=lambda name:  name.split()[1])

# sort a list of CSV lines by the 2nd column in the file
slines = sorted(lines, lambda x: x.split(',')[2])
```

We will see lambdas used in functions we'll use for functional programming in this section, such as map(), filter() and reduce() (in Python 3, **reduce()** is part of the **functools** module).

Review: list comprehensions; set comprehensions and dict comprehensions

List, set and dict comprehensions can filter or transform sequences in a single statement.

Functional programming (and algorithms in general) often involves the processing of sequences. List comprehensions provide a flexible way to filter and modify values within a list.

list comprehension: return a list

```
nums = [1, 2, 3, 4, 5]
dblnums = [ val * 2 for val in nums ]
print dblnums                                  # [2, 4, 6, 8, 10]

print [ val * 2 for val in nums if val > 2]    # [6, 8, 10]
```

set comprehension: return a set

```
states = { line.split(':')[3]
           for line in open('student_db.txt').readlines()[1:] }
```

dict comprehension: return a dict

```
student_states = { line.split(':')[0]: line.split(':')[3]
                   for line in open('student_db.txt').readlines()[1:] }
```

map() and filter() as alternatives to list comprehensions

Although list comprehensions have nominally replaced **map()** and **filter()**, these functions are still used in many functional programming algorithms.

map(): apply a transformation function to each item in a sequence

```
# square some integers
sqrd = map(lambda x: x ** 2, range(6))    # [1, 4, 9, 16, 25]

# get string lengths
lens = map(len, ['some', 'words', 'to', 'get', 'lengths', 'from'])
print lens    # [4, 5, 2, 3, 7, 4]
```

filter(): apply a filtering function to each item in a sequence

```
pos = filter(lambda x: x > 0, [-5, 2, -3, 17, 6, 4, -9])
print pos     # [2, 17, 6, 4]
```

reduce() for accumulation of values

Like map() or filter(), **reduce()** applies a function to each item in a sequence, but *accumulates* a value as it iterates.

Note that in Python 3, **reduce()** must be imported from the **functools** module:

```
from functools import reduce
```

reduce accumulates values through a second variable to its processing function. In the below examples, the accumulator is **a** and the current value of the iteration is **x**. **a** grows through the accumulation, as if the function were saying **a = a + x** or **a = a * x**.

Here is our arithmetic series for integers 1-10, done with reduce():

```
from functools import reduce     # only needed for python 3

def addthem(a, x):
    return a + x

intsum = reduce(addthem, range(1, 11))

# same using a lambda
intsum = reduce(lambda a, x: a + x, range(1, 11))
```

Just as easily, a factorial of integers 1-10:

```
facto = reduce(lambda a, x: a * x, range(1, 11))       # 3628800
```

default value

Since **reduce()** has to start with a value in the accumulator, it will attempt to begin with the first element in the source list. However, if each value is being transformed before being accumulated, the first computation may result in an error:

```
from functools import reduce     # only needed for python 3

strsum = reduce(lambda a, x: a + int(x), ['1', '2', '3', '4', '5'])

TypeError: cannot concatenate 'str' and 'int' objects
```

This is apparently happening because python is trying to add **int('1')** to **''** (i.e., **reduce()** does not see the transform and so uses an 'empty' version of the type, in this case an empty string).

In these cases we can supply an initial value to **reduce()**, so it knows where to begin:

```
from functools import reduce     # only needed for python 3

strsum = reduce(lambda a, x: a + int(x), ['1', '2', '3', '4', '5'], 0)
```

Higher-Order Functions
Any function that takes a function as an argument, or that returns a function as a return value, is a *higher-order function*.

map(), **filter()**, **reduce()**, **sorted** all take functions as arguments.

@properties, **@staticmethod**, **@classmethod** all take a given function as argument and return a modified function as a return value.

Decorators are functions that "intercept" function's arguments and return a function to be called in its place with additional functionality.

any() and all(): return True based on truth of elements

Also iterating functions, **any()** and **all()** will return **True** if any or all of the items passed to it are **True**.

any(): return True if any elements are True

```
any([1, 0, 2, 0, 3])        # True:  at least one item is True

any([0, [], {}, ''])        # False: none of the items is True
```

all(): return True if all elements are True

```
all([1, 5, 0.0001, 1000])   # True:  all items are True

all([1, 5, 9, 10, 0, 20])   # False:  one item is not True
```

ternary (if/then/else) assignment; conditional assignment

These statements allow for an if/else that can be made part of a statement

ternary assignment

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

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

Return the first item if the test is **True**; otherwise return the second item.

conditional assignment

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

Return the first item if the item is **True**; otherwise return the second item.

iterators

an iterator is an object that supports the *iterator protocol* (__**iter**__ and **next()**)

The Range class simulates the xrange() function:

```
class Range(object):

    def __init__(self, start, stop):
        if not isinstance(start, int) or not isinstance(stop, int):
            raise ValueError('start and stop indices must be ints')
        self.start = start - 1
        self.stop = stop

    def __iter__(self):
        return self

    def next(self):
        self.start = self.start + 1
        if self.start == self.stop:
            raise StopIteration
        return self.start

for i in Range(0, 10):
    print i
```

Any object that properly impelements __**iter**__ and **next()** is said to be an *iterator*. This means that it can be placed in a **for** loop or function that expects an iterator (some we have seen are **sum()**, **list()**, **sorted()**, etc.

The __**iter**__ method is expected to return **self**.

When he **next()** method is called, it is expected to return the next value in the sequence; if it "reaches the end" of its sequence (this is determined entirely by the method, not by any existing condition), it is expected to raise a **StopIteration** exception. We do not see this exception, it is instead trapped by the **for** loop or function, at which point it stops looping.

generators

Generators are iterators that can calculate and generate any number of items.

Generators behave like iterators, except that they **yield** a value rather than **return** one, and *they remember the value of their variables* so that the next time the class' **next()** method is called, it picks up where the last left off (at the point of *yield*). .

As a generator is an iterator, **next()** calls the function again to produce the next item; and **StopIteration** causes the generator to stop iterating. (**next()** is called automatically by iterators like **for**.

Generators are particularly useful in producing a sequence of **n** values, i.e. not a fixed sequence, but an unlimited sequence. In this example we have prepared a generator that generates primes up to the specified limit.

```
def get_primes(num_max):
    """ prime number generator """
    candidate = 2
    found = []
    while True:
        if all(candidate % prime != 0 for prime in found):
            yield candidate
            found.append(candidate)
        candidate += 1
        if candidate >= num_max:
            raise StopIteration

my_iter = get_primes(100)
print my_iter.next()        # 2
print my_iter.next()        # 3
print my_iter.next()        # 5

for i in get_primes(100):
    print i
```

Generator comprehensions

These are list comprehensions that are used as iterators, computing values only as necessary -- usually, in the context of iteration (e.g. looping).

```
x = ( a for a in [1, 2, 3] )   # note the parentheses
print x                        # <generator object <genexpr> at 0x101934eb8>

for item in x:
    print item
```

We may prefer a generator if our comprehensions is expected to produce a very large dataset, or even an infinite one.

recursive functions

A recursive function calls itself until a condition has been reached.

Recursive functions are appropriate for processes that iterate over an unknown number of items or events. Such situations could include files within a directory tree, where *listing the directory* is performed over and over until all directories within a tree are exhausted; or similarly, visiting links to pages within a website, where *listing the links in a page* is performed repeatedly.

Recursion features three elements: a *recursive call*, which is a call by the function to itself; the function process itself; and a *base condition*, which is the point at which the chain of recursions finally returns.

In this example, we are presented with a tree of companies organized into a hierarchy of parent and child companies. We are interested in finding the "top parent" of any given company -- so if **Acme Pens'** parent is **Acme Office Supply** and taht company's parent is **Megacorp**, then **Acme Pens'** "top parent" is **Megacorp**.

The dict below shows each company's parent. If the company has no parent, its value is **None**.

```
parents = {
            'Megacorp':           None,

            'Acme Office Supply': 'Megacorp',
            'Acme Paper':         'Acme Office Supply',
            'Acme Pens':          'Acme Office Supply',

            'Best Electronics':   None,
            'Best Audio':         'Best Electronics',
            'Best TV':            'Best Electronics',

            'Celera Publishing':  'Megacorp',
            'Celera Books':       'Celera Publishing',
            'Celera Web':         'Celera Publishing'
                                                  }

def top_parent(company_id):
    if ( company_id not in parents or       # base case (no parent):  return
         not parents[company_id] ):

        return company_id

    return top_parent(parents[company_id])  # recursive call with parent id


for id in sorted(parents.keys()):
    print '{}:  {}'.format(id, top_parent(id))
```

The above outputs:

```
Acme Office Supply:  Megacorp
Acme Paper:          Megacorp
Acme Pens:           Megacorp
Best Audio:          Best Electronics
Best Electronics:    Best Electronics
Best TV:             Best Electronics
Celera Books:        Megacorp
Celera Publishing:   Megacorp
Celera Web:          Megacorp
Megacorp:            Megacorp
```

Here's a solution to the factorial problem using recursion:

```
def factorial(n):
   if n < 1:                            # base case (reached 0):  returns
       return 1
   else:
       return_num = n * factorial(n - 1)  # recursive call
       return return_num

print factorial(5)       # 120
```

Testing

Testing: Introduction

Code without tests is like driving without seatbelts.

All code is subject to errors -- not the SyntaxErrors and TypeErrors encountered during development, but errors related to unexpected data anomalies or user input, or the unforeseen effects of functions run in untested combinations.

Many developers say they won't take a software package seriously unless it comes with tests. **Unit testing** is the front line of the effort to ensure code quality.

testing: a brief rundown
Unit testing is the most basic form and the one we will focus on in this unit. Other styles of testing:

unit test: testing individual units (function/method)
integration test: testing multiple units together
regression test: testing to see if changes have introduced errors
end-to-end test: testing the entire program

Unit Testing

"Unit" refers to a function. Unit testing calls individual functions and validates the output or result of each.

The most easily tested scripts are made up of small functions that can be called and validated in isolation. Therefore "pure functions" (functions that do not refer or change "external state" -- i.e., global variables) are best for testing.

Testing for success, testing for failure
A **unit test script** performs test by importing the script to be tested and calling its functions with varying arguments, including ones intended to cause an error. Basically, we are hammering the code as many ways as we can to make sure it succeeds properly *and fails properly*.

Test-driven development
As we develop our code, we can write tests simultaneously and run them periodically as we develop. This way we can know that further changes and additions are not interfering with anything we have done previously. Any time in the process we can run the testing program and it will run all tests.

In fact commonly accepted wisdom supports writing tests before writing code! The test is written with the function in mind: after seeing that the tests fail, we write a function to satisfy the tests. This called *test-driven development*.

The *assert* statement

assert raises an **AssertionError** exception if the test returns **False**

```
assert 5 == 5        # no output

assert 5 == 10       # AssertionError raised
```

We can incorporate this facility in a simple testing program:

program to be tested: "myprogram.py"

```
import sys

def doubleit(x):
    var = x * 2
    return var

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

testing program: "test_myprogram.py"

```
import myprogram

def test_doubleit_value():
    assert myprogram.doubleit(10) == 20
```

If **doubleit()** didn't correctly return **20** with an argument of **10**, the **assert** would raise an **AssertionError**. So even with this basic approach (without a testing module like **pyttest** or **unittest**), we can do testing with **assert**.

pytest Basics

All programs named **test_*something*.py** that have functions named **test_*something*()** will be noticed by **pytest** and run automatically when we run the pytest script **py.test**.

running py.test from the command line

```
$ py.test
 ================================== test session starts ===================================
platform darwin -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/dblaikie/testpytest, inifile:
collected 1 items

test_myprogram.py .

 =================================== 1 passed in 0.01 seconds ====================================
```

noticing failures

```
def doubleit(x):
    var = x * 2
    return x      # oops, returned the original value rather than the doubled value
```

Having incorporated an error, run **py.test** again:

```
$ py.test
 ================================= test session starts ==================================
platform darwin -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/dblaikie/testpytest, inifile:
collected 1 items

test_myprogram.py F

 ======================================= FAILURES =======================================
_____________________________________ test_doubleit_value _____________________________________

    def test_doubleit_value():
>       assert myprogram.doubleit(10) == 20
E       assert 10 == 20
E        +  where 10 = (10)
E        +    where  = myprogram.doubleit

test_myprogram.py:7: AssertionError
 ================================ 1 failed in 0.01 seconds ================================
```

Testing the expected raising of an exception

Many of our tests will deliberately pass bad input and test to see that an appropriate exception is raised.

```
import sys

def doubleit(x):
    if not isinstance(x, (int, float)):           # make sure the arg is the right type
        raise TypeError, 'must be int or float'   # if not, raise a TypeError
    var = x * 2
    return var

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

Note that without type testing, the function could work, but incorrectly (for example if a string or list were passed instead of an integer).

To verify that this error condition is correctly raised, we can use **with pytest.raises(TypeError)**.

```
import myprogram
import pytest

def test_doubleit_value():
    assert myprogram.doubleit(10) == 20

def test_doubleit_type():
    with pytest.raises(TypeError):
        myprogram.doubleit('hello')
```

with is the same context manager we have used with **open()**: it can also be used to detect when an exception occured inside the **with** block.

Grouping tests into a class

We can organize related tests into a class, which can also include setup and teardown routines that are run automatically (discussed next).

```
""" test_myprogram.py -- test functions in a testing class """

import myprogram
import pytest

class TestDoubleit(object):

    def test_doubleit_value(self):
        assert myprogram.doubleit(10) == 20

    def test_doubleit_type(self):
        with pytest.raises(TypeError):
            myprogram.doubleit('hello')
```

So now the same rule applies for how py.test looks for tests -- if the class begins with the word **Test**, pytest will treat it as a testing class.

Mock data: setup and teardown

Tests should not be run on "live" data; instead, it should be simulated, or "mocked" to provide the data the test needs.

```
""" myprogram.py -- makework functions for the purposes of demonstrating testing """

import sys

def doubleit(x):
    """ double a number argument, return doubled value """
    if not isinstance(x, (int, float)):
        raise TypeError, 'arg to doublit() must be int or float'
    var = x * 2
    return var

def doublelines(filename):
    """open a file of numbers, double each line, write each line to a new file"""
    with open(filename) as fh:
        newlist = []
        for line in fh:                   # file is assumed to have one number on each line
            floatval = float(line)
            doubleval = doubleit(floatval)
            newlist.append(str(doubleval))
    with open(filename, 'w') as fh:
        fh.write('\n'.join(newlist))

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

For this demo I've invented a rather arbitrary example to combine an external file with the **doubleit()** routine: **doublelines()** opens and reads a file, and for each line in the file, doubles the value, writing each value as a separate line to a new file (supplied to **doublelines()**).

```
""" test_myprogram.py --

import myprogram
import os
import pytest
import shutil

class TestDoubleit(object):

    numbers_file_template = 'testnums_template.txt'  # template for test file (stays the same)
    numbers_file_testor = 'testnums.txt'             # filename used for testing
                                                     # (changed during testing)

    def setup_class(self):
        shutil.copy(TestDoubleit.numbers_file_template, TestDoubleit.numbers_file_testor)

    def teardown_class(self):
        os.remove(TestDoubleit.numbers_file_testor)

    def test_doublelines(self):
        myprogram.doublelines(TestDoubleit.numbers_file_testor)
        old_vals = [ float(line) for line in open(TestDoubleit.numbers_file_template) ]
        new_vals = [ float(line) for line in open(TestDoubleit.numbers_file_testor) ]
        for old_val, new_val in zip(old_vals, new_vals):
            assert float(new_val) == float(old_val) * 2

    def test_doubleit_value(self):
        assert myprogram.doubleit(10) == 20

    def test_doubleit_type(self):
        with pytest.raises(TypeError):
            myprogram.doubleit('hello')
```

setup_class and **teardown_class** run automatically. As you can see, they prepare a dummy file and when the testing is over, delete it. In between, tests are run in order based on the function names.

Profiling, Benchmarking and Efficiency

Efficiency: Introduction

Runtime Efficiency refers to two things: *memory* efficiency (whether a lot of RAM memory is being used up during a process) and *time* efficiency (how long execution takes). And these are often related -- it takes time to allocate memory.

As a "scripting" language, Python is more convenient, but less efficient than "programming" languages like C and Java:

- Parsing, compilation and execution take place in one step
- Memory is allocated based on anticipation of what your code will do at runtime
- Python handles expanded memory requests seamlessly -- "no visible limits"

Achieving runtime efficiency requires a more or less direct tradeoff with development speed -- it takes time to make our programs more efficient -- so we either spend more of our own (developer) time making our programs more efficient so they run faster, or we spend less time developing our programs and allow them to run slower. Of course just the choice of a convenient scripting language over a more efficient programming language means that rapid development, ease of use, etc. are more important to us than runtime efficiency, so in many applications efficiency is not a consideration because there's plenty of memory and time to get the job done. However, advanced Python developers may be asked to increase the efficiency of their programs, because the task at hand is simply taking longer than it should -- the data has grown past anticipated limits, the program's responsibilities and complexity has grown, or an unknown inefficiency is bogging down execution. In this section

we'll discuss the more efficient container structures and ways to analyze the speed of the various units in our programs. **Collections**: high performance container datatypes

- **array**: type-specific list
- **deque**: "double-ended queue"
- **Counter**: a counting dictionary
- **defaultdict**: a dict with automatic default for missing keys

timeit: unit timer to compare time efficiency of various Python algorithms **cProfile**: overall time profile of a Python program **"enhanced" Python packages**

Collections

The **collections** module provides more efficient containers.

array: a list of a uniform type
An array's type must be specified at initialization. A uniform type makes an array more efficient than a list, which can contain any type.

```
from array import array

myarray = array('i', [1, 2])

myarray.append(3)

print myarray           # array('i', [1, 2, 3])

print myarray[-1]       # acts like a list
for val in myarray:
    print val

myarray.append(1.3)     # error
```

(array is not part of the Collections module but is usefully discussed here)

Available **array** types:

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2 (see note)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

deque: "double-ended queue" for fast adds/removals **list**s are optimized for fixed-length operations, i.e., things like sorting, checking for membership, index access, etc. They are not optimized for appends, although this is of course a common use for them. A **deque** is designed specifically for fast adds -- to the beginning or end of the sequence:

```
from collections import deque

x = deque([1, 2, 3])

x.append(4)               # x now [1, 2, 3, 4]
x.appendleft(0)           # x now [0, 1, 2, 3, 4]

popped = x.pop()          # removes '4' from the end

popped2 = x.popleft()     # removes '1' from the start
```

A deque can also be sized, in which case appends will push existing elements off of the ends:

```
x = deque(['a', 'b', 'c'], 3)      # maximum size:  3
x.append(99)                       # now: deque(['b', 'c', 99])  ('a' was pushed off of the start)
x.appendleft(0)                    # now: deque([0, 'b', 'c'])   (99 was pushed off of the end)
```

Counter: a counting dictionary

This structure inherits from **dict** and is designed to allow an integer count as well as a default **0** value for new keys. So instead of doing this:

```
c = {}
if 'a' in c:
    c['a'] = 0
else:
    c['a'] + 1
```

We can do this:

```
from collections import Counter

c = Counter()
c['a'] = c['a'] + 1
```

Counter also has related functions that return a list of its keys repeated that many times, as well as a list of tuples ordered by frequency:

```
from collections import Counter

c = Counter({'a': 2, 'b': 1, 'c': 3, 'd': 1})

for key in c.elements():
    print key,            # c c c a a b b

print ','.join(c.elements())   # c,c,c,a,a,b,b


print c.most_common(2)   # [('c', 3), ('a', 2)]
                         # 2 arg says "give me the 2 most common"

c.clear()                # set all counts to 0 (but not remove the keys)
```

And, you can use **Counter**'s implementation of the math operators to work with multiple counters and have them sum their values:

```
c = Counter({'a': 1, 'b': 2})
d = Counter({'a': 10, 'b': 20})

print c + d                     # Counter({'b': 22, 'a': 11})
```

defaultdict: a default object for new missing keys

Similar to **Counter**, **defaultdict** allows for a default value if a key doesn't exist; but it will accept a function that provides a default value.

```
from collections import defaultdict

ddict = defaultdict(list)

ddict['a'].append(1)
ddict['b']

print ddict                    # defaultdict(, {'a': [1], 'b': []})
```

Benchmarking a Python function with timeit

The **timeit** module provides a simple way to time blocks of Python code.

