Python

This handout covers a small subset of the Python language. The complete language and its libraries are well-covered in books and online sites. The starting point for finding documentation and anything else about Python is www.python.org. The version of Python being used in this class is 2.7, which is compatible with a number of graphics and numerical libraries we need. A few things that are changed in Python 3 versions are noted.

Editing and Running a Python Program

For this course, we will write and edit Python code using Emacs and invoke it from the command line of the Unix operating system that underlies the Mac OS X user interface. Emacs and Unix will be covered in other handouts.

Statements

A Python program is a collection of **statements**, each of which manipulates an **object** in some way, such as by changing the value stored in a variable name. In the absence of other indications, **a single line of code is a single statement**. Other indications include:

\ If a statement is too long for a single line, placing a backslash at the end of one line indicates that the next line is a **continuation line**, meaning that it continues the current statement.

([{ Leaving an **open left parenthesis, left square bracket, or left curly bracket** before completing a line also makes the subsequent line or lines into a continuation line. The statement continues over as many lines as needed to find the matching right parenthesis or bracket. Bracket or parenthesis pairs left open accidentally are a common source of errors, and the error message will be confusing because it will usually point to the line *after* the one with the mistake.

A hash mark anywhere on a line turns the rest of that line into a **comment**. Comments are notes outside the language, ignored by the computer but intended to help a human reader understand the code and to document its purpose. When annotating a program, keep in mind that the person most likely to read your program next is yourself at some time in the future. You may save future time by writing notes in the present. For exercises in this course, please put your name and the exercise number in comments at the top of each program.

; A semicolon may be used to break a line into two or more statements. Doing this makes the code harder to read and should be avoided except for a series of short initialiizations.

Objects

Any item of data or information in Python is an **object**. Objects are categorized by usage: each object is in a **class** or has a **type**.

Objects are created when an object **identifier** or **object name** is attached to either a **literal representation** of the object or to an **expression**.

object_name = *expression*

or

object_name = *literal value*

where *object_name* is the identifier – a name the programmer makes up. Identifiers may only contain letters (`A`–`Z`, `a`–`z`), digits (`0`–`9`), and the underscore character (`_`); and the first character cannot be a digit. Starting a name with an underscore character has a special meaning; do not start a name with underscore in this class. Case matters in Python identifiers: `Dog`, `dog`, and `DoG` are as different from each other as from `cat`. The length (in characters) of an object identifier is unlimited, but try to be concise while being descriptive. In a situation where `T` could ambiguously refer to various things, `Tsun` or `SunTemp` might be nicely descriptive while `Temperature_of_the_Sun` is verbose and messy.

The literal represention of an object is a way of representing an object value within a Python program. The form or existence of a literal varies with the class of the object. Five fundamental classes of Python object are:

string Any set of characters: alphabetic, numeric, or special characters. These are understood by Python as a string of character codes. The string `'123'` is understood as three characters in sequence, not as the number 123.

Literal representation of string objects is as a string of characters surrounded by either single or double quotation marks: `'George'`, `"New Hampshire"`, or `'10/29/1975'` as examples. The apostrophes and quotation marks are not part of the string; they are just delimiters.

float Short for **floating-point number**, this is the type used to represent data that may need large ranges, tiny precisions, or decimal fractions. Most physical quantities are represented by floating-point numbers; they are a computer analog to the "real" number set from mathematics.

Literal representation of float objects is any string of digits that includes either a decimal point somewhere or the `E` symbol that indicates a power of 10, with or without a leading sign. (In the following examples, the last number is 2.76×10^{-6}):
`1.0, 3.14159, -25.4, 2.76E-6`

integer Another number type, but with some restrictions compared to float. An integer cannot have any decimal point or power-of-ten information, but can only be a positive or negative whole number, or zero.

Literal representation of integer objects is any string of digits, optionally with a sign, but never including a decimal point. (Neither integer nor floating-point numbers may include commas to separate blocks of three digits.)
`2, 1234567, -101`

Distinctions between *integer* and *float* are important, even though both store numbers. Integer objects will never contain fractional parts, and fractional parts created by arithmetic (see below) will be discarded. Float objects have a greater range and precision, but they are always an approximation and cannot be used for certain intrinsically whole-number programming purposes.

file The file object does not actually store the information in a file, but rather establishes a connection between the Python program and the real-world file, which is a collection of data stored somewhere accessible on the computer's filesystems. The file object name, in the Python program, is a form of alias or pointer that provides an internal (only within the program) name for what you would normally refer to via paths or directories and filenames in other software.