We can thus decide whether varying approaches might make our programs more efficient. Here we compare execution time of four approaches to joining a range of integers into a very large string ("1-2-3-4-5...", etc.)

```
from timeit import timeit

# 'straight concatenation' approach
def joinem():
    x = '1'
    for num in range(100):
        x = x + '-' + str(num)
    return x

print timeit('joinem()', setup='from __main__ import joinem', number=10000)

# 0.457356929779


# generator comprehension
print timeit('"-".join(str(n) for n in range(100))', number=10000)

# 0.338698863983


# list comprehension
print timeit('"-".join([str(n) for n in range(100)])', number=10000)

# 0.323472976685


# map() function
print timeit('"-".join(map(str, range(100)))', number=10000)

# 0.160399913788
```

map() is fastest probably because it is compiled in **C**.

Repeating a test

You can conveniently repeat a test multiple times by calling a method on the object returned from **timeit()**. Repetitions give you a much better idea of the average time it might take.

```
from timeit import repeat

print repeat('"-".join(map(str, range(100)))', number=10000, repeat=3)

# [0.15206599235534668, 0.1909959316253662, 0.2175769805908203]


print repeat('"-".join([str(n) for n in range(100)])', number=10000, repeat=3)

# [0.35890698432922363, 0.327725887298584, 0.3285980224609375]


print repeat('"-".join(map(str, range(100)))', number=10000, repeat=3)

# [0.14228010177612305, 0.14016509056091309, 0.14458298683166504]
```

setup= parameter for setup before a test

Some tests make use of a variable that must be initialized before the test:

```
print timeit('x.append(5)', setup='x = []', number=10000)

# 0.00238704681396
```

Additionally, **timeit()** does not share the program's global namespace, so imports and even global variables must be imported if required by the test:

```
print timeit('x.append(5)', setup='import collections as cs; x = cs.deque()', number=10000)

# 0.00115013122559
```

Here we're testing a function, which as a global needs to be imported from the __**main**__ namespace:

```
def testme(maxlim):
    return [ x*2 for x in range(maxlim) ]

print timeit('testme(5000)', setup='from __main__ import testme', number=10000)

# 10.2637062073
```

Keep in mind that a function tested in isolation may not return the same results as a function using a different dataset, or a function that is run as part of a larger program (that has allocated memory differently at the point of the function's execution). The **cProfile** module can test overall program execution.

Profiling a Python program with cProfile

The profiler runs an entire script and times each unit (call to a function).

If a script is running slowly it can be difficult to identify the bottleneck. **timeit()** may not be adequate as it times functions in isolation, and not usually with "live" data.

This test program (**ptest.py**) deliberately pauses so that some functions run slower than others:

```
import time

def fast():
    print("I run fast!")


def slow():
    time.sleep(3)
    print("I run slow!")


def medium():
    time.sleep(0.5)
    print("I run a little slowly...")


def main():
    fast()
    slow()
    medium()

if __name__ == '__main__':
    main()
```

We can profile this code thusly:

```
>>> import cProfile
>>> import ptest
>>> cProfile.run('ptest.main()')
I run fast!
I run slow!
I run a little slowly...
         8 function calls in 3.500 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.500    3.500 :1()
        1    0.000    0.000    0.500    0.500 ptest.py:15(medium)
        1    0.000    0.000    3.500    3.500 ptest.py:21(main)
        1    0.000    0.000    0.000    0.000 ptest.py:4(fast)
        1    0.000    0.000    3.000    3.000 ptest.py:9(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    3.499    1.750    3.499    1.750 {time.sleep}
```

According to these results, the **slow()** and **main()** functions are the biggest time users. The overall execution of the module itself is also shown. Comparing our code to the results we can see that **main()** is slow only because it calls **slow()**, so we can then focus on the obvious culprit, **slow()**.

It's also possible to insider profiling *in our script* around particular function calls so we can focus our analysis.

```
profile = cProfile.Profile()
profile.enable()
main()                          # or whatever function calls we'd prefer to focus on
profile.disable()
```

Command-line interface to cProfile

```
python -m cProfile -o output.bin ptest.py
```

The **-m** flag on any Python invocation can import a module automatically. **-o** directs the output to a file. The result is a binary file that can be analyzed using the **pstats** module (which we see results in largely the same output as **run()**:

```
>>> import pstats
>>> p = pstats.Stats('output.bin')
>>> p.strip_dirs().sort_stats(-1).print_stats()
Thu Mar 20 18:32:16 2014    output.bin

         8 function calls in 3.501 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.501    3.501 ptest.py:1()
        1    0.001    0.001    0.500    0.500 ptest.py:15(medium)
        1    0.000    0.000    3.501    3.501 ptest.py:21(main)
        1    0.001    0.001    0.001    0.001 ptest.py:4(fast)
        1    0.001    0.001    3.000    3.000 ptest.py:9(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    3.499    1.750    3.499    1.750 {time.sleep}


<pstats.Stats instance at 0x017C9030>
```

Caveat: don't optimize prematurely

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
-- Donald Knuth

Common wisdom suggests that optimization should happen only once the code has reached a working, clear-to-finalized state. If you think about optimization too soon, you may do work that has to be undone later; or your optimizations may themselves get undone as you complete the functionality of your code.

Note: some of these examples taken from the "Mouse vs. Python (http://www.blog.pythonlibrary.org/)" blog.

Python Enhancement Packages

These packages provide varying approaches toward writing and running more efficient Python code.

- **PyPy**: a "Just in Time (https://en.wikipedia.org/wiki/Just-in-time_compilation)" compiler for Python -- can speed up almost any Python code.
- **Cython**: superset of the Python language that additionally supports calling of C functions and declaring C types -- good for building Python modules in C.
- **Pyrex**: compiler that lets you combine Python code with C data types, compiling your code into a C extension for Python.
- **Weave**: allows embedding of C code within Python code.
- **Shed Skin**: an experimental module that can translate Python code into optimized C++.

While **PyPy** is a no-brainer for speeding up code, the other libraries listed here require a knowledge of **C**. The deepest analysis of Python will incorporate efficient C code and/or take into account its underlying C implementation. Python is written in C, and operations that we invoke in Python translate to actions being taken by the compiled C code. The most advanced Python developer will have a working knowledge in C, and study the C structures that Python employs.

Team Tools

iPython

The iPython shell is a "smart shell" for running Python programs. It's a fantastic tool not only for running code snippets (in the same way as the Python interactive shell) but can be used to run scripts directly through the shell and cause the script to drop back to the shell for testing and analysis (similar to **pdb** but with enhanced capabilities).

command/statement history

command history	[up arrow to cycle through]
show all history (prior iPython commands)	%history
show prior commands (last 3 commands)	%history 1-3
upload prior commands to github's **gist** repo	%pastebin 1-3

environment and object inspection

show currently active variables	%who
attribute tab completion	type an object name and 'dot', then [Tab]
show info about object (docstring, function signature, file location)	[object]?
show source, file, function signature	[object]??
show object docstring (for functions)	%pdoc [object]
search for a variable	%psearch [name]

run python scripts and code

run a python script, stop without exiting with variables available	%run [python script]
run a program in debug mode	%debug [scriptname]
open a temporary editor to enter code	%edit

external commands

built-in shell commands	%ls, %cd, %pwd
set aliases to shell commands	%alias [shortcut] [command-line command]

git: version control system

Version control refers to a system that keeps track of all changes to a set of documents (i.e., the revisions, or *versions*). Its principal advantages are: 1) the ability to restore a team's codebase to any point in history, and 2) the ability to *branch* a parallel codebase in order to do development without disturbing the **main** branch, and then *merge* a development branch back into the **main** branch.

Git was developed by Linus Torvalds (https://en.wikipedia.org/wiki/Linus_Torvalds) (the creator of Linux) to create an open-source version of **BitKeeper**, which is a distributed, lightweight and very fast VCS. Other popular systems include **CSV**, **SVN** and **ClearCase**.

Git's docs (https://git-scm.com/) are very clear and extensive. Here is a quick rundown of the highlights:

Creating or cloning a new repository (https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository)
Adding, changing and committing files to the local repository (https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository)
Reviewing commit history (https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History) with git log

Sign up for git (if new to git)
To begin working with git, you must have a github.com account. On the home page (http://www.github.com) enter a username, email address and password and click "Sign Up" to get started.

Creating a new repository
Each repository holds any number of folders and files that relate to a group of software projects (usually the codebase for a team of developers).

A *repository* is used to house a *codebase* -- that is, all of the software projects for a company, a team, a class, an overall software production effort, or an individual. One team can use the same repository for years, or you can choose to create separate repositories for separate projects.

See the Git: Getting Started () handout for specific instructions.

The git commit cycle: add or change file, commit, push

This cycle is repeated as we make changes and add these changes to git.

Here is the life cycle of a change to the repository:

add or change a file
Whether we are creating a new file or changing an existing file, **git** will track the changes we make.

git add the file (whether added or changed) to the staging area
This applies to new or changed files -- "add" simply refers to any changes.

```
$ git add test-1.py
```

git commit the file to the *local repository*
The changes are committed *locally* -- they are not yet pushed to the server.

```
$ git commit -m 'important change to some files'
```

git pull to pull down any changes that have been made by other contributors
We customarily do this before pushing any changes.

```
$ git pull
```

git push to push local commit(s) to the remote repo
This command pushes any *committed changes* from the local to the remote repository.

```
$ git push
```

Once we've pushed changes, we should be able to see the changes on **github.com** (or our company's private remote repo).

git branching basics

A branch is a "line of development", made up of a series of committed changes toward a particular new feature or bug fix.

see what branches are available	**git branch**
create a new branch	**git branch *newfeature***
switch to the new branch	**git checkout *newfeature***
add and/or change file(s)	[edit file]
commit changes	git commit -m 'making changes'
push to the branch	git push
if needed, push the branch to the remote repo	**git push origin HEAD**
switch back to master branch	**git checkout master**
merge the changed branch back to master	**git merge *newfeature***
push the branch to remote	**git push origin master**
delete the *newfeature* branch locally	**git branch -d *newfeature***
delete the *newfeature* branch remotely	**git push origin :*newfeature***

git's tutorial on branches (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell) is an extremely clear approach to documenting this sometimes challenging topic.

PEP257: docstrings

PEP257 (https://www.python.org/dev/peps/pep-0257/) defines the overall spec for docstrings.

If you violate these conventions, the worst you'll get is some dirty looks. But some software (such as the Docutils docstring processing system) will be aware of the conventions, so following them will get you the best results.

-- David Goodger, Guido van Rossum

The actual format is discretionary; below I provide a basic format.

A **docstring** is a string that appears, *standalone*, as the first string in one of the following:

a script/module
a class
a function or method

The presence of this string *as the very first statement* in the module/class/function/method populates the __doc__ attribute of that object.

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

"""myprog.py:  show a sample of docstrings"""

def myfunc():
    """this is myfunc"""

print __doc__                # 'myprog.py:  show a sample of docstrings'
print myfunc.__doc__         # 'this is myfunc'
```

All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the __init__ constructor) should also have docstrings. A package may be documented in the module docstring of the __init__.py file in the package directory.

It is also allowable to add a docstring after each variable assignment inside a module, function or class. Everything except for the format of the initial docstring (which came from a module I wrote for AppNexus) is an example from PEP257:

```
"""
NAME
    populate_account_tables.py

DESCRIPTION
    select data from SalesForce, vertica and mysql
    make selected changes
    insert into fiba mysql tables

SYNOPSIS
    populate_account_tables.py
    populate_account_tables.py table1 table2     # process some tables, not others

VERSION
    2.0 (using Casserole)

AUTHOR
    David Blaikie (dblaikie@appnexus.com)
"""

num_tried = 3
"""Number of times to try the database before failing out."""

class Process(object):

    """ Class to process data. """

    c = 'class attribute'
    """This is Process.c's docstring."""

    def __init__(self):
        """Method __init__'s docstring."""

        self.i = 'instance attribute'
        """This is self.i's docstring."""

def split(x, delim):
    """Split strings according to a supplied delimeter."""

    splitlist = x.split(delim)
    """the resulting list to be returned"""

    return splitlist
```

These docstrings can be automatically read and formatted by a docstring reader. **docutils** is a built-in processer for docs, although the documentation for *this* module is a bit involved. We'll discuss the more-easy-to-use **Sphinx** next.

Sphinx: easy docs from docstrings

Docstrings are read directly from Python code, and automatically built into documentation documents in the desired format (default HTML).

install Sphinx
Sphinx is included with Anaconda Python. It requires **docutils** and **Jinja2**, which are also included with Anaconda Python.

create a directory for docs as well as for source
It's important to have a dedicated source directory for a project (customarily called **src**, but up to your discretion). You should also create a separate directory for documentation.

run sphinx-quickstart
This automated script asks a series of questions, most of which should be accepted with the suggested defaults, except for two: say 'yes' to separate source and build directories; and say 'yes' to 'autodoc' (which automatically geneates documentation)

edit conf.py
This file generated by sphinx-quickstart should be informed of the location of your source with a basic python statement:

```
sys.path.insert(0, '../src')   # relative path to your docs
```

run sphinx-apidoc
This script finds all of the modules and classes within your code and creates special .rst files for each

make html
This command generates documentation.

I found the process of generating documentation to be less than smooth, but with practice it became easier. A few hours following Giselle Zeno's tutorial (http://gisellezeno.com/tutorials/sphinx-for-python-documentation.html) should be sufficient to become comfortable with Sphinx.

I built some Sphinix docs against a test lib - they look very pretty (../data/docs/build/html/py-modindex.html) and suspiciously similar to documentation for Flask (http://flask.pocoo.org/docs/0.10/).

pdb: the Python Debugger

Print statements and **raw_input()** are useful tools in debugging. But sometimes a dedicated *debugger* can give us a lot more power in reviewing our programs *as they are running*.

A **pdb.set_trace()** statement placed in your code will cause program execution to pause (similar to **raw_input()**), but allowing you to interact with the script's variables while the script is paused.

```
import pdb

def dosomething(aa):
    print "now I'm inside dosomething()
    aa = aa + 5
    return aa

print 'hello, debugger'

for counter in range(5):
    countsum = 0
    countsum = countsum + counter
    pdb.set_trace()

pdb.set_trace()
newval = dosomething(5)
print "now I'm moving on"
```

At the point that python reaches the **pdb.set_trace()** statement, it will pause and you now have the opportunity to examine and set variables, call functions, etc.

There are several control options when working within the paused shell, but here are the ones I use most often:

```
n(ext):  execute next statement
s(tep):  execute next statement (and enter a function if one is encountered)
c(ontinue):  resume program execution until the next stopping point
```

The Zen of Python

One of the advantages of Python is its emphasis on design elegance. The non-backwards-compatible **Python 3** is a clear indication that doing things "the right way" (however it may be defined) is more important than popularity.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

PEP8: the Python design standard

A **PEP** or **Python Enhancement Proposal** is a series of numbered discussion documents that propose new features to be added to Python. A significant discussion over whether or not an addition is useful, meaningful, worthwhile, and "Pythonic" accompanies each proposal. Proposals are accepted or rejected by common consensus (with special weight given to the opinion of **Guido van Rossum**, the proclaimed **Benevolent Dictator for Life**).

PEP8 (https://www.python.org/dev/peps/pep-0008/) is often cited in online discussions as the standard guide for Python style and design. It treats both specific style decisions (such as whether to include a space before an operator; the case and format of variable names; whether to include one blank line or two after a function definition) as well as design considerations (how to design for inheritance, when/how to write return statements, etc.).

While the proposal itself proclaims "A Foolish Consistency is the Hobgoblin of Little Minds" it is generally accepted as the standard to follow.

"*A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!*"

-- Tim Peters on comp.lang.python, 2001-06-16

important conventions in **PEP8**

4-space indents
maximum line length: 79 characters
wrapping long lines with parentheses
2 blank lines before and after functions
imports on separate lines, 'absolute' imports recommended
single space on either sides of operators
naming style is not recommended, but general convention is lowercase_with_underscores

Adhering to the 79-character limit

The original CRT terminal screens had an 80-character width; this is how the 79-character limit was born (wrapped lines are difficult to read).

Many text editors and IDEs can be set to display the 80-character limit onscreen and/or warn the user when the limit exceeded.

Even though modern screens allow for pretty much any length (since you can size the text to any size), the 79-character limit is still considered an important discipline, since it will allow multiple files to be displayed side-by-side.

When attempting to adhere to this limit, we must learn how to wrap our lines. Wrapping long lines is usually done with parentheses, for example:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

I have found that my preference for explicit variable names makes respecting this limit to make life more difficult.

Some developers hold an opinion (acknowledged by PEP8) that the 79-character limit is antiquated (since it was based in the old terminal width) and that another limit, up to 100 characters, should be adopted. PEP8 acknowledges that with agreement among developers, it should be permissible to extend the limit. You can see some of what is said regarding the debate in this spirited discussion (http://stackoverflow.com/questions/4841226/how-do-i-keep-python-code-under-80-chars-without-making-it-ugly).

Getting Started with Github

Launching a new Repository

A *repository* is used to house a *codebase* -- that is, all of the software projects for a company, a team, a class, an overall software production effort, or an individual. One team can use the same repository for years, or you can choose to create separate repositories for separate projects.

Create a new repository
Start by creating a new repository on **github.com**. (You can also use an existing repository if you have created or have access to one.)

1. Create and name the repository
a. If this is the first project on your github account, click the green 'Start a Project' button; or if not the first, click on 'Repositories' and click the green 'New' button.
b. Name the repository; leave the 'Public' option selected
c. Click the green 'Create repository' button

2. Prepare to create the local project folder. Github.com shows you options for creating the local project folder; we will use the command-line version (i.e., not using github Desktop).
a. Click the SSH button under "Quick Setup -- if you've done this kind of thing before"; the text beneath changes slightly
b. Copy to clipboard the text beneath under "Quick setup" (you can use the clipboard button to the right)

3. At the terminal, decide upon a location and create a new directory; cd into that directory

```
mkdir projdir           # (where projdir is the name of your repository)
cd projdir
```

4. Initialize a new (blank) local repository.

```
git init
```

This creates a special **.git** directory here. You'll see a message to this effect.

5. Create a new README.md text file to add to the repo. In the text file put the following text:

```
# reponame      # (where reponame is the name of your project)
```

Save this file in the new directory.

6. Add the new file

```
git add README.md
```

7. Commit and add the file

```
git commit –m "first commit"
```

You'll see a message ackowledging the changes and the commit.

8. Add the remote repo and push committed changes. Here you will use the text copied to clipboard, where the italicized text is shown:

```
git remote add origin git@github.com:username/reponame.git    # replace italicized text with text copied to
                                                               # (this will be needed only once)
git push –u origin master
```

You'll see a series of messages indicating a warning (re: adding a RSA host)

Having successfully pushed a first file to the remote repository, your local folder is now connected and you can push changes remotely with **git push** at any time.

If you make a mistake initially and find you are having problems pushing, you can simply delete the repo by logging into github.com, clicking on the repository, choosing Settings, and scrolling down to "Delete this repository" at the bottom. (The entire repo including history of changes, comments, etc., will be deleted, so it should only be done to start from scratch.) Please follow the steps carefully and let me know if you have any problems -- thanks!

The git commit cycle: add or change file, commit, push

This cycle is repeated as we make changes and add these changes to git.

add or change a file (in this case I modified **test-1.py**)

git status to see that the file has changed
This command shows us the status of all files on our system: modified but not staged, staged but committed and committed but not pushed.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)

        modified:   test-1.py

no changes added to commit (use "git add" and/or "git commit -a")
$
```

git add the file (whether added or changed) to the staging area

```
$ git add test-1.py
$
```

git status to see that the file has been added

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD ..." to unstage)

        modified:   test-1.py

$
```

git commit the file to the *local repository*

```
$ git commit -m 'made a trivial change'
[master e6309c9] made a trivial change
 1 file changed, 4 insertions(+)
$
```

git status to see that the file has been committed, and that our *local* repository is now "one commit ahead" of the *remote* repository (known as *origin*

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
```

git pull to pull down any changes that have been made by other contributors

```
$ git pull
Already up-to-date.
$
```

git push to push local commit(s) to the remote repo
The *remote repo* in our case is **github.com**, although many companies choose to host their own private remote repository.

```
$ git push
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/NYU-Python/david-blaikie-solutions
   2ce8e49..e6309c9  master -> master
$
```

Once we've pushed changes, we should be able to see the changes on **github.com** (or our company's private remote repo).