Literal representation of file objects does not happen within Python code, but rather file object names do have to be connected to their real-world files using a character string (literal or variable) to identify the file and path. Such character strings are *operating system dependent*. Examples of defining file objects with the *open* function and file references are shown in the Input/Output section.

list Any collection of strings, floating point numbers, or integer numbers, listed in some order, can be a list.

Literal representation of list objects is a comma-separated list of items enclosed in square brackests. Following are a list of integers, a list of floating-point numbers, and a list of strings:

```
[23, 45, 67, 89]
[3.14159254, 1.76E-34, 8314.4]
[ 'Eric', 'Graham', 'John', 'Michael', 'Terry' ]
```

Arrays are a special form of list in which all of the elements are numbers of the same type. Defining array objects with the Numeric Python module is discussed below.

Python has **keywords** which cannot be used as object names—they are reserved for use in defining program constructs. The current (Python 3) list is:

False	**class**	**finally**	**is**	**return**
None	**continue**	**for**	**lambda**	**try**
True	**def**	**from**	**nonlocal**	**while**
and	**del**	**global**	**not**	**with**
as	**elif**	**if**	**or**	**yield**
assert	**else**	**import**	**pass**	
break	**except**	**in**	**raise**	

Assignments and Arithmetic with Numbers

Object creation and modification statements often look like algebraic or arithmetic assignment, as in

object_name = *expression*

where the *expression* can be a literal representation of the object, the object name for an object that has already been defined, or an arithmetic expression. The *object_name* on the left side of the equals sign may also appear in the *expression*, in which case this is a **modification** statement instead of a creation statement – the old value of *object_name* from before the statement is used to calculate the *expression*, and the new value then **replaces** the old value. Examples of defining a string, a floating-point number, and a list containing integers, using their literal representations:

```
a = "Uncle John's Band"
b = 5.67E-8
c = [ 1, 2, 4, 5, 6 ]
```

Numeric arithmetic Symbols used for binary operations (operations that turn two numbers into a single result) in Python are

`+`	Addition	`-`	Subtraction
`*`	Multiplication	`/`	Division
`**`	Exponentiation (i.e., x^2 is written as `x ** 2`)		

Using these operators, expressions that look very much like algebra can often be generated. For example, the algebraic expression $a = 2b + c$ translates to

```
a = 2 * b + c
```

The multiplication implied by $2b$ in algebra *requires* a multiplication symbol in Python. However, the algebraic expression and the Python statement have different meanings. The algebraic statement is a static statement that implies something unchangeable about the nature of a, b, and c. The Python statement is an executable statement that tells the computer to do something: calculate the value of `2 * b + c` and label that result with the object name `a`. Thus, the equation $a = a + 5$ is totally impossible in algebra—it can never be true—but the statement

```
a = a + 5
```

is a perfectly legal Python instruction to take the existing value of `a`, add 5 to it, and make the new value be associated with `a` (the old value of `a` is thrown away).

The usual **order of precedence** is followed by Python:

`()` Subexpressions inside parentheses first. Only round parentheses `(` and `)` are used, but you may nest as many pairs as you need.

Function references next.

`**` Exponentials are first operator, right to left.

`* /` Multiply and divide next, left to right

`+ -` Add and subtract last, left to right

Python also includes shortcut operators for modifying an existing number with another number – note that the operater must be *before* the equals sign.

`linecount += 1`	is equivalent to	`linecount = linecount + 1`
`average /= linecount`	is equivalent to	`average = average / linecount`
`balance -= payment`	is equivalent to	`balance = balance - payment`
`population *= growth`	is equivalent to	`population = population * growth`

Integer arithmetic. Any numerical calculation between two integers or that is stored into an integer variable will produce an integer result. Any fractional parts of a result are thrown away, not rounded off. This is called **truncating** towards zero. Thus, $-19/10$ would become -1 and $32/3$ would become 10.

Functions. Python supports a wide array of functions, which are invoked on the expression side (right side) of an assignment statement. In addition to the useful set of built-in functions, Python supports a large set of "calculator-button" functions in the *math* module. All of these are invoked by having a function name followed by a list of function arguments in parentheses.