Flask

Prelude: HTML Pages and HTTP Requests

HTML pages can be saved and viewed locally in a browser as well as placed on a server as part of a website.

An extremely **basic HTML page** (../python_data/basic.html)

```
<HTML>
  <HEAD>
    <TITLE>The Page of Greeting</TITLE>
  </HEAD>
  <BODY>
  Hello!
  </BODY>
</HTML>
```

"Static" HTML pages are plaintext files that sit on a computer's disk. When requested through a web server, the server simply finds the file and returns it to the browser. When requested through your local computer's filesystem, the browser simply reads the file off of the disk in the same way any file reader might.

We can use links in HTML pages to call Flask apps, and Flask apps to present HTML pages to **display information** (../python_data/greeting.html).

greeting.html

```
<HTML>
  <HEAD>
    <TITLE>The Page of Greeting</TITLE>
  </HEAD>
  <BODY>
    <H1>Greetings!</H1>
    You've reached my page.  <BR><BR><BR>


    Visit this LINK:  <A HREF="https://www.yahoo.com/">yahoo!</A><BR><BR><BR>


    Check out these PREFORMATTED FIGURES:<BR>
    <PRE>
           23.9      22.8     1117.0
            1.8      17.0       55.0
    </PRE>
    <BR><BR>

    Here is a TABLE:<BR>
    <TABLE border="1" width="75%" cellpadding="20" align="center">
      <TR>
        <TH>Date</TH><TH>City</TH><TH>Avg. Temp (F)</TH>
      </TR>
      <TR>
        <TD>2017-03-09</TD><TD>Hamburg</TD><TD align="right">63</TD>
      </TR>
      <TR>
        <TD>2017-03-10</TD><TD>Paree</TD><TD align="right">61</TD>
      </TR>
    </TABLE>
    <BR><BR><BR>


    Here is a FORM!<BR><BR>
    <FORM action="http://localhost:5000/">
      <INPUT NAME="id" TYPE="hidden">             # submits a non-visible values
      <B>What is your name?</B><BR>
      <INPUT NAME="name" size="50"><BR><BR>
      <B>What is your favorite color?</B><BR>
      <INPUT NAME="color" size="50"><BR><BR>
      <INPUT TYPE="submit" VALUE="answer!">
    </FORM>


  </BODY>
</HTML>
```

The basic items to note above are these:

overall structure	<HTML>, <HEAD>, <TITLE> and <BODY>
heading	<H1> (or <H2>, <H3>, etc. -- larger numbers reduce size)
hyperlinks	<A HREF>
plain text can go here -- convenient for formatting	<PRE>
standard HTML table	<TABLE>, <TR>, <TH>, <TD>
form elements for submitting info from a page	<FORM>, <INPUT>

Web Frameworks

Introduction

A "web framework" is an application or package that facilitates web programming. Server-side apps (for example: a catalog, content search and display, reservation site or most other interactive websites) use a *framework* to handle the details of the web network request, page display and database input/output -- while freeing the programmer to supply just the logic of how the app will work.

Full Stack Web Frameworks
A web application consists of layers of components that are configured to work together (i.e., built into a "stack"). Such components may include:

* authenticating and identifying users through cookies
* handling data input from forms and URLs
* reading and writing data to and from persistant storage (e.g. databases)
* displaying templates with dynamic data inserted
* providing styling to templates (e.g., with css)
* providing dynamic web page functionality, as with AJAX

The term "full stack developer" used by schools and recruiters refers to a developer who is proficient in all of these areas.

Django is possibly the most popular web framework. This session would probably focus on Django, but its configuration and setup requirements are too lengthy for the available time.

"Lightweight" Web Frameworks
A "lightweight" framework provides the base functionality needed for a server-side application, but allows the user to add other stack components as desired. Such apps are typically easier to get started with because they require less configuration and setup.

Flask is a popular lightweight framework with many convenient defaults allows us to get our web application started quickly.

The Flask app object, the app.run() method and @app.route dispatch decorators

"@app.route()" functions describe what happens when the user visits a particular "page" or URL shown in the decorator.

hello_flask.py

Here is a basic template for a **Flask app** (http://homework-davidbpython.rhcloud.com/hello).

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

import flask
app = flask.Flask(__name__)           # a Flask object

@app.route('/hello')                 # called when visiting web URL 127.0.0.1:5000/hello/
def hello_world():
    print '*** DEBUG:  inside hello_world() ***'
    return '<PRE>Hello, World!</PRE>'            # expected to return a string (usu. the HTML to display)

if __name__ == '__main__':
    app.run(debug=True, port=5000)    # app starts serving in debug mode on port 5000
```

The first two lines and last two lines will always be present (production apps will omit the **debug=** and **port=** arguments).

app is an object returned by Flask; we will use it for almost everything our app does. We call it a "god object" because it's always available and contains most of what we need.

app.run() starts the Flask application server and causes Flask to wait for new web requests (i.e., when a browser visit the server).

@app.route() functions are called when a particular URL is requested. The decorator specifies the string to be found at the end of the URL. For example, the above decorator **@app.route('/hello')** specifies that the URL to reach the function should be **http://localhost:5000/hello/**

The string returned from the function is passed on to Flask and then to the browser on the other end of the web request.

Running a Flask app locally

Flask comes complete with its own self-contained app server. We can simply run the app and it begins serving locally. No internet connection is required.

```
$ python hello_flask.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
127.0.0.1 - - [13/Nov/2016 15:58:16] "GET / HTTP/1.1" 200 -
*** DEBUG:  inside hello_world() ***
```

The Flask app prints out web server log messages showing the URL requested by each visitor. You can also print error messages directly from the application, and they will appear in the log (these were printed with *** strings, for visibility.)

Changes to Flask code are detected and cause Flask to restart the server; errors cause it to exit

Whenever you make a change and save your script, the Flask server will restart -- you can see it issue messages to this effect:

```
 * Detected change in '/Users/dblaikie/Dropbox/tech/nyu/advanced_python/solutions/flask/guestbook/guestbook
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 161-369-356
```

If there is an error in your code that prevents it from running, the code will raise an exception and exit.

```
 * Detected change in '/Users/dblaikie/Dropbox/tech/nyu/advanced_python/solutions/flask/guestbook/guestbook
 * Restarting with stat
  File "./guestbook_simple.py", line 17
    return hello, guestbook_id ' + guestbook_id
                                                ^
SyntaxError: EOL while scanning string literal
```

At that point, the browser will simply report "This site can't be reached" with no other information.

Therefore we must keep an eye on the window to see if the latest change broke the script -- fix the error in the script, and then re-run the script in the Terminal.

Redirection and Event Flow

Once an 'event' function is called, it may return a string, call another function, or redirect to another page.

Return a plain string

A plain string will simply be displayed in the browser. This is the simplest way to display text in a browser.

```
@app.route('/hello')
def hello_world():
    return '<PRE>Hello, World!</PRE>'            # expected to return a string (usu. the HTML to display)
```

Return HTML (also a string)

HTML is simply tagged text, so **it is also returned as a string** (http://homework-davidbpython.rhcloud.com/hello_template).

```
@app.route('/hello_template')
def hello_html():
    return """
<HTML>
  <HEAD>
    <TITLE>My Greeting Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Hello, world!</H1>
  </BODY>
</HTML>"""
```

Return an HTML Template (to come)

This method also returns a string, but it is returned from the **render_template()** function.

```
@app.route('/hello_html')
def hello_html():
    return flask.render_template('response.html')    # found in templates/response.html
```

Return another function call

Functions that aren't intended to return strings but to perform other actions (such as making database changes) can simply call other functions that represent the desired destination:

```
def hello_html():
    return """
<HTML>
  <HEAD>
    <TITLE>My Greeting Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Hello, world (from another function)  !</H1>
  </BODY>
</HTML>"""

@app.route('/hello_template')
def hello():
    return hello_html()
```

Because **hello()** is calling **hello_html()** in its **return** statement, whatever is returned from there will be returned from **hello()**.

Redirecting to another program URL with flask.redirect() and flask.url_for()

At the end of a function we can call the flask app again through a page redirect -- that is, to have **the app** (http://homework-davidbpython.rhcloud.com/shello) call itself with new parameters.

```
from datetime import date

@app.route('/sunday_hello')
def sunday_hello():
    return "It's Sunday!  Take a rest!"

@app.route('/shello')
def hello():
    if date.today().strftime('%a') == 'Sun':
        return flask.redirect(flask.url_for('sunday_hello'))
    else:
        return 'Hello, workday (or Saturday)!'
```

redirect() issues a redirection to a specified URL; this can be **http://www.google.com** or any desired URL.

url_for() simply produces the URL that will call the flask app with **/login**.

Building a Multi-Page Application

Use flask.url_for() to build links to other apps

This app (http://homework-davidbpython.rhcloud.com/question) has three pages; we can build a "closed system" of pages by having each page link to another within the site.

```
happy_image = 'http://davidbpython.com/advanced_python/python_data/happup.jpg'
sad_image = 'http://davidbpython.com/advanced_python/python_data/sadpup.jpg'

@app.route('/question')
def ask_question():
    return """
<HTML>
  <HEAD><TITLE>Do you like puppies?</TITLE></HEAD>
  <BODY>
      <H3>Do you like puppies?</H3>
      <A HREF="{}">arf!</A><BR>
      <A HREF="{}">I prefer cats...</A>
  </BODY>
</HTML>""".format(flask.url_for('yes'), flask.url_for('no'))


@app.route('/yes')
def yes():
    return """
<HTML>
  <HEAD><TITLE>C'mere Boy!</TITLE></HEAD>
  <BODY>
      <H3>C'mere, Boy!</H3>
      <IMG SRC="{}"><BR>
      <BR>
      Change your mind?  <A HREF="{}">Let's try again.</A>
  </BODY>
</HTML>""".format(happy_image, flask.url_for('ask_question'))


@app.route('/no')
def no():
    return """
<HTML>
  <HEAD><TITLE>Aww...</TITLE></HEAD>
  <BODY>
      <H3>Aww...really?</H3>
      <IMG SRC="{}"><BR>
      <BR>
      Change your mind?  <A HREF="{}">Let's try again.</A>
  </BODY>
</HTML>""".format(sad_image, flask.url_for('ask_question'))
```

Simple Templating

Use {{ varname }} to create template tokens and flask.render_template() to insert to them.

HTML pages are rarely written into Flask apps; instead, we use standalone *template* files.

The template files are located in a **templates** directory placed in the same directory as your Flask script.

question.html

```
<HTML>
  <HEAD><TITLE>Do you like puppies?</TITLE></HEAD>
  <BODY>
      <H3>Do you like puppies?</H3>
      <A HREF="{{ yes_link }}">arf!</A><BR>
      <A HREF="{{ no_link }}">I prefer cats...</A>
  </BODY>
</HTML>
```

puppy.html

```
<HTML>
  <HEAD><TITLE><{{ title_message }}</TITLE></HEAD>
  <BODY>
      <H3>{{ title_message }}</H3>
      <IMG SRC="{{ puppy_image }}"><BR>
      <BR>
      Change your mind?  <A HREF="{{ question_link }}">Let's try again.</A>
  </BODY>
</HTML>
```

puppy_question.py

```
happy_image = 'http://davidbpython.com/advanced_python/python_data/happup.jpg'
sad_image =   'http://davidbpython.com/advanced_python/python_data/sadpup.jpg'


@app.route('/question')
def ask_question():
    return flask.render_template('question.html',
                                 yes_link=flask.url_for('yes'),
                                 no_link=flask.url_for('no'))

@app.route('/yes')
def yes():
    return flask.render_template('puppy.html',
                                 puppy_image=happy_image,
                                 question_link=flask.url_for('ask_question'),
                                 title_message='Cmere, boy!')

@app.route('/no')
def no():
    return flask.render_template('puppy.html',
                                 puppy_image=sad_image,
                                 question_link=flask.url_for('ask_question'),
                                 title_message='Aww... really?')
```

Embedding Python Code in Templates

Use {% %} to embed Python code for looping, conditionals and some functions/methods **from within the template** (http://homework-davidbpython.rhcloud.com/template_test).

Template document: "template_test.html"

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Important Stuff</title>
  </head>
  <body>

  <h1>Important Stuff</h1>

  Today's magic number is {{ number }}<br><br>

  Today's strident word is {{ word.upper() }}<br><br>

  Today's important topics are:<br>
  {% for item in mylist %}
      {{ item }}<br>
  {% endfor %}
  <br><br>

  {% if reliability_warning %}
      WARNING:  this information is not reliable
  {% endif %}

  </body>
</html>
```

Flask code

```
@app.route('template_test')
def template_test():
    return flask.render_template('template_test.html', number=1035,
                                                       word='somnolent',
                                                       mylist=['children', 'animals', 'bacteria'],
                                                       reliability_warning=True)
```

As before, **{{ variable }}** can used for variable insertions, as well as instance attributes, method and function calls

{% for this in that %} can be used for 'if' tests, looping with 'for' and other basic control flow

Reading args from URL or Form Input

Input from a page can come from a link URL, or from a form submission.

name_question.html

```
<HTML>
  <HEAD>
  </HEAD>
  <BODY>
    What is your name?<BR>
    <FORM ACTION="{{ url_for('greet_name') }}" METHOD="post">
      <INPUT NAME="name" SIZE="20">
      <A HREF="{{ url_for('greet_name') }}?no_name=1">I don't have a name</A>
      <INPUT TYPE="submit" VALUE="tell me!">
    </FORM>
  </BODY>
</HTML>
```

flaskapp.py (http://homework-davidbpython.rhcloud.com/name_question)

```
@app.route('/name_question')
def ask_name():
    return flask.render_template('name_question.html')

@app.route('/greet', methods=['POST', 'GET'])
def greet_name():
    name =    flask.request.form.get('name')     # from a POST (form with 'method="POST"')
    no_name = flask.request.args.get('no_name')  # from a GET (URL)

    if name:
        msg = 'Hello, {}!'.format(name)
    elif no_name:
        msg = 'You are anonymous.  I respect that.'
    else:
        raise ValueError('\nraised error:  no "name" or "no_name" params passed in request')

    return '<PRE>{}</PRE>'.format(msg)
```

Base templates

Many times we want to apply the same HTML formatting to a group of templates -- for example the <head> tag, which my include css formatting, javascript, etc.

We can do this with base templates:

```
{% extends "base.html" %}         # 'base.html' can contain HTML from another template
    <h1>Special Stuff</h1>
    Here is some special stuff from the world of news.
```

The base template "surrounds" any template that imports it, inserting the importing template at the **{% block body%}** tag:

```
<html>
  <head>
  </head>
  <body>
    <div class="container">
      {% block body %}
         <H1>This is the base template default body.</H1>
      {% endblock %}
    </div>
  </body>
</html>
```

There are **many other features** (http://jinja.pocoo.org/docs/dev/templates/) of Jinja2 as well as ways to control the API, although I have found the above features to be adequate for my purposes.

Sessions

Sessions (usually supported by cookies) allow Flask to identify a user between requests (which are by nature "anonymous").

When a session is set, a cookie with a specific ID is passed from the server to the browser, which then returns the cookie on the next visit to the server. In this way the browser is constantly re-identifying itself through the ID on the cookie. This is how most websites keep track of a user's visits.

flask_session.py (http://homework-davidbpython.rhcloud.com/sess)

```
import flask
app = flask.Flask(__name__)

app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'  # secret key

@app.route('/index')
def hello_world():

    # see if the 'login' link was clicked:  set a session ID
    user_id = flask.request.args.get('login')
    if user_id:
        flask.session['user_id'] = user_id
        is_session = True

    # else see if the 'logout' link was clicked:  clear the session
    elif flask.request.args.get('logout'):
        flask.session.clear()

    # else see if there is already a session cookie being passed:  retrieve the ID
    else:
        # see if a session cookie is already active between requests
        user_id = flask.session.get('user_id')

    tell the template whether we're logged in (user_id is a numeric ID, or None)
    return flask.render_template('session_test.html', is_session=user_id)


if __name__ == '__main__':
    app.run(debug=True, port=5001)    # app starts serving in debug mode on port 5001
```

session_test.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Session Test</title>
  </head>
  <body>

  <h1>Session Test</h1>

  {% if is_session %}
  <font color="green">Logged In</font>
  {% else %}
  <font color="red">Logged Out</font>
  {% endif %}

  <br><br>

  <a href="index?login=True">Log In</a><br>
  <a href="index?logout=True">Log Out</a><br>

  </body>
</html>
```

Config Values

Configuration values are set to control how Flask works as well as to be set and referenced by an individual application.

Flask sets a number of variables for its own behavior, among them **DEBUG=True** to display errors to the browser, and **SECRET_KEY='!jmNZ3yX R~XWX/r]LA098j/,?RTHH'** to set a session cookie's secret key.

A list of Flask default configuration values is **here**. (http://flask.pocoo.org/docs/0.11/config/)

Retrieving config values

```
value = app.config['SERVER_NAME']
```

Setting config values individually

```
app.config['DEBUG'] = True
```

Setting config values from a file

```
app.config.from_pyfile('flaskapp.cfg')
```

Such a file need only contain python code that sets uppercased constants -- these will be added to the config.

Setting config values from a configuration Object
Similarly, the class variables defined within a custom class can be read and applied to the config with **app.config.from_object()**. Note in the example below that we can use inheritance to distribute configs among several classes, which can aid in organization and/or selection:

In a file called **configmodule.py**:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite://:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

In the flask script:

```
app.config.from_object('configmodule.ProductionConfig')
```

Environment Variables

Environment Variables are system-wide values that are set by the operating system and apply to all applications. They can also be set by individual applications.

The OpenShift web container sets a number of environment variables, among them **OPENSHIFT_LOG_DIR** for log files and **OPENSHIFT_DATA_DIR** for data files. Flask employs **jinja2** templates.

A list of Openshift environment variables can be found (https://developers.openshift.com/managing-your-applications/environment-variables.html)here (https://developers.openshift.com/managing-your-applications/environment-variables.html).

Flask and security

An important caveat regarding web security: Flask is not considered to be a secure approach to handling sensitive data.

...at least, that was the opinion of a former student, a web programmer who worked for Bank of America, about a year ago -- they evaluated Flask and decided that it was not reliable and could have security vulnerabilities. His team decided to use **CGI** -- the baseline protocol for handling web requests.

Any framework is likely to have vulnerabilities -- only careful research and/or advice of a professional can ensure reliable privacy.

However for most applications security is not a concern -- you will simply want to avoid storing sensitive data on a server without considering security.

Flask-Specific Errors

Keep these Flask-specific errors in mind.

Page not found

URLs specified in @app.route() functions should not end in a trailing slash, and URLs entered into browsers must match

```
@app.route(/'hello')
```

If you omit a trailing slash from this path but include one in the URL, the browser will respond that it can't find the page.

What's worse, some browsers sometimes try to 'correct' your URL entries, so if you type a URL with a trailing slash and get "Page not found", the next time you type it differently (even if correctly) the browser may attempt to "correct" it to the way you typed it the first time (i.e., incorrectly). This can be extremely frustrating; the only remedy I have found is to clear the browser's browsing data.

Functions must *return* strings (or redirect to another function or URL); they *do not print* page responses.

```
@app.route(/'hello')
def hello():
    return 'Hello, world!'        # not print 'Hello, world!'
```

Each routing function expects a string to be returned -- so the function must do one of these:

1) return a string (this string will be displayed in the browser)
2) call another **@app.route()** function that will return a string
3) issue a URL redirect (described later)

Method not allowed usually means that a form was submitting that specifies **method="POST"** but the **@app.route** decorator doesn't specify **methods=['POST']**. See "Reading args from URL or Form Input", above.

name_question.html

```
<HTML>
  <HEAD>
  </HEAD>
  <BODY>
    What is your name?<BR>
    <FORM ACTION="{{ url_for('greet_name') }}" METHOD="post">
      <INPUT NAME="name" SIZE="20">
      <A HREF="{{ url_for('greet_name') }}?no_name=1">I don't have a name</A>
      <INPUT TYPE="submit" VALUE="tell me!">
    </FORM>
  </BODY>
</HTML>
```

If the form above submits data as a "post", the **app.route()** function would need to specify this as well:

```
@app.route('/greet', methods=['POST', 'GET'])
def greet_name():
   ...
```

Data Persistence

JSON for list/dict serialization

JSON is a text format that stores Python list and dict objects.

In a file called mystruct.json:

```
{
   "key1":  ["a", "b", "c"],
   "key2":  {
              "innerkey1": 5,
              "innerkey2": "woah"
            },
   "key3":  55.09,
   "key4":  "hello"
}
```

Python code to read mystruct.json

```
fh = open('mystruct.json')    # file contains JSON
mys = json.load(fh)           # load from a file
fh.close()

print mys['key2']['innerkey2']   # woah
```

Python code to write JSON to file

```
mys['key2']['innerkey2'] = 'oh yes'   # change struct

wfh = open('newstruct.json', 'w')  # open file for writing
json.dump(mys, wfh)                # write JSON to file
```

JSON format is stringent
Although it resembles Python structures, JSON notation is slightly less forgiving than Python: double quotes are required around strings, and no trailing comma is allowed after the last element in a dict or list.

Violating any JSON rules results in a failure to load the file, and a challenging error message:

```
{
  "key1":  "goodbye",
  "key4":  "hello",        # no trailing comma allowed
}
```

When I then tried to load it, the **json** module complained with an error location (although in a larger file this could take time to locate in the file):

```
ValueError: Expecting property name: line 3 column 18 (char 42)
```

Similarly, if you receive the message **JSON object could not be decoded**, it means that the **json** module couldn't make sense of the file you asked it to open. If you opened and read an empty file into json you would get this message. The solution is to make sure to use the module to write the file (using **dump()** or **dumps()**) before trying to open it for reading. You might also get the error if you attempted to hand-edit the JSON file and made a syntactic mistake. Unfortunately JSON does not make it easy to find the error -- the better solution is to write the file afresh using the module.

pickle for object serialization

Any Python data object can be written to disk without modification.

```
import pickle
br = [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
F = open('datafile.pickle', 'w')
pickle.dump(br, F)
F.close()

## later
G = open('datafile.txt')
J = pickle.load(G)
print J
    #  [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
```

This saves the added step of rearranging data in an object into a structure that can be stored conventionally (for example in a relational database, CSV file or even JSON file).

pickle with complex data objects

For ease of use, custom-designed objects are superior. As they can behave like Python objects, they can be used in a variety of familiar situations. As they are custom-designed, they can be made to hold data in any form.

Since custom-designed objects can be serialized using pickle, they can be stored directly without using object<->relational or object<->JSON translations. pickle is slower than RDBS and JSON, but it is a convenient option for lightweight applications.

Consider this pair of data objects: one holds items and the other is an item:

```
import pickle

class ItemList(object):

    def __init__(self, name):
        self.name = name
        self.items = []
        self.index = -1

    def add(self, text):
        self.items.append(Item(text))

    def __iter__(self):
        return self

    def next(self):
        self.index += 1
        if self.index > len(self.items) - 1:
            raise StopIteration
        return self.items[self.index]

class Item(object):
    def __init__(self, text):
        self.text = text
```

ItemList is designed to store and recover **Item** objects, and **Item** objects are designed to hold a line of text (obviously each of these could be much more complex).

Since we designed these objects, they are easy to use:

```
this_list = ItemList('mylist')

this_list.add('line 1')
this_list.add('line 2')
this_list.add('line 3')
```

pickle makes object storage and recovery a snap:

```
wfh = open('picklefile.pickle', 'w')
pickle.dump(this_list, wfh)
wfh.close()

### THE NEXT DAY...

fh = open('picklefile.pickle')

resurrected_list = pickle.load(fh)

print 'items for {}'.format(resurrected_list.name)
for item in resurrected_list:
    print '  ' + item.text
```

But because we have designed the objects around certain built-in functionality, it's possible to plug them directly into an unrelated feature such as Jinja2 template, without performing any transformations:

Template show_items.html

```
<h1>Items for {{ resurected_list.name }}</h1>
{% for item in resurrected_list %}
  {{ item }}
{% endfor %}
```

Flask code for above

```
  fh = open('picklefile.pickle')
  thislist = pickle.load(fh)

  return render_template('show_items.html', resurrected_list=thislist)
```

SQLite

SQLite is a simple database, similar to MySQL. We can

```
def insert_db(fname, lname):
    INSERT_QUERY = "INSERT INTO {} (fname, lname) VALUES '{}', '{}'"
    sql = INSERT_QUERY.format(TABLE_NAME, fname, lname)
    execute_query(sql)

def get_db_data(table_name):
    SELECT_QUERY = "SELECT * from {}"
    sql = SELECT_QUERY.format(table_name)
    c = execute_query(sql)
    return c.fetchall()
```

We have also defined a function that can execute any query provided to it. Our system creates a database connection when needed, and then stores it in the attribute of a special object called **flask.g**. This all-purpose storage object is available through the module directly, so it has global scope.

```
def execute_query(sql):
    conn = get_db()
    c = conn.cursor()
    c.execute(sql)
    conn.commit()
    return c

def get_db():
    db = getattr(flask.g, '_database', None)
    if db is None:
        db = flask.g._database = sqlite3.connect(DATABASE)
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(flask.g, '_database', None)
    if db is not None:
        db.close()
```

Also included here is a special function that Flask can invoke automatically when the program is exiting -- it closes the database connection we stored in **flask.g**.

Regular Expressions: Matching

Regular Expressions: Introduction

Regular Expressions, or "Regexes" refer to a declarative language used to match patterns in text, and are used for text validation/inspection, and text extraction.

Previously, we have had limited tools for inspecting text:

In the case of *fixed-width* text, we have been able to use a **slice**.

```
line = '19340903  3.4 0.9'
year = line[0:4]                # year == 1934
```

In the case of *delimited* text, we have been able to use **split()**

```
line = '19340903,3.4,0.9'
els = line.split(',')

mkt_rf = els[1]                  # 3.4
```

In the case of *formatted* text, there is no obvious way to do it.

Regular Expressions: Preview

This regex pattern matches elements of a web server log line in a single statement, without resorting to complicated splitting, slicing or string inspection methods.

The bolded portions show log line elements that we wish to extract; we are doing so using the parentheses.

```
import re

log_line = '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'

reg = re.search(r'(\d{2,3}\.\d{2,3}\.\d{2,3}\.\d{2,3}) - - \[(\d\d\/\w{3}\/\d{4}):(\d\d:\d\d:\d\d) (\-?\d\d
print type(reg)

print reg.group(1)   # 66.108.19.165
print reg.group(2)   # 09/Jun/2003
print reg.group(3)   # 19:56:33
print reg.group(4)   # -0400
```

Patterns look for various *classes* of text (for example, numbers or letters), as well as literal characters, in various *quantities*, reading through consecutive characters of the text.

Reading from left to right, the pattern (shown in the **r''** string) says this:

```
  2-3 digits, a period, 2-3 digits, a period, 2-3 digits, a period, 2-3 digits,
  followed by a space, dash, space, dash,
  followed by an open square bracket, 2 digits, forward slash, 3 word characters, forward slash, 4 digits,
  followed by a colon, 2 digits, colon, 2 digits, colon, 2 digits, space
  followed by a dash and 4 digits.
```

The parentheses identify text to be extracted. You can see the text that was extracted in the output.

The *re* module and *re.search()* function

re.search() returns **True** if the pattern matches the text.

```
import re          # import the regex library

if re.search(r'~jjk265', line):
    print line                          # prints any line with the characters ~jjk265
```

re.search() takes two arguments: the *string pattern*, and the string to be searched. Normally used in an **if** expression, it will evaluate to **True** if the pattern matched.

```
# weblog contains string lines like this:
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'

# script snippet:
for line in weblog.readlines():
  if re.search(r'~jjk265', line):
    print line                      # prints 2 of the above lines
```

The raw string (r'')

The raw string is like a normal string, but it does not process *escapes*. An escaped character is one preceded by a backslash, that turns the combination into a special character. **\n** is the one we're famliar with - the escaped **n** converts to a *newline character*, which marks the end of a line in a multi-line string.

A raw string wouldn't process the escape, so **r'\n'** is literally a backslash followed by an n.

```
var = "\n"           # one character, a newline
var2 = r'\n'         # two characters, a backslash followed by an n
```

"not" for negating a search

not is used to negate a search: "if the pattern does not match".

search file

```
  '66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:44 -0400] "GET /~dbb212/mysong.mp3 HTTP/1.1" 200 175449'
  '66.108.19.165 - - [09/Jun/2003:19:56:45 -0400] "GET /~jjk265/cd2.jpg HTTP/1.1" 200 175449'
```

code

```
for line in weblog.readlines():
    if not re.search(r'~jjk265', line):
        print line                    # prints 1 of the above lines -- the one without jjk265
```

The re Vocabulary

These terms, used in combination, cover most needs in composing text matching patterns.

Anchor Characters and the Boundary Character	$, ^, \b
Character Classes	\w, \d, \s, \W, \S, \D
Custom Character Classes	[aeiou], [a-zA-Z]
The Wildcard	.
Quantifiers	+, *, ?
Custom Quantifiers	{2,3}, {2,}, {2}
Groupings	(parentheses groups)

Patterns can match anywhere, but must match on consecutive characters

Read the string from left to right, looking for the first place the pattern matches on consecutive characters.

```
import re

str1 = 'hello there'
str2 = 'why hello there'
str3 = 'hel lo'

if re.search(r'hello', str1):  print 'matched'   # matched
if re.search(r'hello', str2):  print 'matched'   # matched
if re.search(r'hello', str3):  print 'matched'   # does not match
```

Note that 'hello' matches at the start of the first string and the middle of the second string. But it doesn't match in the third string, even though all the characters we are looking for are there. This is because the space in **str3** is unaccounted for - always remember - matches take place on *consecutive characters*.

Anchors and Boundary

Boundary characters **$** and **^** require that the pattern match starts at the beginning of the string or ends at the end of the string.

This program lists only those files in the directory that end in '.txt':

```
import os, re
for filename in os.listdir(r'/path/to/directory'):
    if re.search(r'\.txt$', filename):     # look for '.txt' at end of filename
        print filename
```

This program prints all the lines in the file that don't begin with a hash mark:

```
for text_line in open(r'/path/to/file.py'):
    if not re.search(r'^#', text_line):        # look for '#' at start of filename
        print text_line
```

When they are used as anchors, we will always expect **^** to appear at the start of our pattern, and **$** to appear at the end.

Character Classes

A *character class* is a special pattern entity can match on any of a group of characters: any of the "digit" class (0-9), any of the "word" class (letters, numbers and underscore), etc.

```
user_input = raw_input('please enter a single-digit integer: ')
if not re.search(r'^\d$', user_input):
    exit('bad input:  exiting...')
```

\d	[0-9]	(Digits)
\w	[a-zA-Z0-9_]	(Word characters -- letters, numbers or underscores)
\s	[\n\t]	('Whitespace' characters -- spaces, newlines, or tabs)

So a **\d** will match on a **5**, **9**, **3**, etc.; a **\w** will match on any of those, or on **a**, **Z**, _ (underscore). Keep in mind that although they match on any of several characters, a single instance of a character class matches on only one character. For example, a **\d** will match on a single number like '5', but it won't match on both characters in '55'. To match on 55, you could say **\d\d**.

Built-in Character Class: digits

The **\d** character class matches on any digit.

This example lists only those files with names formatted with a particular syntax -- YYYY-MM-DD.txt:

```
import re
dirlist = ('.', '..', '2010-12-15.txt', '2010-12-16.txt', 'testfile.txt')
for filename in dirlist:
    if re.search(r'^\d\d\d\d-\d\d-\d\d\.txt$', filename):
        print filename
```

Here's another example, validation: this regex uses the pattern ^\d\d\d\d$ to check to see that the user entered a four-digit year:

```
import re
answer = raw_input("Enter your birth year in the form YYYY\n")
if re.search(r'^\d\d\d\d$', answer):
    print "Your birth year is ", answer
else:
    print "Sorry, that was not YYYY"
```

Built-in Character Class: "word" characters

The **\w** character class matches on any number, letter or underscore.

In this example, we require the user to enter a username with any word characters:

```
username = raw_input()
if not re.search(r'^\w\w\w\w\w$', username):
    print "use five numbers, letters, or underscores\n"
```

As you can see, the anchors force the match to start at the start of the string and end at the end of the string -- thus matching only on the whole string.

Built-in Character Classes: spaces

The **\s** character class matches on a space, a newline (**\n**) or a tab (**\t**).

This program searches for a space anywhere in the string and if it finds it, the match is successful - which means the input isn't successful:

```
new_password = raw_input()
if re.search(r'\s', new_password):
    print "password must not contain spaces"
```

Note in particular that the regex pattern **\s** is not anchored anywhere. So the regex will match if a space occurs anywhere in the string.

You may also reflect that we treat spaces pretty roughly - always stripping them off. They're always causing problems! And they're invisible, too, and still get in the way. What a nuisance.

Inverse Character Classes

Each built-in has a corresponding "uppercased" character class that matches on *anything but* the original class.

Not a digit: **\D**
\D matches on any character that is not a digit, including letters, underscores, punctuation, etc. This program checks for a non-digit in the user's account number:

```
account_number = raw_input()
if re.search(r'\D', account_number):
    print "account number must be all digits!"
```

Not a word character: **\W**

\W matches on any character that is not a word character, including punctuation and spaces.

```
account_number = raw_input()
if re.search(r'\W', account_number):
    print "account number must be only letters, numbers, and underscores"
```

Not a space character: **\S**

\S matches on any character that is not a space (i.e. letters, numbers, special characters, etc.)

These two regexes check for a non-space at the start and end of the string:

```
sentence = raw_input()
if re.search(r'^\S', sentence) and re.search(r'\S$', sentence):
    print "the sentence does not begin or end with a space, tab or newline."
```

Custom Character Classes

The collection of characters included in a character class can be custom-defined.

Consider this table of character classes and the list of characters they match on:

class designation	members
\d	[0123456789]
\w	[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOOPQRSTUVWXYZ0123456789_] or [a-zA-Z0-9_]
\s	[\t\n]

In fact, the bracketed ranges can be used to create our own character classes. We simply place members of the class within the brackets and use it in the same way we might use **\d** or the others. A custom class can contain a *range* of characters. This example looks for letters only (there is no built-in class for letters):

```
import re
input = raw_input("please enter a username, starting with a letter:  ")
if not re.search(r'^[a-zA-Z]', input):
    exit("invalid user name entered")
```

This custom class [.,;:?!] matches on any one of these punctuation characters, and this example identifies single punctuation characters and removes them:

```
import re
text_line = 'Will I?  I will.  Today, tomorrow; yesterday and before that.'
for word in text_line.split():
    while re.search(r'[.,;:?! -]$', word):
        word = word[:-1]
    print word
```

Negative Custom Character Classes

Any customer character class can be "inversed" when preceded by a carat character.

Like **\S** for **\s**, the inverse character class matches on anything *not* in the list. It is designated with a carrot just

inside the open bracket:

```
import re
for text_line in open('unknown_text.txt'):
    for word in text_line.split():
        while re.search(r'[^a-zA-Z]$', word):
            word = word[:-1]
        print word
```

It would be easy to confuse the carrot at the start of a string with the carrot at the start of a custom character class -- just keep in mind that one appears at the very start of the string, and the other at the start of the bracketed list.

The Wildcard (.)

The wildcard matches on any character that is not a newline.

```
import re
username = raw_input()
if not re.match(r'^.....$', username):   # five dots here
    print "you can use any characters except newline, but there must \
    be five of them.\n"
```

We might surmise this is because we are often working with line-oriented input, with pesky newlines at the end of every line. Not matching on them means we never have to worry about stripping or watching out for newlines.)

Quantifiers: specifies how many to look for

The quantifier specifies how many of the immediately preceding character may match by the pattern.

We can say three digits (\d{3}), between 1 and 3 word characters (\w{1,3}), one or more letters [a-zA-Z]+, zero or more spaces (\s*), one or more x's (x+). Anything that matches on a character can be quantified.

+	1 or more
*	0 or more
?	0 or 1
{3,10}	between 3 and 10

In this example directory listing, we are interested only in files with the pattern **config**_ followed by an integer of any size. We know that there could be a config_1.txt, a config_12.txt, or a config_120.txt. So, we simply specify "one or more digits":

```
import re
filenames = ['config_1.txt', 'config_10.txt', 'notthis.txt', '.', '..']
wanted_files = []
for file in filenames:
    if re.search(r'^config_\d+\.txt$', file):
        wanted_files.append(file)
```

Here, we validate user input to make sure it matches the pattern for valid NYU ID. The pattern for an NYU Net ID is: two or three letters followed by one or more numbers:

```
import re
input = raw_input("please enter your net id:  ")
if not re.search(r'^[A-Za-z]{2,3}\d+$', input):
    print "that is not valid NYU Net ID!"
```

A *simple* email address is one or more word characters followed by an @ sign, followed by a period, followed by 2-4 letters:

```
import re
email_address = raw_input()
if re.search(r'^\w+@\w+\.[A-Za-z]{2,}$', email_address):
   w print "email address validated"
```

Of course email addresses can be more complicated than this - but for this exercise it works well.

re.search(), re.compile() and the compile object

re.search() is the one-step method we've been using to test matching. Actually, regex matching is done in two steps: *compiling* and *searching*. **re.search()** conveniently puts the two together.

In some cases, a pattern should be compiled first before matching begins. This would be useful if the pattern is to be matched on a great number of strings, as in this weblog example:

```
import re
access_log = '/home1/d/dbb212/public_html/python/examples/access_log'
weblog = open(access_log)
patternobj = re.compile(r'edg205')
for line in weblog.readlines():
  if patternobj.search(line):
    print line,
weblog.close()
```

The **pattern object** is returned from **re.compile**, and can then be called with **search**. Here we're calling **search** repeatedly, so it is likely more efficient to compile once and then search with the compiled object.

Grouping for Alternates: Vertical Bar

We can group several characters together with parentheses. The parentheses do not affect the match, but they do designate a part of the matched string to be handled later. We do this to allow for alternate matches, for quantifying a portion of the pattern, or to extract text.

Inside a group, the vertical bar can indicate allowable matches. In this example, a string will match on any of these words, and because of the anchors will not allow any other characters:

```
r'^(y|yes|yeah|yep|yup|yu-huh)$'   # matches any of these words
```

A group may indicate alternating choices within a larger pattern:

```
r'(John|John D.)\s+Rockefeller'
                          # matches John Rockefeller or John D. Rockefeller
```

Grouping for Quantifying

Another reason to group would be to quantify a sequence of the pattern. For example, we could search for the "John Rockefeller or John D. Rockefeller" pattern by making the 'D.' optional:

```
r'John\s+(D\.\s+)?Rockefeller'
```

Grouping for Extraction: Memory Variables

We use the **group()** method of the **match** object to extract the text that matched the group:

```
string1 = "Find the nyu id, like dbb212, in this sentence"
matchobj = re.search(r'([a-z]{2,3}\d+)', string1)
id = matchobj.group(1)                          # id is 'dbb212'
```

Here's an example, using our log file. What if we wanted to capture the last two numbers (the status code and the number of bytes served), and place the values into structures?

```
log_lines = [
'66.108.19.165 - - [09/Jun/2003:19:56:33 -0400] "GET /~jjk265/cd.jpg HTTP/1.1" 200 175449',
'216.39.48.10 - - [09/Jun/2003:19:57:00 -0400] "GET /~rba203/about.html HTTP/1.1" 200 1566',
'216.39.48.10 - - [09/Jun/2003:19:57:16 -0400] "GET /~dd595/frame.htm HTTP/1.1" 400 1144'
]

import re
bytes_sum = 0
for line in log_lines:
  matchobj = re.search(r'(\d+) (\d+)$', line) # last two numbers in line
  status_code = matchobj.group(1)
  bytes = matchobj.group(2)
  bytes_sum += int(bytes)                     # sum the bytes
```

groups()

If you wish to grab all the matches into a tuple rather than call them by number, use **groups()**. You can then read variables from the tuple, or assign **groups()** to named variables, as in the last line below:

```
import re

name = "Richard M. Nixon"
matchobj = re.search(r'(\w+)\s+(\w)\.\s+(\w+)', name)
name_tuple = matchobj.groups()
print name_tuple    # ('Richard', 'M', 'Nixon')

(first, middle, last) = matchobj.groups()
print "%s, %s, %s" % ( first, middle, last )
```

findall() for multiple matches

findall() with a groupless pattern

Usually **re** tries to a match a pattern once -- and after it finds the first match, it quits searching. But we may want to find as many matches as we can -- and return the entire set of matches in a list. **findall()** lets us do that:

```
text = "There are seven words in this sentence";
words = re.findall(r'\w+', text)
print words  # ['There', 'are', 'seven', 'words', 'in', 'this', 'sentence']
```

This program prints each of the words on a separate line. The pattern **\b\w+\b** is applied again and again, each time to the text remaining after the last match. This pattern could be used as a word counting algorithm (we would count the elements in **words**), except for words with punctuation.

findall() with groups

When a match pattern contains more than one grouping, **findall** returns multiple tuples:

```
text = "High: 33, low: 17"
temp_tuples = re.findall(r'(\w+):\s+(\d+)', text)
print temp_tuples                       # [('High', '33'), ('low', '17')]
```

sub() for substitutions

Regular expressions are used for matching so that we may inspect text. But they can also be used for substitutions, meaning that they have the power to modify text as well.

This example replaces Microsoft '\r\n' line ending codes with Unix '\n'.

```
text = re.sub(r'\r\n', '\n', text)
```

Here's another simple example:

```
string = "My name is David"
string = re.sub('David', 'John', string)

print string                            # 'My name is John'
```

matching on multi-line files

This example opens and reads a web page (which we might have retrieved with a module like **urlopen**), then looks to see if the word "advisory" appears in the text. If it does, it prints the page:

```
file = open('weather-ny.html')
text = file.read()
if re.search(r'advisory', text, re.I):
  print "weather advisory:  ", text
```

re.MULTILINE: ^ and $ can match at start or end of line

We have been working with text files primarily in a line-oriented (or, in database terminology, *record-oriented* way, and regexes are no exception - most file data is oriented in this way. However, it can be useful to dispense with looping and use regexes to match within an entire file - read into a string variable with **read()**.

In this example, we surely can use a loop and split() to get the info we want. But with a regex we can grab it straight from the file in one line:

```
# passwd file:
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false

# python script:
import re
passwd_text = open('/etc/passwd').read()
mobj =  re.search(r'^root:[^:]+:[^:]+:[^:]:([^:]+):([^:]+)', passwd_text, re.MULTILINE)
if mobj:
    info = mobj.groups()
    print "root:  Name %s, Home Dir %s" % (info[0], info[1])
```

We can even use findall to extract all the information rfrom a file - keep in mind, this is still being done in two lines:

```
import re
passwd_text = open('/etc/passwd').read()
lot = re.findall(r'^(\w+):[^:]+:[^:]+:[^:]+:([^:]+)', passwd_text, re.MULTILINE)

mydict = dict(lot)

print mydict
```

re.DOTALL -- allow the wildcard (.) to match on newlines

Normally, the wildcard doesn't match on newlines. When working with whole files, we may want to grab text that spans multiple lines, using a wildcard.

```
# search file sample.txt
some text we don't want
==start text==
this is some text that we do want.
the extracted text should continue,
including just about any character,
until we get to
==end text==
other text we don't want

# python script:
import re
text = open('sample.txt').read()
matchobj = re.search(r'==start text==(.+)==end text==', text, re.DOTALL)
print matchobj.group(1)
```

flags: re.IGNORECASE

We can modify our matches with qualifiers called *flags*. The **re.IGNORECASE** flag will match any letters, whether upper or lowercase. In this example, extensions may be upper or lowercase - this file matcher doesn't care!

```
import re
dirlist = ('thisfile.jpg', 'thatfile.txt', 'otherfile.mpg', 'myfile.TXT')
for file in dirlist:
  if re.search(r'\.txt$', file, re.IGNORECASE):   #'.txt' or '.TXT'
    print file
```

The flag is passed as the third argument to **search**, and can also be passed to other **re** search methods.

Web Clients

Basic HTTP

Headers, Cookie Headers and Response Codes

HTTP (HyperText Transport Protocol) is the protocol for sending and receiving messages from a browser (the *client*) to a web server (the *server*) and back again. We call the browser's message the *request* and the server's response the *response*.

request
A request consists of two parts: the URL (along with any *parameters*) and (optionally) any *content*.

A request is generally one of two *methods*: **GET** (for retrieving data) or **POST** (for sending data, like form input). (Note in this context **method** is unrelated to a Python method.)

HTTP Headers are meta information sent along with a request. This may include session (cookie) information.

response
The **content** of a response is the HTML, text or other data (can be binary data, or anything else a browser can send or receive).

The response may also include *headers*. These include the response code, the size of the response, and may also contain cookies.

The *response code* of a response indicates whether the request was handled without error (**200**), whether there was a server error (**500**), etc.

Python on the Client side: the requests module

requests is highly popular for web client functionality

A *web client* is any program that can send HTTP requests to a web server. Your browser is a web client.

The **requests** module lets us issue HTTP requests in the same way a browser would. We can therefore have our Python program behave like a browser, i.e. visit web pages and download data.

Requesting a web page through requests.get()

A page is usually requested as a GET request

```
import requests

response = requests.get('http://www.nytimes.com')

page_text =   response.text
status_code = response.status_code

page_text = page_text.encode('utf-8')      # if necessary

print 'status code:  {}'.format(status_code)
print '======================== page text ========================'
print page_text
```

Decoding a JSON response

API calls are also made over HTTP, and if the call returns JSON, this can easily be decoded through **requests**.

```
import requests

response = requests.get('http://api.wunderground.com/api/d2e101aa48faa661/conditions/q/CA/San_Francisco.jso
conditions_json = response.json()

print conditions_json["current_observation"]["temp_f"]        # 84.5 (accessing data within the JSON)
```

Requesting a web page with parameters

Many web requests include parameters specifying what content or action is desired. Here's a link to my homework application:

```
http://homework-davidbpython.rhcloud.com/route_view?assignment_id=1.1&student_id=bill_hanson
```

The parameters here are **assignment_id** (value **1.1**) and **student_id** (value **bill_hanson**)

To pass parameters, we simply include a dict keyed to **params**.

```
param_dict = {'assignment_id': '1.1', 'student_id': 'bill_hanson'}
response = requests.get('http://homework-davidbpython.rhcloud.com/route_view', params=param_dict)
```

posting data to a web address, with parameter input

A *form submission* is usually sent as a POST request and includes parameter data. Here is a sample form:

```
<FORM ACTION="http://www.mywebsite.com/user" METHOD="POST">
  <INPUT NAME="firstname"><BR>
  <INPUT NAME="lastname"><BR>
  <INPUT NAME="password" TYPE="password">
  <INPUT TYPE="submit">
</FORM>
```

This form produces key/value data in the body of the request.

We can replicate this kind of request by using **request.post()** and passing a dict of param keys and values:

```
userdata = {"firstname": "John", "lastname": "Doe", "password": "jdoe123"}

resp = requests.post('http://www.mywebsite.com/user', params=userdata)
```

Some other features of **requests**:

* International Domains and URLs
* Keep-Alive & Connection Pooling
* Sessions with Cookie Persistence
* Browser-style SSL Verification
* Basic/Digest Authentication
* Elegant Key/Value Cookies
* Automatic Decompression
* Unicode Response Bodies
* Multipart File Uploads
* Connection Timeouts

Beautiful Soup (bs4)

Beautiful Soup parses XML or HTML documents, making text and attribute extraction a snap.

Here we are passing the text of a web page (obtained by **requests**) to the BS parser:

```
from bs4 import BeautifulSoup
import requests

response = requests.get('http://www.nytimes.com')

soup = BeautifulSoup(response.text, 'html.parser')

# show HTML in "pretty" form
print soup.prettify()

# show all plain text in a page
print soup.get_text()
```

The result is a BeautifulSoup object which we can use to search for tags and data.

For the following examples, let's use the HTML provided on the Beatiful Soup Quick Start page:

```
<!doctype html>
<html>
  <head>
    <title>The Dormouse's story</title>
  </head>
  <body>
    <p class="story_title"><b>The Dormouse's story</b></p>

    <p class="story">Once upon a time there were three little sisters; and their names were
        <a href="http://example.com/elsie" class="sister eldest" id="link1">Elsie</a>,
        <a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
        <a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
    and they lived at the bottom of a well.</p>

    <p class="story">They were happy, and eventually died.  The End.</p>
  </body>
</html>
```

bs4: finding a tag with attributes, find() and find_all()

Finding the first tag by name using *soup.attribute*

The **BeautifulSoup** object's attributes can be used to search for a tag. The first tag with a name will be returned.

```
# first (and only) <title> tag
print soup.title              # <title>The Dormouse's story</title>

# first (of several) <p> tags
print soup.p                  # <p class="title"><b>The Dormouse's story</b></p>
```

Attributes can be chained to drill down to a particular tag:

```
print soup.body.p.b           # <b>The Dormouse's story</b>
```

However keep in mind these represent the first of each tag listed.

Finding the first tag by name: find()

find() works similarly to an attribute, but filters can be applied (discussed shortly).

```
print soup.find('a')          <a class="sister eldest" href="http://example.com/elsie" id="link1">Elsie</a>
```

Finding all tags by name: find_all()

findall() retrieves a list of all tags with a particular name.

```
tags = soup.find_all('a')
```

bs4: the Tag object

Tags' attributes and contents can be read; they can also be queried for tags and text within

```
body_text = """
    <BODY class="someclass otherclass">
        <H1 id='mytitle'<This is a headings</H1>
        <A href="mysite.com"<This is a link</A>
    </BODY>
"""
```

An HTML tag has four types of data:
1. The tag's name ('BODY' 'H1' or 'A')
2. The tag's attributes (<BODY class=, H1 id= or <A href=)
3. The tag's text ('This is a header' or 'This is a link')
4. The tag's contents (i.e., tags within it -- for <BODY>, the <H1> and <A> tags)

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(body_text, 'html.parser')

h1 = soup.body.h1        # h1 is a Tag object
print h1.name            # u'h1'
print h1.get('id')       # u'mytitle'
print h1.attrs           # {u'id': u'mytitle'}
print h1.text            # u'This is a heading'

body = soup.body         # body is a Tag object
print body.name          # u'body'
print body.get('class')  # ['someclass', 'otherclass']
print body.attrs         # {'class': ['someclass', 'otherclass']}
print body.text          # u'\nThis is a heading\nThis is a link\n'
```

A tag's child tags can be searched the same as the BeautifulSoup object

```
body = soup.body         # find the <body> tag in this document

atag = body.find('a')    # find first <a> tag in this <body> tag
```

bs4: finding a tag using varying criteria

Tag criteria can focus on a tag's name, its attributes, or text within the tag.

SEARCHING NAME, ATTRIBUTE OR TEXT

Finding a tag by name

Links in a page are marked with the <A> tag (usually seen as <A HREF="">). This call pulls out all links from a page:

```
link_tags = soup.find_all('a')
```

Finding a tag by tag attribute and/or name and tag attribute

```
# all <a> tags with an 'id' attribute of link1
link1_a_tags = soup.find_all('a', id="link1")

# all tags (of any name) with an 'id' attribute of link1
link1_tags = soup.find_all(id="link1")
```

"multi-value" tag attribute

CSS allows multiple values in an attribute:

```
<a href="http://example.com/elsie" class="sister eldest" id="link1">Elsie</a>
```

If we'd like to find a tag through this value, we pass a list:

```
link1_elsie_tag = soup.find(class_=['sister', 'eldest'])
```

Finding a tag by string within the tag's text

All <a> tags containing text 'Dormouse'

```
elsie_tags = soup.find_all('a', text='Dormouse')
```

FILTER TYPES: STRING, LIST, REGEXP, FUNCTION

string: filter on the tag's name

```
tags = soup.find_all('a')          # return a list of all <a> tags
```

list: filter on tag names

```
tags = soup.find_all(['a', 'b'])   # return a list of all <a> or <b> tags
```

regexp: filter on pattern match against name

```
import re
tags = soup.find_all(re.compile('^b'))      # a list of all tags whose names start with 'b'
```

re.compile() produces a pattern object that is applied to tag names using **re.match()**

function: filter if function returns True

```
soup.find_all(lambda tag: tag.name == 'a' and 'mysite.com' in tag.get('href'))
```

Sending email with smtplib

Sending mail is simple when you have an SMTP server running and available on your host computer. Python's **smtplib** module makes this easy:

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

# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Create a text/plain message formatted for email
msg = MIMEText('Hello, email.')

from_address = 'dbb212@nyu.edu'
to_address = 'david.beddoe@gmail.com'
subject = 'Test message from a Python script'

msg['Subject'] = subject
msg['From'] =    from_address
msg['To'] =      to_address

s = smtplib.SMTP('localhost')
s.sendmail(from_address, [to_address], msg.as_string())
s.quit()
```

Sidebar: the urllib2 module

A Python program can also take the place of a browser, requesting and downloading HTML pages and other files. Your Python program can work like a web spider (for example visiting every page on a website looking for particular data or compiling data from the site), can visit a page repeatedly to see if it has changed, 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 HTML, Javascript, and possibly other kinds of data. It is not designed for human reading, but it does contain the info that we would need to retrieve from a page. To do so programmatically (instead of visually), we need to engage in *web scraping*. Most simply, this is done with regexes (coming up).

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>
```

pandas and numpy

pandas and numpy: Introduction

pandas is a Python module used for manipulation and analysis of tabular data.

* Excel-like numeric calculations, particularly column-wise and row-wise calculations (*vectorization*)

* SQL-like merging, grouping and aggregating

* emphasis on aligning data from multiple sources and cleaning and normalizing missing data

* ability to read and write to CSV, XML, Excel, database queries, etc.

numpy is a data analysis library that underlies pandas. We sometimes make direct calls to numpy - some of its variables (such as **np.nan**), variable-generating functions (such as **np.arange**) and some processing functions.

pandas Reference

Various documentation sources will be necessary as the pandas library has many features.

cheat sheet (Treehouse)

```
https://s3.amazonaws.com/assets.datacamp.com/blog_assets/PandasPythonForDataScience.pdf (https://s3.amazona
```

docs on any pandas function or DataFrame method

```
import pandas as pd

help(pd.read_csv)       # help on the read_csv function of pandas

df = pd.DataFrame()     # initialize a DataFrame
help(df.join)           # help on the join() method of a DataFrame
```

pandas official documentation

```
http://pandas.pydata.org/pandas-docs/stable (http://pandas.pydata.org/pandas-docs/stable)

http://pandas.pydata.org/pandas-docs/version/0.19.0/pandas.pdf (http://pandas.pydata.org/pandas-docs/versio
```

pandas cookbook

```
http://pandas.pydata.org/pandas-docs/stable/cookbook.html (http://pandas.pydata.org/pandas-docs/stable/cook
```

pandas textbook "Python for Data Analysis" by Wes McKinney

```
http://www3.canisius.edu/~yany/python/Python4DataAnalysis.pdf (http://www3.canisius.edu/~yany/python/Python
```

(If the above link goes stale, simply search **Python for Data Analysis pdf**.)

Please keep in mind that pandas is in active developemnt (latest version: 0.19.0)

pandas objects

The *DataFrame* is the primary object in pandas; a DataFrame column or row can be isolated as a *Series* object; columns and rows are enumerated with the *Index* object.

DataFrame:
* is like an Excel spreadsheet - rows, columns, and row and column labels
* is like a "dict of dicts" in that it holds *column*-indexed Series
* offers database-like and excel-like manipulations (merge, groupby, etc.)

```
import pandas as pd
df = pd.DataFrame({'a': [1, 2, 3], 'b': [10, 20, 30], 'c': [100, 200, 300]},
                  index=['r1', 'r2', 'r3'])

print df
            #      a   b    c
            # r1   1  10  100
            # r2   2  20  200
            # r3   3  30  300

print df['c']['r2']    # 200
```

Series
* a "dictionary-like list" -- ordered values by associates them with an *index*
* has a *dtype* attribute that holds its objects' common type

```
# read a column as a Series
bcol = df['b']
print bcol
            # r1    10
            # r2    20
            # r3    30
            # Name: b, dtype: int64


# read a row as a Series (using .loc)
oneidx = df.loc['r2']
print oneidx
            # a      2
            # b     20
            # c    200
            # Name: r2, dtype: int64
```

Index
* an object that provides indexing for both the Series (its item index) and the DataFrame (its column or row index).

```
columns = df.columns   # Index([u'a',  u'b',  u'c'],  dtype='object')
idx = df.index         # Index([u'r1', u'r2', u'r3'], dtype='object')
```

Reading pandas DataFrames from Data Sources

Pandas can read in a Dataframe object from CSV, JSON, Excel and XML formats.

CSV

```
# read from file
df = pd.read_csv('quarterly_revenue_2017Q4.csv')


# write to file
wfh = open('output.csv', 'w')
df.to_csv(wfh, na_rep='NULL')


# reading from Fama-French file (the abbreviated file, no header)
# sep= indicates the delimiter on which to split() the fields
# names= indicates the column heads
df = pd.read_csv('FF_abbreviated.txt', sep='\s+',
                                       names=['date', 'MktRF', 'SMB', 'HML', 'RF'])


# reading from Fama-French non-abbreviated (the main file including headers and footers)
# skiprows=5:  start reading 5 rows down
df = pd.read_csv('F-F_Research_Data_Factors_daily.txt', skiprows=5, sep='\s+',
                                                        names=['date', 'MktRF', 'SMB', 'HML', 'RF'])
```

Excel

```
# reading from excel file (2 steps)
xls_file = pd.ExcelFile('data.xls')    # produce a file 'reader' object
df = xls_file.parse('Sheet1')          # parse a selected sheet to a DataFrames

# write to excel
df.to_excel('data2.xls', sheet_name='Sheet1')
```

JSON

```
# sample df for demo purposes
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )



# write dataframe to JSON
pd.json.dump(df, open('df.json', 'w'))

mydict = pd.json.load(open('df.json'))
new_df = pd.DataFrame(mydict)
```

From Clipboard

This option is excellent for cutting and pasting data from websites

```
df = pd.read_clipboard(skiprows=5, sep='\s+',
                       names=['date', 'MktRF', 'SMB', 'HML', 'RF'])
```

The DataFrame: Initializing and Slicing

THE *dataframe* is the pandas workhorse structure. It is a 2-dimensional structure with columns and rows (i.e., like a spreadsheet).

Initializing

```
import pandas as pd
import numpy as np

# initialize a new, empty DataFrame
df = pd.DataFrame()


# init with list of lists
df = pd.DataFrame( [  [ 'a', 'b', 'c' ],
                      [  1,   2,   3  ],
                      [  10,  20, 30  ]   ]  )

print df

            #      0   1   2
            #  0   a   b   c
            #  1   1   2   3
            #  2  10  20  30


# init with dict of lists and index
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'],
                    'd': [100, 200, 300, 400] },
                    index=['r1', 'r2', 'r3', 'r4'] )

print df

            #     a    b  c    d
            # r1  1  1.0  a  100
            # r2  2  1.5  b  200
            # r3  3  2.0  c  300
            # r4  4  2.5  d  400
```

Accessing columns or rows (Series objects)

```
cola = df['a']       # Series with [1, 2, 3, 4] and index ['r1', 'r2', 'r3', 'r4']
cola = df.a          # same

row2 = df.loc['r2']  # Series [2, 1.5, 'b', 200] and index ['a', 'b', 'c', 'd']
```

Accessing DataFrame slices (DataFrame objects)

```
dfslice1 = df[ ['a', 'b', 'c'] ]  # slice out 1st 3 columns
dfslice2 = df['r1': 'r2']         # slice out 1st 2 rows (label indexing upper bound is inclusive!)
dfslice3 = df[0:2]                # slice out same 1st 2 rows
```

The Series: subscripting and slicing

A *Series* is pandas object representing a column or row in a DataFrame.

A Series can be initialized on its own and made part of a DataFrame

```
s1 = pd.Series([1, 2, 3, 4])
s2 = pd.Series([1.0, 1.5, 2.0, 2.5])

df = pd.DataFrame({'a': s1, 'b': s2})
```

A DataFrame can be seen as a list of Series objects.

```
df = pd.DataFrame( { 'a': [1, 2, 3, 4],
                     'b': [1.0, 1.5, 2.0, 2.5],
                     'c': ['a', 'b', 'c', 'd']  },
                     index=['r1', 'r2', 'r3', 'r4'] )

print df

                  #     a    b  c
                  # r1  1  1.0  a
                  # r2  2  1.5  b
                  # r3  3  2.0  c
                  # r4  4  2.5  d
```

DataFrame subscript accesses a column

```
s1 = df['a']      # Series object (column)

                  # r1    1
                  # r2    2
                  # r3    3
                  # r4    4
                  # Name: a, dtype: int64
```

(Series name is the column head, index are the row labels)

DataFrame **.loc** indexer accesses the rows

```
r1 = df.loc['r2'] # Series object (row)

                  # a      2
                  # b    1.5
                  # c      b
                  # Name: r2, dtype: object
```

(Series name is the row label, index are the column heads)

Series items can be accessed through the index labels, or by index position.

```
print r1['a']        # 2
print r1.a           # 2
print r1[0]          # 2
```

Working with Series Objects as part of a DataFrame

We usually work with the DataFrame subscript directly, resulting in a double-subscript

Initialize a DataFrame

```
import pandas as pd

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df

                  #     a    b  c
                  # r1  1  1.0  a
                  # r2  2  1.5  b
                  # r3  3  2.0  c
                  # r4  4  2.5  d
```

Access a Series (DataFrame column)

```
print df['b']

                  # r1    1.0
                  # r2    1.5
                  # r3    2.0
                  # r4    2.5
                  # Name: b, dtype: float64
```

Access a single value in a DataFrame

```
# by Series index
print df['b'][0]            # 1.0

# by Series row label
print df['b']['r2']         # 1.5
```

Access a slice in a DataFrame

```
# by Series indices
print df['c'][1:3]

                  # r2    b
                  # r3    c
                  # Name: c, dtype: object


# by Series row labels
print df['c'][['r2', 'r3', 'r4']]

                  # r2    b
                  # r3    c
                  # r4    d
                  # Name: c, dtype: object
```

Note this last slice: we specify a *list of labels*, then pass that list into **df['c']** Series' subscript (square brackets) -- thus the nested square bracket syntax.

Create a DataFrame as a portion of another DataFrame

Oftentimes we want to eliminate one or more columns from our DataFrame. We do this by slicing Series out of the DataFrame, to produce a new DataFrame:

```
dfi = pd.DataFrame({'c1': [0,   1,  2, 3,   4],
                    'c2': [5,   6,  7, 8,   9],
                    'c3': [10, 11, 12, 13, 14],
                    'c4': [15, 16, 17, 18, 19],
                    'c5': [20, 21, 22, 23, 24],
                    'c6': [25, 26, 27, 28, 29] },
           index = ['r1', 'r2', 'r3', 'r4', 'r5'])

print dfi
                  #     c1  c2  c3  c4  c5  c6
                  # r1   0   5  10  15  20  25
                  # r2   1   6  11  16  21  26
                  # r3   2   7  12  17  22  27
                  # r4   3   8  13  18  23  28
                  # r5   4   9  14  19  24  29


print dfi[['c1', 'c3']]
                  #     c1  c3
                  # r1   0  10
                  # r2   1  11
                  # r3   2  12
                  # r4   3  13
                  # r5   4  14
```

```
print dfi.ix[['r1', 'r3', 'r5']]

                  #     c1  c2  c3  c4  c5  c6
                  # r1   0   5  10  15  20  25
                  # r3   2   7  12  17  22  27
                  # r5   4   9  14  19  24  29


print dfi.ix[['r1':'r3']]

                  #     c1  c2  c3  c4  c5  c6
                  # r1   0   5  10  15  20  25
                  # r2   1   6  11  16  21  26
                  # r3   2   7  12  17  22  27
```

2-dimensional slicing

```
print dfi[['c1', 'c2']]['r1': 'r2']

                  #     c1  c2
                  # r1   0   5
                  # r2   1   6
```

Slicing columns by index

```
dfslice = dfi.iloc[:, [0, 1, 2, 3]]   # 1st 4 columns of data frame
print dfslice

                  #     c1  c2  c3  c4
                  # r1   0   5  10  15
                  # r2   1   6  11  16
                  # r3   2   7  12  17
                  # r4   3   8  13  18
                  # r5   4   9  14  19
```

The Index

An Index object is used to specify a DataFrame's columns or index, or a Series' index.

Columns and Indices

A DataFrame makes use of two Index objects: one to represent the columns, and one to represent the rows.

```
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'],
                    'd': [100, 200, 300, 400] },
                    index=['r1', 'r2', 'r3', 'r4'] )
```

Columns or index labels can be reset using the dataframe's **rename()** method.

```
df = df.rename(columns={'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'},
               index={'r1': 'R1', 'r2': 'R2', 'r3': 'R3', 'r4': 'R4'})
```

The columns or index can also be set directly using the dataframe's attributes (although this is more prone to error).

```
df.columns = ['A', 'B', 'C', 'D']

# reset indices to integer starting with 0
df.reset_index()

# set name for index and columns
df.index.name = 'year'
df.columns.name = 'state'

# reindex ordering by index:
df = df.reindex(reversed(df.index))

df.reindex(columns=reversed(df.columns))
```

Dataframe and Series dtypes

Unlike core Python containers (but similar to a database table), pandas cares about object type. Wherever possible, pandas will assign a type to a column Series and attempt to maintain the type's integrity.

This is done for the same reason it is done with database tables: speed and space efficiency.

In the below DataFrame, Python "sniffs out" the type of a column Series. If all values match, pandas will set the type.

```
import pandas as pd

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df
                  #     a    b  c
                  # r1  1  1.0  a
                  # r2  2  1.5  b
                  # r3  3  2.0  c
                  # r4  4  2.5  d


print df.dtypes

                  # a      int64       # note special pandas types int64 and float64
                  # b    float64
                  # c     object       # 'object' is general-purpose type, covers strings or mixed-type col
                  # dtype: object
```

You can use the regular integer index to *set* element values in an existing Series. However, the new element value must be the same type as that defined in the Series.

```
df['b'][0] = 'hello'
ValueError: could not convert string to float: hello
```

Note that we never told pandas to store these values as floats. But since they are all floats, pandas decided to set the type.

We can change a dtype for a Series:

```
df.a = df.a.astype('object')         # or df['a'] = df['a'].astype('object')

df['a'][0] = 'hello'
```

Vectorized Operations

Operations to Series are *vectorized*, meaning they are propagated across the Series.

```
import pandas as pd

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df

                  #     a    b  c
                  # r1  1  1.0  a
                  # r2  2  1.5  b
                  # r3  3  2.0  c
                  # r4  4  2.5  d

# 'single value':  assign the same value to all cells in a column Series
df.a = 0
print df

                  #     a    b  c
                  # r1  0  1.0  a
                  # r2  0  1.5  b
                  # r3  0  2.0  c
                  # r4  0  2.5  d


# 'calculation':  compute a new value for all cells in a column Series
df.b = df.b * 2

print df

                  #     a    b  c
                  # r1  0  2.0  a
                  # r2  0  3.0  b
                  # r3  0  4.0  c
                  # r4  0  5.0  d
```

Adding New Columns with Vectorized Values

We can also add a new column to the Dataframe based on values or computations:

```
df = pd.DataFrame( {'a': [0, 0, 0, 0],
                    'b': [2.0, 3.0, 4.0, 5.0],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )


df['d'] = 3.14           # new column, each field set to same value

print df

                  #     a    b  c  d
                  # r1  1  2.0  a  3.14
                  # r2  2  3.0  b  3.14
                  # r3  3  4.0  c  3.14
                  # r4  4  5.0  d  3.14


df['e'] = df.a + df.b    # vectorized computation to new column

print df

                  #     a    b  c     d  e
                  # r1  1  2.0  a  3.14  3.0
                  # r2  2  3.0  b  3.14  5.0
                  # r3  3  4.0  c  3.14  7.0
                  # r4  4  5.0  d  3.14  9.0
```

mask: conditional vectorization

Oftentimes we want to broadcast a computation conditionally, i.e. only for some elements based on their value. To do this, we establish a *mask*, which goes into subscript-like square brackets:

```
import pandas as pd

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [-1.0, -1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )
print df

                    #     a   b  c
                    # r1  1 -20  a
                    # r2  2 -10  b
                    # r3  3  10  c
                    # r4  4  20  d

df['b'][ df['b'] < 0 ] = 0    # for each 'b' value that is < 0, set to 0

print df

                    #     a   b  c
                    # r1  1   0  a
                    # r2  2   0  b
                    # r3  3  10  c
                    # r4  4  20  d
```

The mask by itself returns a *boolean Series*. This mask can of course be assigned to a name and used by name:

```
mask = df['a'] > 2
print mask          # printing just for illustration

                    # r1    False
                    # r2    False
                    # r3    True
                    # r4    True
                    # Name: a, dtype: bool
```

Of course a vector operation can be filtered with a mask:

```
mask = df['a'] > 2
df['a'][ mask ] = df['b'] * 2

print df
                  #      a   b  c
                  # r1   1   0  a      # 'a' not > 3:  no effect
                  # r2   2   0  b      # 'a' not > 3:  no effect
                  # r3  20  10  c      # 'a' > 3, so now a == b * 2
                  # r4  40  20  d      # 'a' > 3, so now a == b * 2
```

You can think of this mask as being placed over the Series in question ('a'), using the criteria **< 3** to determine whether the element is visible.

negating a mask

a tilde (**~**) in front of a mask creates its inverse:

```
mask = df['a'] > 2
df['a'][ ~mask ] = 0

print df
                  #      a   b  c
                  # r1   0   0  a
                  # r2   0   0  b
                  # r3  20  10  c
                  # r4  40  20  d
```

Series.apply(): vectorize a function call over a column

Sometimes our computation is more complex than simple math, or we need to apply a function to each element. We can use **apply()**:

```
import pandas as pd

df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )

print df

                  #     a    b  c
                  # r1  1  1.0  a
                  # r2  2  1.5  b
                  # r3  3  2.0  c
                  # r4  4  2.5  d


df['d'] = df.c.apply(str.upper)

print df
                  #     a    b  c  d
                  # r1  1  1.0  a  A
                  # r2  2  1.5  b  B
                  # r3  3  2.0  c  C
                  # r4  4  2.5  d  D
```

Many times, though, we use a custom named function or a lambda, because we want some custom work done:

```
df['e'] = df['a'].apply(lambda x: '$' + str(x * 1000) )


print df
                  #     a    b  c  d      e
                  # r1  1  1.0  a  A  $1000
                  # r2  2  1.5  b  B  $2000
                  # r3  3  2.0  c  C  $3000
                  # r4  4  2.5  d  D  $4000
```

Standard Python operations with DataFrame

DataFrames behave as you might expect

```
df = pd.DataFrame( {'a': [1, 2, 3, 4],
                    'b': [1.0, 1.5, 2.0, 2.5],
                    'c': ['a', 'b', 'c', 'd'] }, index=['r1', 'r2', 'r3', 'r4'] )


print len(df)             # 4

print len(df.columns)     # 3

print max(df['a'])        # 4

print list(df['a'])       # [1, 2, 3, 4]      (column for 'a')

print list(df.ix['r2'])   # [2, 1.5, 'b']     (row for 'r2')

print set(df['a'])        # set([1, 2, 3, 4])


# looping - loops through columns
for colname in df:
    print '{}:  {}'.format(colname, df[colname])

                          # 'a':  pandas.core.series.Series
                          # 'b':  pandas.core.series.Series
                          # 'c':  pandas.core.series.Series


# looping with iterrows -- loops through rows
for index, row in df.iterrows():
    print 'row {}:  {}'.format(index, list(row))


                          # row r1:  [1, 1.0, 'a']
                          # row r2:  [2, 1.5, 'b']
                          # row r3:  [3, 2.0, 'c']
                          # row r4:  [4, 2.5, 'd']
```

Although keep in mind that we generally prefer vectorized operations across columns or rows to looping.

nan and fillna()

If pandas can't insert a value (because indexes are misaligned or for other reasons), it inserts a special value call **NaN** (not a number) in its place.

If we wish to fill the dataframe with an alternate value, we can use **fillna()**, which like all operations vectorizes across the structure:

```
print df
                  #     c1  c2   c3
                  # 0    6  NaN   2
                  # 0    6    1   2
                  # 0  NaN    3   2

df = df.fillna(0)
print df
                  #     c1  c2  c3
                  # 0    6   0   2
                  # 0    6   1   2
                  # 0    0   3   2
```

Concatenating / Appending

concat() can join dataframes either horizontally or vertically.

```
df3 = pd.concat([df, df2])            # horizontal concat
df4 = pd.concat([df, df2], axis=1)    # vertical concat
```

merge

Merge performs a relational database-like **join** on two dataframes. We can join on a particular field and the other fields will align accordingly.

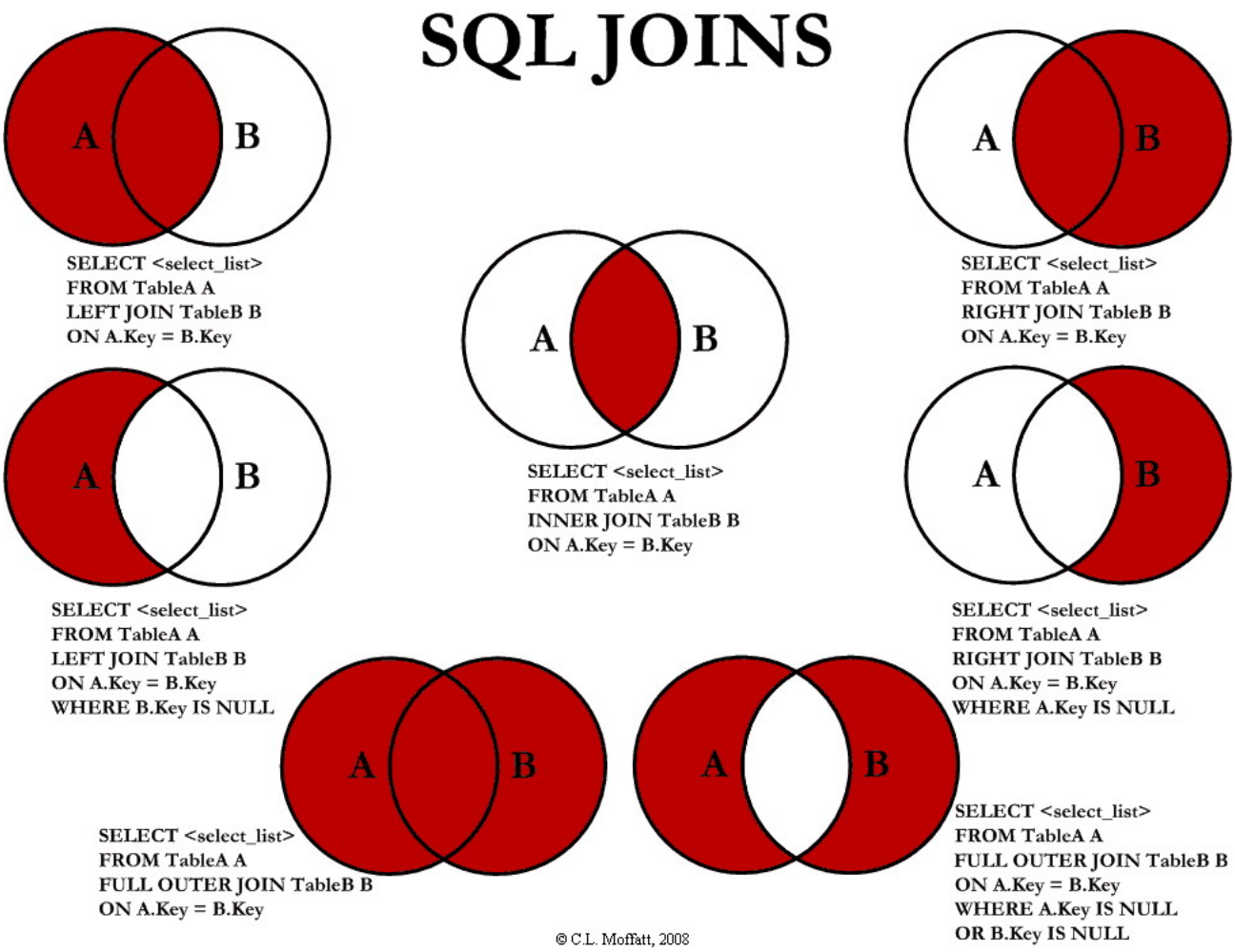
```
print dfi
                  #     c1  c2  c3  c4  c5
                  # r1   0   1   2   3   4
                  # r2   5   6   7   8   9
                  # r3  10  11  12  13  14
                  # r4  15  16  17  18  19
                  # r5  20  21  22  23  24
                  # r6  25  26  27  28  29

print dfi2
                  #     c1  c6  c7
                  # r1   0  41  42
                  # r2   5  51  52
                  # r3  10  61  62
                  # r4  15  71  72
                  # r5  20  81  82
                  # r6  25  91  92

dfi.merge(dfi2, on='c1', how='left')
                  #     c1  c2  c3  c4  c5  c6  c7
                  # r1   0   1   2   3   4  41  42
                  # r2   5   6   7   8   9  51  52
                  # r3  10  11  12  13  14  61  62
                  # r4  15  16  17  18  19  71  72
                  # r5  20  21  22  23  24  81  82
                  # r6  25  26  27  28  29  91  92
```

The merge joins the table. You can choose to join on the index, or one or more columns. **how=** describes the type of join, and the choices are similar to that in relationship databases:

```
Merge method  SQL Join Name   Description
left    LEFT OUTER JOIN   Use keys from left frame only
right   RIGHT OUTER JOIN  Use keys from right frame only
outer   FULL OUTER JOIN   Use union of keys from both frames
inner   INNER JOIN    Use intersection of keys from both frames
```



groupby

A groupby operation performs the same type of operation as the database GROUP BY. Grouping rows of the table by the value in a particular column, you can perform aggregate sums, counts or custom aggregations.

This simple hypothetical table shows client names, regions, revenue values and type of revenue.

```
df = pd.DataFrame( { 'company': ['Alpha', 'ALPHA', 'ALPHA', 'BETA', 'Beta', 'Beta', 'Gamma', 'Gamma', 'Gamm
                     'region':  ['NE', 'NW', 'SW', 'NW', 'SW', 'NE', 'NE', 'SW', 'NW'],
                     'revenue': [10, 9, 2, 15, 8, 2, 16, 3, 9],
                     'revtype': ['retail', 'retail', 'wholesale', 'wholesale', 'wholesale',
                                 'retail', 'wholesale', 'retail', 'retail'] } )

print df

                #   company region  revenue    revtype
                # 0   Alpha     NE       10     retail
                # 1   ALPHA     NW        9     retail
                # 2   ALPHA     SW        2  wholesale
                # 3    BETA     NW       15  wholesale
                # 4    Beta     SW        8  wholesale
                # 5    Beta     NE        2     retail
                # 6   Gamma     NE       16  wholesale
                # 7   Gamma     SW        3     retail
                # 8   Gamma     NW        9     retail
```

(Due to a quirk in the data, the client names are either all uppercase or titlecase, and we're choosing not to normalize these values - we'll need to correct for this in one of our aggregations.)

Built-in Aggregation Functions

Aggregations are provided by the DataFrame **groupby()** method, which returns a special **groupby** object. If we'd like to see revenue aggregated by region, we can simply select the column to aggregate and call an aggregation function on this object:

```
# revenue sum by region
rsbyr = df.groupby('region').sum()   # call sum() on the groupby object
print rsbyr

                  #         revenue
                  # region
                  # NE           28
                  # NW           33
                  # SW           13


# revenue average by region
rabyr = df.groupby('region').mean()
print rabyr

                  #           revenue
                  # region
                  # NE       9.333333
                  # NW      11.000000
                  # SW       4.333333
```

The result is dataframe with the 'region' as the index and 'revenue' as the sole column.

Note that although we didn't specify the revenue column, pandas noticed that the other columns were not numbers and therefore should not be included in a sum or mean.

If we ask for a count, python counts each column (which will be the same for each). So if we'd like the analysis to be limited to one or more coluns, we can simply slice the dataframe first:

```
# count of all columns by region
print df.groupby('region').count()

                  #         company  revenue  revtype
                  # region
                  # NE            3        3        3
                  # NW            3        3        3
                  # SW            3        3        3


# count of companies by region
dfcr = df[['company', 'region']]       # dataframe slice:  only 'company' and 'region'
print dfcr.groupby('region').count()

                  #         company
                  # region
                  # NE            3
                  # NW            3
                  # SW            3
```

Multi-column aggregation

To aggregate by values in two combined columns, simply pass a list of columns by which to aggregate -- the result is called a "multi-column aggregation":

```
print df.groupby(['region', 'revtype']).sum()

                  #                   revenue
                  # region revtype
                  # NE     retail          12
                  #        wholesale       16
                  # NW     retail          18
                  #        wholesale       15
                  # SW     retail           3
                  #        wholesale       10
```

List of selected built-in groupby functions

```
               count()
               mean()
               sum()
               min()
               max()
               describe() (prints out several columns including sum, mean, min, max)
```

Custom groupby functions

We can design our own custom functions -- we simply use **apply()** and pass a function (you might remember similarly passing a function from the **key=** argument to **sorted()**). Here is the equivalent of the **sum()** function, written as a custom function:

```
def get_sum(df_slice):
    return sum(df_slice['revenue'])

print df.groupby('region').apply(get_sum)  # custom function: same as groupby('region').sum()

                  # region
                  # NE    28
                  # NW    33
                  # SW    13
                  # dtype: int64
```

As was done with **sorted()**, pandas calls our groupby function multiple times, once with each group. The argument that Python passes to our custom function is a dataframe slice containing just the rows from a single grouping -- in this case, a specific region (i.e., it will be called once with a silce of **NE** rows, once with **NW** rows, etc. The function should be made to return the desired value for that slice -- in this case, we want to see the sum of the revenue column (as mentioned, this is simply illustrating a function that does the same work as the built-in **.sum()** function).

(For a better view on what is happening with the function, print **df_slice** inside the function -- you will see the values in each slice printed.)

Here is a custom function that returns the median ("middle value") for each region:

```
def get_median(df):
    listvals = sorted(list(df['revenue']))
    lenvals = len(listvals)
    midval = listvals[ lenvals / 2 ]
    return midval

print df.groupby('region').apply(get_median)

                  # region
                  # NE    10
                  # NW     9
                  # SW     3
                  # dtype: int64
```

Custom aggregator function

Most aggregations aggregate based on a column value or a combination of column values. If more work is needed to identify a group, we can supply a custom function for this operation as well.

In this case, remember that our quirky dataset has company names that are uppercase or lowercase -- thus, aggregating on the company name will treat different casing as a different company:

```
print df.groupby('company').sum()

                  #          revenue
                  # company
                  # ALPHA         11
                  # Alpha         10
                  # BETA          15
                  # Beta          10
                  # Gamma         28
```

So, we can process this column value (or even include other column values) by referencing a function in the call to **groupby()**:

```
def get_lowercase(index):
    row = df.ix[index]              # a Series with the row values for this index
    return row['company'].lower()   # returning this row's company name, lowercased

print df.groupby(get_lowercase).sum()

                  #        revenue
                  # alpha       21
                  # beta        25
                  # gamma       28
```

The value passed to the function is the index of a row. We can thus use the **ix** attribute with the index value to access the row. This function isolates the company name within the row and returns its lowercased value.

using lambdas

Of course any of these simple functions can be rewritten as a lambda (and in many cases, should be, as in the above case since the function references the dataframe directly):

```
def get_lowercase(index):
    row = df.ix[index]              # a Series with the row values for this index
    return row['company'].lower()   # returning this row's company name, lowercased

print df.groupby(lambda index:  df.ix[index]['company'].lower()).sum()
```

Matplotlib

numpy

numpy can provide value ranges useful in plotting

np.arange: produce an array of values

This function produces an **array** object containing a range of values progressing by a certain interval:

```
# args are (min, max, interval)
np.arange(0, 100,   10)    # 0 to 100 by 10's
np.arange(0.,  5., 0.2).   # 0 to 5 by 0.2's
```

np.linspace: produce evenly spaced values within a range

Similarly, this function produces a specific amount of numbers within a range:

```
print np.linspace(0., 5., 2)       # array([ 0., 5.])                       # 2 values between 0 and 5
print np.linspace(0., 5., 3)       # array([ 0., 2.5, 5. ])                 # 3 values between 0 and 5
print np.linspace(0., 5., 4)       # array([ 0., 1.66666667, 3.33333333, 5.]) # 4 values between 0 and 5
```

np.sin and np.cos: produce sine waves

```
X = np.linspace(-np.pi, np.pi, 256)       # 256 values beween -3.14 and 3.14
C,S = np.cos(X), np.sin(X)                # calculate cosine and sine waves from values
```

Matplotlib for visualizations

matplotlib.pyplot is the plotting object; the *figure* is saved within pyplot

* *pylab* was the original module meant to emulate Matlab functionality.

* *Matlab* is a proprietary software product that provides a numeric computing environment.

* *matplotlib* is the Python package of modules that now encompasses **pylab** and its emulation of Matlab's functionality.

A clear tutorial on the central plotting function **pyplot** (part of which was used for this presentation) can be found here: https://matplotlib.org/users/pyplot_tutorial.html (https://matplotlib.org/users/pyplot_tutorial.html)

matplotlib.pyplot

The **pyplot** module is our interface to plotting

```
# plot a line from 0 to 10
import numpy as np
import matplotlib.pyplot as plt
import random

x = range(10)

print x           # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
plt.plot(x)       # plot a line graph with above values along a 10-point horizontal access
plt.ylabel('some values')
plt.xlabel('integer index')

plt.savefig('myplot.png')
```

* Note that in **matplotlib.pyplot as plt**, **plt** represents the plotting library, and much of what we do with our plots will take place through this module.

* The calls to **plt.plot()**, **plt.clf()** and **plt.savefig()**, etc. show that the *figure*, the object that represents the "current" plot, "resides" in the module and not in an external variable. Individual figures can be returned and assigned to variables, but we often work with the default.

Matplotlib within Jupyter Notebook

Use the **%pylab** magic command to invoke plotting functionality

We can begin a plotting session in Jupyter notebook thusly:

```
%pylab                            # load matplotlib visualization functionality

import numpy as np                # we'll use this for number generation
import matplotlib.pyplot as plt
```

When you plot() within a cell, Notebook pops up a window showing the plot.

```
# plot a line of random values from 0 to 10

#plt.clf()        # clear prior figure (useful for multiple plots within a session)
x = [ random.randint(0, 10) for i in range(10)]

plt.plot(x)       # plot a line graph with above values along a 10-point horizontal access
```

However, note what happens when you run the above cell multiple times: the same plot is retained and updated with the new plot.

Matplotlib allows this so you can apply multiple plots in separate calls to the same figure.

If you wish to clear the figure and start fresh, use **plt.clf()** (you can uncomment this line in example above).

plt.plot()

(Note in all examples, **plt** stands for matplotlib.pyplot.)

plot() with a list plots the values along the **y** axis, indexed by list indices along the **x** axis (0-4):

```
plt.plot([1, 2, 3, 4])      # indexed against 0, 1, 2, 3 on x axis
```

With two lists, plots the values in the first list along the **y** axis, indexed by the second list along the **x** axis:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

With a third argument, changes the style of the line:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16] ,'ro')     # style line as red dots
```

Style choices are listedin the api reference: https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot (https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot)

multiple plots in a single call

```
t = np.arange(0., 5., 0.2)
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

plt.axis() to control view of the axis

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.axis([0, 10, 0, 50])                # set 'y' axis to 0-10 and 'x' axis to 0-50
```

Other Chart Types

.

Bar Chart:

```
langs =    ('Python', 'C++', 'Java', 'Perl', 'Scala', 'Lisp')
langperf = [     10,     8,      6,      4,       2,      1]

y_pos = np.arange(len(langs))

plt.bar(y_pos, langperf, align='center', alpha=0.5)
plt.xticks(y_pos, langs)
plt.ylabel('Usage')
plt.title('Programming language usage')
```

Pie Chart:

```
plt.pie([2, 3, 10, 20])
```

Scatter Plot:

```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
```

Visualizing pandas Series and DataFrame

pandas has fully incorporated matplotlib into its API.

pandas Series objects have a **plot()** method that works

```
import pandas as pd
import numpy as np

ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()
ts.plot(kind="line")   # "line" is default
```

DataFrame.plot()

```
df = pd.read_csv('aapl.csv')
df.index = df['Date']         # make the Date column into the index
df = df['Close']              # isolate just the close price for this DataFrame
df.plot()                     # "line" is default style
```

Pandas DataFrames also have a set of methods that create the type of chart desired.

```
df.plot.area     df.plot.barh     df.plot.density  df.plot.hist     df.plot.line     df.plot.scatter
df.plot.bar      df.plot.box      df.plot.hexbin   df.plot.kde      df.plot.pie
```

The pandas visualization documentation can be found here: http://pandas.pydata.org/pandas-docs/stable/visualization.html (http://pandas.pydata.org/pandas-docs/stable/visualization.html)

Figures and Subplots

Matplotlib allows for multiple

A fine discussion can be found at http://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html#figures-subplots-axes-and-ticks (http://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html#figures-subplots-axes-and-ticks)

Benchmarking and Efficiency

Efficiency: Introduction

Runtime Efficiency refers to two things: *memory* efficiency (whether a lot of RAM memory is being used up during a process) and *time* efficiency (how long execution takes). And these are often related -- it takes time to allocate memory.

As a "scripting" language, Python is more convenient, but less efficient than "programming" languages like C and Java:

* Parsing, compilation and execution take place during runtime
  (C and Java are compiled ahead of time)

* Memory is allocated based on anticipation of what your code will do at runtime
  (C in particular requires the developer to indicate what memory will be needed)

* Python handles expanded memory requests seamlessly -- "no visible limits"
  (C and Java make use of "finite" resources, they do not expand indefinitely)

Achieving runtime efficiency requires a tradeoff with required development time -- so we either spend more of our own (developer) time making our programs more efficient so they run faster and use less memory, or we spend less time developing our programs, and allow them to run slower (as Python handles memory allocation for us).

Of course just the choice of a convenient scripting language over a more efficient programming language means that rapid development, ease of use, etc. are more important to us than runtime efficiency, so in many applications efficiency is not a consideration because there's plenty of memory and time to get the job done.

However, advanced Python developers may be asked to increase the efficiency of their programs, because the task at hand is simply taking longer than it should -- the data has grown past anticipated limits, the program's responsibilities and complexity has been extended, or an unknown inefficiency is bogging down execution.

In this section we'll discuss the more efficient container structures and ways to analyze the speed of the various units in our programs.

Collections: high performance container datatypes
* **array**: type-specific list
* **deque**: "double-ended queue"
* **Counter**: a counting dictionary
* **defaultdict**: a dict with automatic default for missing keys

timeit: unit timer to compare time efficiency of various Python algorithms

cProfile: overall time profile of a Python program

"enhanced" Python packages

Collections

The **collections** module provides more efficient containers.

array: a list of a uniform type
An array's type must be specified at initialization. A uniform type makes an array more efficient than a list, which can contain any type.

```
from array import array

myarray = array('i', [1, 2])

myarray.append(3)

print myarray           # array('i', [1, 2, 3])

print myarray[-1]       # acts like a list
for val in myarray:
    print val

myarray.append(1.3)     # error
```

(array is not part of the Collections module but is usefully discussed here)

Available **array** types:

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2 (see note)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

deque: "double-ended queue" for fast adds/removals **list**s are optimized for fixed-length operations, i.e., things like sorting, checking for membership, index access, etc. They are not optimized for appends, although this is of course a common use for them. A **deque** is designed specifically for fast adds -- to the beginning or end of the sequence:

```
from collections import deque

x = deque([1, 2, 3])

x.append(4)               # x now [1, 2, 3, 4]
x.appendleft(0)           # x now [0, 1, 2, 3, 4]

popped = x.pop()          # removes '4' from the end

popped2 = x.popleft()     # removes '1' from the start
```

A deque can also be sized, in which case appends will push existing elements off of the ends:

```
x = deque(['a', 'b', 'c'], 3)      # maximum size:  3
x.append(99)                       # now: deque(['b', 'c', 99])  ('a' was pushed off of the start)
x.appendleft(0)                    # now: deque([0, 'b', 'c'])   (99 was pushed off of the end)
```

Counter: a counting dictionary

This structure inherits from **dict** and is designed to allow an integer count as well as a default **0** value for new keys. So instead of doing this:

```
c = {}
if 'a' in c:
    c['a'] = 0
else:
    c['a'] + 1
```

We can do this:

```
from collections import Counter

c = Counter()
c['a'] = c['a'] + 1
```

Counter also has related functions that return a list of its keys repeated that many times, as well as a list of tuples ordered by frequency:

```
from collections import Counter

c = Counter({'a': 2, 'b': 1, 'c': 3, 'd': 1})

for key in c.elements():
    print key,            # c c c a a b b

print ','.join(c.elements())   # c,c,c,a,a,b,b


print c.most_common(2)   # [('c', 3), ('a', 2)]
                         # 2 arg says "give me the 2 most common"

c.clear()                # set all counts to 0 (but not remove the keys)
```

And, you can use **Counter**'s implementation of the math operators to work with multiple counters and have them sum their values:

```
c = Counter({'a': 1, 'b': 2})
d = Counter({'a': 10, 'b': 20})

print c + d                    # Counter({'b': 22, 'a': 11})
```

defaultdict: a default object for new missing keys

Similar to **Counter**, **defaultdict** allows for a default value if a key doesn't exist; but it will accept a function that provides a default value.

```
from collections import defaultdict

ddict = defaultdict(list)

ddict['a'].append(1)
ddict['b']

print ddict                    # defaultdict(, {'a': [1], 'b': []})
```

Benchmarking a Python function with timeit

The **timeit** module provides a simple way to time blocks of Python code.

We can thus decide whether varying approaches might make our programs more efficient. Here we compare execution time of four approaches to joining a range of integers into a very large string ("1-2-3-4-5...", etc.)

```
from timeit import timeit

# 'straight concatenation' approach
def joinem():
    x = '1'
    for num in range(100):
        x = x + '-' + str(num)
    return x

print timeit('joinem()', setup='from __main__ import joinem', number=10000)

# 0.457356929779


# generator comprehension
print timeit('"-".join(str(n) for n in range(100))', number=10000)

# 0.338698863983


# list comprehension
print timeit('"-".join([str(n) for n in range(100)])', number=10000)

# 0.323472976685


# map() function
print timeit('"-".join(map(str, range(100)))', number=10000)

# 0.160399913788
```

map() is fastest probably because it is compiled in **C**.

Repeating a test

You can conveniently repeat a test multiple times by calling a method on the object returned from **timeit()**. Repetitions give you a much better idea of the average time it might take.

```
from timeit import repeat

print repeat('"-".join(map(str, range(100)))', number=10000, repeat=3)

# [0.15206599235534668, 0.1909959316253662, 0.2175769805908203]


print repeat('"-".join([str(n) for n in range(100)])', number=10000, repeat=3)

# [0.35890698432922363, 0.327725887298584, 0.3285980224609375]


print repeat('"-".join(map(str, range(100)))', number=10000, repeat=3)

# [0.14228010177612305, 0.14016509056091309, 0.14458298683166504]
```

***setup*= parameter for setup before a test**
Some tests make use of a variable that must be initialized before the test:

```
print timeit('x.append(5)', setup='x = []', number=10000)

# 0.00238704681396
```

Additionally, **timeit()** does not share the program's global namespace, so imports and even global variables must be imported if required by the test:

```
print timeit('x.append(5)', setup='import collections as cs; x = cs.deque()', number=10000)

# 0.00115013122559
```

Here we're testing a function, which as a global needs to be imported from the __**main**__ namespace:

```
def testme(maxlim):
    return [ x*2 for x in range(maxlim) ]

print timeit('testme(5000)', setup='from __main__ import testme', number=10000)

# 10.2637062073
```

Keep in mind that a function tested in isolation may not return the same results as a function using a different dataset, or a function that is run as part of a larger program (that has allocated memory differently at the point of the function's execution). The **cProfile** module can test overall program execution.

Profiling a Python program with cProfile

The profiler runs an entire script and times each unit (call to a function).

If a script is running slowly it can be difficult to identify the bottleneck. **timeit()** may not be adequate as it times functions in isolation, and not usually with "live" data.

This test program (**ptest.py**) deliberately pauses so that some functions run slower than others:

```
import time

def fast():
    print("I run fast!")


def slow():
    time.sleep(3)
    print("I run slow!")


def medium():
    time.sleep(0.5)
    print("I run a little slowly...")


def main():
    fast()
    slow()
    medium()

if __name__ == '__main__':
    main()
```

We can profile this code thusly:

```
>>> import cProfile
>>> import ptest
>>> cProfile.run('ptest.main()')
I run fast!
I run slow!
I run a little slowly...
         8 function calls in 3.500 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.500    3.500 :1()
        1    0.000    0.000    0.500    0.500 ptest.py:15(medium)
        1    0.000    0.000    3.500    3.500 ptest.py:21(main)
        1    0.000    0.000    0.000    0.000 ptest.py:4(fast)
        1    0.000    0.000    3.000    3.000 ptest.py:9(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    3.499    1.750    3.499    1.750 {time.sleep}
```

According to these results, the **slow()** and **main()** functions are the biggest time users. The overall execution of the module itself is also shown. Comparing our code to the results we can see that **main()** is slow only because it calls **slow()**, so we can then focus on the obvious culprit, **slow()**.

It's also possible to insider profiling *in our script* around particular function calls so we can focus our analysis.

```
profile = cProfile.Profile()
profile.enable()
main()                          # or whatever function calls we'd prefer to focus on
profile.disable()
```

Command-line interface to cProfile

```
python -m cProfile -o output.bin ptest.py
```

The **-m** flag on any Python invocation can import a module automatically. **-o** directs the output to a file. The result is a binary file that can be analyzed using the **pstats** module (which we see results in largely the same output as **run()**:

```
>>> import pstats
>>> p = pstats.Stats('output.bin')
>>> p.strip_dirs().sort_stats(-1).print_stats()
Thu Mar 20 18:32:16 2014    output.bin

         8 function calls in 3.501 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.501    3.501 ptest.py:1()
        1    0.001    0.001    0.500    0.500 ptest.py:15(medium)
        1    0.000    0.000    3.501    3.501 ptest.py:21(main)
        1    0.001    0.001    0.001    0.001 ptest.py:4(fast)
        1    0.001    0.001    3.000    3.000 ptest.py:9(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    3.499    1.750    3.499    1.750 {time.sleep}


<pstats.Stats instance at 0x017C9030>
```

Caveat: don't optimize prematurely

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
-- Donald Knuth

Common wisdom suggests that optimization should happen only once the code has reached a working, clear-to-finalized state. If you think about optimization too soon, you may do work that has to be undone later; or your optimizations may themselves get undone as you complete the functionality of your code.

Note: some of these examples taken from the "Mouse vs. Python (http://www.blog.pythonlibrary.org/)" blog.

Python Enhancement Packages

These packages provide varying approaches toward writing and running more efficient Python code.

- **PyPy**: a "Just in Time (https://en.wikipedia.org/wiki/Just-in-time_compilation)" compiler for Python -- can speed up almost any Python code.
- **Cython**: superset of the Python language that additionally supports calling of C functions and declaring C types -- good for building Python modules in C.
- **Pyrex**: compiler that lets you combine Python code with C data types, compiling your code into a C extension for Python.
- **Weave**: allows embedding of C code within Python code.
- **Shed Skin**: an experimental module that can translate Python code into optimized C++.

While **PyPy** is a no-brainer for speeding up code, the other libraries listed here require a knowledge of **C**. The deepest analysis of Python will incorporate efficient C code and/or take into account its underlying C implementation. Python is written in C, and operations that we invoke in Python translate to actions being taken by the compiled C code. The most advanced Python developer will have a working knowledge in C, and study the C structures that Python employs.

Algorithmic Complexity Analysis and "Big O"

Introduction: Algorithmic Complexity Analysis

Algorithms can be analyzed for efficiency based on how they respond to varying amounts of input data.

Algorithm: a block of code designed for a particular purpose.
You may have heard of a sort algorithm, a mapping or filtering algorithm, a computational algorithm; Google's vaunted search algorithm or Facebook's "feed" algorithm; all of these refer to the same concept -- a block of code designed for a particular purpose. But any block of code is an algorithm, including simple ones.

Since algorithms can be well designed or poorly designed, time efficient or inefficient, memory efficient or inefficient, it becomes a meaningful discipline to analyze the efficiency of one approach over another.

Several of the examples and information in this presentation can be found in a really clear textbook on the subject, Problem Solving with Algorithms and Data Structures (http://www.amazon.com/gp/product/1590282574/ref=pd_lpo_sbs_dp_ss_1/190-8133639-3080607?pf_rd_m=ATVPDKIKX0DER&pf_rd_s=lpo-top-stripe-1&pf_rd_r=1D0RQVRFS16CYYH3A67G&pf_rd_t=201&pf_rd_p=1944687722&pf_rd_i=1590280539), also available as a free PDF (https://www.cs.auckland.ac.nz/courses/compsci105s1c/resources/ProblemSolvingwithAlgorithmsandDataStructures.p

The *order* ("growth rate") of a function

The order describes the growth in *steps* of a function as the input size grows.

A "step" can be seen as any individual statement, such as an assignment or a value comparison. Depending on its design, an algorithm may take take the same number of steps no matter how many elements are passed to input ("constant time"), an increase in steps that matches the increase in input elements ("linear growth"), or an increase that grows faster than the increase in input elements ("logarithmic", "linear logarithmic", "quadratic", etc.).

Order is about growth of number of steps, not absolute number of steps
Consider this simple file field summer. How many more steps for a file of 5 lines than a file of 10 lines (double the growth rate)? How many more for a file of 1000 lines?

```
def sum_fieldnum(filename, fieldnum, delim):
    this_sum = 0.0
    fh = open(filename)
    for line in fh:
        items = line.split(delim)
        value = float(items[fieldnum])
        this_sum = this_sum + value
    fh.close()
    return this_sum
```

Obviously several steps are being taken -- 5 steps that don't depend on the data size (initial assignment, opening of filehandle, closing of filehandle and return of summed value) and 3 steps taken once for each line of the file (split the line, convert item to float, add float to sum)

Therefore, with varying input file sizes, we can calulate the steps:

```
   5 lines:  5 + (3 * 5),    or 5 + 15,   or 20 steps
  10 lines:  5 + (3 * 10),   or 5 + 30,   or 35 steps
1000 lines:  5 + (3 * 1000), or 5 + 3000, or 3005 steps
```

As you can see, the 5 "setup" steps become trivial as the input size grows -- it is 25% of the total with a 5-line file, but 0.0016% of the total with a 1000-line file, which means that we should consider only those steps that are affected by input size -- the rest are simply discarded from analysis.

A simple algorithm: sum up a list of numbers

Here's a simple problem that will help us understand the comparison of algorithmic approaches.

It also happens to be an interview question I heard when I was shadowing an interview:

*Given a maximum value **n**, sum up all values from 0 to the maximum value.*

"range" approach:

```
def sum_of_n_range(n):
    total = 0
    for i in range(1,n+1):
        total = total + i
    return total

print sum_of_n_range(10)
```

"recursive" approach:

```
def sum_of_n_recursive(total, count, this_max):
    total = total + count
    count += 1
    if count > this_max:
        return total
    return sum_of_n_recursive(total, count, this_max)

print sum_of_n_recursive(0, 0, 10)
```

"formula" approach:

```
def sum_of_n_formula(n):
    return (n * (n + 1)) / 2

print sum_of_n_formula(10)
```

We can analyze the respective "order" value for each of these functions by comparing its behavior when we pass it a large vs. a small value. We count each statement as a "step".

The "range" solution begins with an assignment. It loops through each consecutive integer between 1 and the maximum value. For each integer it performs a sum against the running sum, then returns the final sum. So if we call **sum_of_n_range** with **10**, it will perform the sum (total + i) **10** times. If we call it with 1,000,000, it will perform the sum 1,000,000 times. The increase in # of steps increases in a straight line with the # of values to sum. We call this *linear growth*.

The "recursive" solution calls itself once for each value in the input. This also requires a step increase that follows the increase in values, so it is also "linear".

The "formula" solution, on the other hand, arrives at the answer through a mathematic formula. It performs an addition, multiplication and division of values, but the computation is the same regardless of the input size. So whether 10 or 1,000,000, the number of steps is the same. This is known as *constant time*.

"Big O" notation

The ***order*** of a function (the growth rate of the function as its input size grows) is expressed with a mathematical expression colloquially referred to as "Big O".

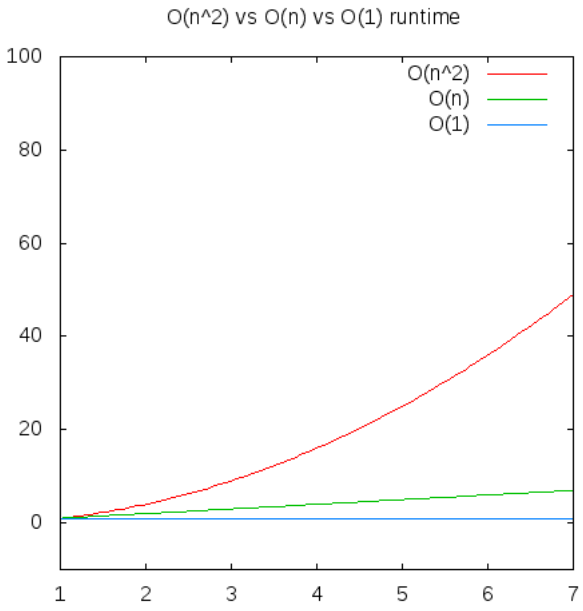
Common function notations for Big O

Here is a table of the most common growth rates, both in terms of their O notation and English names:

```
&sup
```

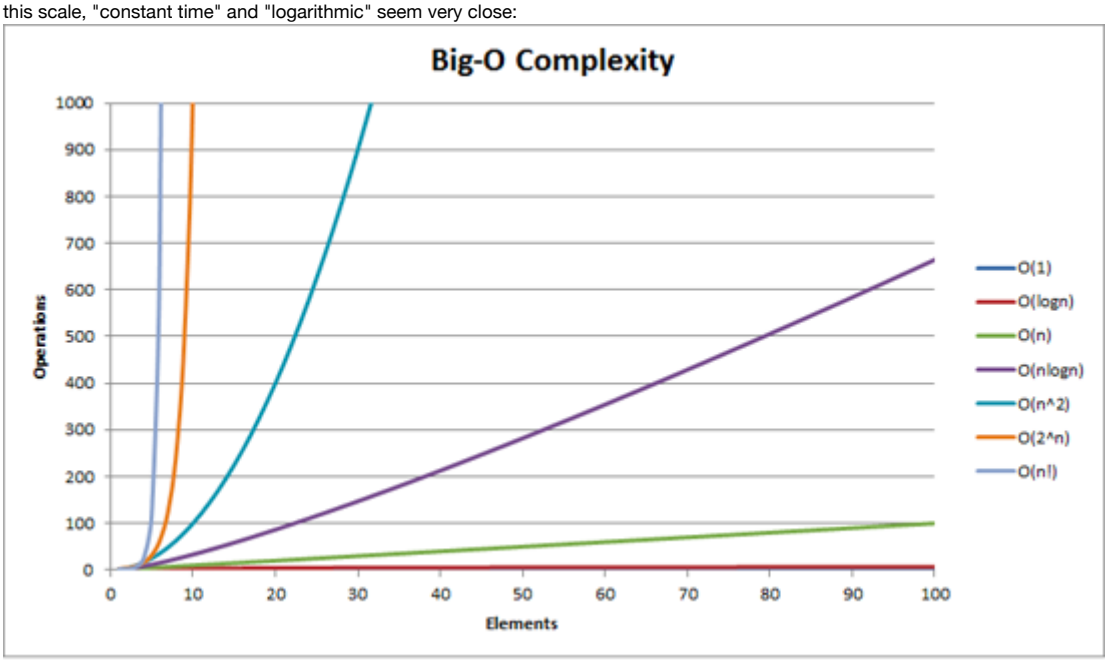
"O" Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log Linear
O(n²)	Quadratic
O(n³)	Cubic
O(2^n) (2 to the power of n)	Exponential

Here's a graph of the constant, linear and exponential growth rates:



Here's a graph of the other major scales. You can see that at

this scale, "constant time" and "logarithmic" seem very close:



Constant time: O(1)

A function that does not grow in steps or operations as the input size grows is said to be running at *constant time*.

```
def sum_of_n_formula(n):
    return (n * (n + 1)) / 2

print sum_of_n_formula(10)
```

That is, no matter how big **n** gets, the number of operations stays the same.

"Constant time" growth is noted as O(1).

Linear growth: O(n)

The growth rate for the "range" solution to our earlier summing problem (repeated below) is known as *linear* growth.

With linear growth, as the input size (or in this case, the integer value) grows, the number of steps or operations grows at the same rate:

```
def sum_of_n_range(n):
    the_sum = 0
    for i in range(1,n+1):
        the_sum = the_sum + i
    return the_sum

print sum_of_n_range(10)
```

Although there is another operation involved (the assignment of the_sum to 0), this additional step becomes trivial as the input size grows. We tend to ignore this step in our analysis because we are concerned with the function's growth, particularly as the input size becomes large.

Linear growth is noted as O(n) where again, n is the input size -- growth of operations matching growth of input size.

Logarithmic growth: O(log n)

A *logarithm* is an equation used in algebra. We can consider a log equation as the *inverse* of an exponential equation:

b^c = a ("b to the power of c equals a")

```
10³ = 1000   ## 10 cubed == 1000
```

is considered equivalent to:

$\log_b a = c$ ("log with base b and value a equals c")

```
log10 1000 = 3
```

A *logarithmic scale* is a nonlinear scale used to represent a set of values that appear on a very large scale and a potentially huge difference between values, with some relatively small values and some exponentially large values. Such a scale is needed to represent all points on a graph without minimizing the importance of the small values.

Common uses of a logarithmic scale include earthquake magnitude, sound loudness, light intensity, and pH of solutions.

For example, the Richter Scale of earthquake magnitude grows in absolute intensity as it moves up the scale -- 5.0 is 10 times that of 4.0; 6.0 is 10 times that of 5.0; 7.0 is 10 times that of 6.0, etc. This is known as a *base 10* logarithmic scale.

In other words, a base 10 logarithmic scales runs as:

```
1, 10, 100, 1,000, 10,000, 100,000, 1,000,000
```

Logarithms in Big O notation

However, the O(log n) ("Oh log of n") notation refers to a *base 2* progression - 2 is twice that of 1, 3 is twice that of 2, etc.

In other words, a base 2 logarithmic scale runs as:

```
1, 2, 4, 8, 16, 32, 64
```

A classic *binary search* algorithm on an *ordered list of integers* is O(log n). You may recognize this as the "guess a number from 1 to 100" algorithm from one of the extra credit assignments.

```
def binary_search(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

print binary_search([1, 3, 4, 9, 11, 13], 11)

print binary_search([1, 2, 4, 9, 11, 13], 6)
```

The assumption is that the search list is sorted. Note that once the algorithm decides whether the search integer is higher or lower than the current midpoint, it "discards" the other half and repeats the binary searching on the remaining values.

Since the number of loops is basically n/2/2/2/2, we are looking at a logarithmic order. Hence O(log n)

Linear logarithmic growth: O(n log n)

The basic approach of a *merge sort* is to halve it; loop through half of it; halve it again.

```
def merge_sort(a_list):
    print("Splitting ", a_list)
    if len(a_list) > 1:
        mid = len(a_list) / 2      # (int/int, so lop off any remainder)
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = 0
        j = 0
        k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                a_list[k] = left_half[i]
                i = i + 1
            else:
                a_list[k] = right_half[j]
                j = j + 1
            k = k + 1

        while i < len(left_half):
                a_list[k] = left_half[i]
                i = i + 1
                k = k + 1

        while j < len(right_half):
                a_list[k] = right_half[j]
                j = j + 1
                k = k + 1
    print("Merging ", a_list)

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
merge_sort(a_list)
print a_list
```

The output of the above can help us understand what portions of the unsorted list are being managed:

```
('Splitting ', [54, 26, 93, 17, 77, 31, 44, 55, 20])
('Splitting ', [54, 26, 93, 17])
('Splitting ', [54, 26])
('Splitting ', [54])
('Merging ', [54])
('Splitting ', [26])
('Merging ', [26])
('Merging ', [26, 54])
('Splitting ', [93, 17])
('Splitting ', [93])
('Merging ', [93])
('Splitting ', [17])
('Merging ', [17])
('Merging ', [17, 93])
('Merging ', [17, 26, 54, 93])
('Splitting ', [77, 31, 44, 55, 20])
('Splitting ', [77, 31])
('Splitting ', [77])
('Merging ', [77])
('Splitting ', [31])
('Merging ', [31])
('Merging ', [31, 77])
('Splitting ', [44, 55, 20])
('Splitting ', [44])
('Merging ', [44])
('Splitting ', [55, 20])
('Splitting ', [55])
('Merging ', [55])
('Splitting ', [20])
('Merging ', [20])
('Merging ', [20, 55])
('Merging ', [20, 44, 55])
('Merging ', [20, 31, 44, 55, 77])
('Merging ', [17, 20, 26, 31, 44, 54, 55, 77, 93])
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Here's an interesting description comparing O(log n) to O(n log n):

```
log n is proportional to the number of digits in n.

n log n is n times greater.

Try writing the number 1000 once versus writing it one thousand times.
The first takes O(log n) time, the second takes O(n log n) time.

Now try that again with 6700000000. Writing it once is still trivial.
Now try writing it 6.7 billion times.
We'll check back in a few years to see your progress.
```

Quadratic growth: O(n²)

O(n²) growth can best be described as "for each element in the sequence, loop through the sequence". This is why it's notated as n².

```
def all_combinations(the_list):
   results = []
   for item in the_list:
       for inner_item in the_list:
           results.append((item, inner_item))
   return results

print all_combinations(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

Clearly we're seeing n * n, so 49 individual tuple appends.

```
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'e'), ('a', 'f'), ('a', 'g'), ('b', 'a'), ('b', 'b')
```

"order" analysis example

Let's take an *arbitrary example* to analyze. This algorithm is working with the variable n -- we have not defined n because it represents the input data, and our analysis will ask: how does the time needed change as n grows? However, we can assume that n is a sequence.

```
a = 5                   # count these up:
b = 6                   # 3 statements
c = 10

for k in range(n):
    w = a * k + 45      # 2 statements:
    v = b * b           # but how many times
                        # will they execute?

for i in range(n):
    for j in range(n):
        x = i * i       # 3 statements:
        y = j * j       # how many times?
        z = i * j

d = 33                  # 1 statement
```

- **We can count assignment statements that are executed once: there are 4 of these.**
- **The 2 statements in the first loop are each being executed once for each iteration of the loop -- and it is iterating n times. So we call this 2n.**
- **The 3 statements in the second loop are being executed n times * n times (a nested loop of range(n). We can call this n^2 ("n squared"). So the order equation can be expressed as $4 + 2n + n^2$ eliminating the trivial factors However, remember that this analysis describes the *growth rate* of the algorithm as input size n grows very large. As n gets larger, the impact of 4 and of 2n become less and less significant compared to n^2; eventually these elements become trivial. So we eliminate the lessor factors and pay attention only to the most significant -- and our final calculation is $O(n^2)$.**

More Big O recursion examples

These were adapted from a stackoverflow question (http://stackoverflow.com/questions/13467674/determining-complexity-for-recursive-functions-big-o-notation).

Just for fun(!) these are presented without answers; answers on the next page.

```
def recurse1(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse1(n-1)
```

```
def recurse2(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse2(n-5)
```

```
def recurse3(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse3(n / 5)
```

```
def recurse4(n, m, o):
    if n <= 0:
        print '{}, {}'.format(m, o)
    else:
        recurse4(n-1, m+1, o)
        recurse4(n-1, m, o+1)
```

```
def recurse5(n):
    for i in range(n)[::2]:       # count to n by 2's (0, 2, 4, 6, 7, etc.)
        pass
    if n <= 0:
        return 1
    else:
        return 1 + recurse5(n-5)
```

More Big O recursion examples: analysis

```
def recurse1(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse1(n-1)
```

This function is being called recursively n times before reaching the base case so it is O(n) (linear)

```
def recurse2(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse2(n-5)
```

This function is called n-5 for each time, so we deduct five from n before calling the function, but n-5 is also O(n) (linear).

```
def recurse3(n):
    if n <= 0:
        return 1
    else:
        return 1 + recurse3(n / 2)
```

This funciton is log(n), for every time we divide by 2 before calling the function.

```
def recurse4(n, m, o):
    if n <= 0:
        print '{}, {}'.format(m, o)
    else:
        recurse4(n-1, m+1, o)
        recurse4(n-1, m, o+1)
```

In this function it's O(2^n), or exponential, since each function call calls itself twice unless it has been recursed n times.

```
def recurse5(n):
    for i in range(n)[::2]:       # count to n by 2's (0, 2, 4, 6, 7, etc.)
        pass
    if n <= 0:
        return 1
    else:
        return 1 + recurse5(n-5)
```

The for loop takes n/2 since we're increasing by 2, and the recursion takes n-5 and since the for loop is called recursively, the time complexity is in (n-5) * (n/2) = (2n-10) * n = 2n^2- 10n, so O(n^2)

Efficiency of core Python Data Structure Algorithms

note: "k" is the list being added/concatenated/retrieved

List	
Operation	Big-O Efficiency
index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n + k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

Dict	
Operation	Big-O Efficiency (avg.)
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
contains (in)	O(1)
iteration	O(n)

A rundown of each structure and the O time to complete operations on each structure are noted here: https://wiki.python.org/moin/TimeComplexity (https://wiki.python.org/moin/TimeComplexity)

Common Algorithms Organized by Efficiency

O(1) time

1. Accessing Array Index (int a = ARR[5])

2. Inserting a node in Linked List

3. Pushing and Poping on Stack

4. Insertion and Removal from Queue

5. Finding out the parent or left/right child of a node in a tree stored in Array
6. Jumping to Next/Previous element in Doubly Linked List
and you can find a million more such examples...

O(n) time
1. Traversing an array
2. Traversing a linked list
3. Linear Search
4. Deletion of a specific element in a Linked List (Not sorted)
5. Comparing two strings
6. Checking for Palindrome
7. Counting/Bucket Sort
and here too you can find a million more such examples....
In a nutshell, all Brute Force Algorithms, or Noob ones which require linearity, are based on O(n) time complexity

O(log n) time
1. Binary Search
2. Finding largest/smallest number in a binary search tree
3. Certain Divide and Conquer Algorithms based on Linear functionality
4. Calculating Fibonacci Numbers - Best Method
The basic premise here is NOT using the complete data, and reducing the problem size with every iteration

O(n log n) time
1. Merge Sort
2. Heap Sort
3. Quick Sort
4. Certain Divide and Conquer Algorithms based on optimizing O(n^2) algorithms
The factor of 'log n' is introduced by bringing into consideration Divide and Conquer. Some of these algorithms are the best optimized ones and used frequently.

O(n^2) time
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Traversing a simple 2D array
These ones are supposed to be the less efficient algorithms if their O(n log n) counterparts are present. The general application may be Brute Force here.

Practice Problem: Sequential Search

Given a test item (an integer), search through a list of integers to see if that item's value is in the list. Perform this in three ways:

1) assuming the list is unsorted
2) assuming the list is sorted

Answers appear on next slide.

Practice Problem: Sequential Search (Answers)