Built-In Functions for Math

float	Convert to floating-point number: `f = float('123')` gives `f` the value `123.0`
int	Convert to an integer: `i = int(123.8)` gives `i` the value `123` (note truncation)
max, *min*	Find the maximum or minimum of a list of numbers. `c=max(s)` returns the largest value in `s` where `s` is a list ojbect. `c=min(a,b,c)` returns the smallest value of individual numbers `a`, `b`, or `c`.
abs	Absolute value (positive magnitude) of a number.

Math Module Functions. Although the math module is intrinsic to Python, these functions are not accessible to a program by default. To make use of one of the "calculator-button" functions, include an **import** statement to the `math` module. For example, to do a distance calculation from latitude and longitude, you need sines, cosines, and arccosines. Begin your program with

```
from math import cos, sin, acos, pi
```

Alternatively, the entire math library can be brought in with

```
import math
```

after which any reference to the functions requires the library name, as in

```
k = math.cos (a * math.pi / 180.0)
```

A library that will be used many times or that has a long name may be shortened to an alias

```
import math as m
```

allows the previous example to be shortened to

```
k = m.cos (a * m.pi / 180.0)
```

The arithmetic inside the argument of `cos` above is a reminder that in most computer programming languages, trigonometric functions work exclusively in **radians**. Arguments into the regular trigonometric functions must be in radians, not degrees. Results from inverse trigonometric functions will return in radians. To shorten the arithmetic, it is common to create a degrees-to-radians multipler equal to $\pi/180$:

```
degrad = pi/180.0
```

If `a` is an angle in degrees, then we can then calculate its cosine from `cos(degrad*a)`.

`sin`, `cos`, `tan`	Regular trigonometric functions, arguments must be radians.
`asin`, `acos`, `atan`	Inverse trigonometric functions, results will be radians.
`log`, `log10`	Logarithms, `log` is natural log and `log10` is common log. Argument must be a positive number.
`exp`	Exponential, e^x.
`sqrt`	Square root, $\sqrt{x}$.
`pi`, `e`	Constants, with the values $\pi = 3.141592\ldots$ and $e = 2.718281\ldots$.

Lists and Arrays

List and Array element references. References to a list element are made using square brackets. The first element is called the 0 element because it is 0 from the beginning (think of it as offset, rather than position). A **slice** of a list is a range of elements separated by a colon. Using the `c` list defined under list arithmetic we can make two new objects,

```
c = [ 1, 2, 4, 5, 6 ]
d = c[4]
e = c[0:2]
```

makes `d` an integer with value 6 and `e` is a list containing `[1,2]` (the last position is given as one *past* the last element included). The *`len`* function (see below under Built-In Functions) can be used to determine how many values are in a list: `len(c)` is 5 in the example shown here. Positions used in a slice can be integer variables. For example, the following sets `g` to be a list containing all but the last element of the previously existing list `h`:

```
k = len(h)
g = h[0:k-1]
```

Built-In Functions for Lists.

`list`	Create a defined *`list`*-type object from a sequence of lists or other objects. `u = list([ 12.0, 3, 'Ferrule'])` makes `u` into a list of three objects. `v = list([ a, b, c])` will make `v` into another list of three objects. In this case, the *values* of `a`, `b`, and `c`, not the object names, will be in in the list.
`range`	Returns a list of integers in specified range, with default starting point 0 and stride 1. *`range`*`(9)` returns `[0,1,2,3,4,5,6,7,8]` *`range`*`(2,9)` returns `[2,3,4,5,6,7,8]` *`range`*`(2,9,3)` returns `[2,5,8]`
`len`	How many numbers are in a list? *`len`*`(`*`range`*`(2,9,3))` produces a value of 3.
`sum`	Adds up the numbers in a list of numbers. Return value will be a single number.

List Methods. The following methods are built-in to Python and work with all kinds of lists, including nonnumeric lists, but are most associated with numeric list (array) manipulation. The sense of the descriptions below is the `L` already is defined and given values as a list, and the **method** applied to it modifies the size or arrangement of `L` or returns some information about the contents of `L`

`L.`*`count`*`(x)`	Count how many times `x` is in `L`.
`L.`*`index`*`(x)`	Gives the index (offset position) of the first occurence of `x` in `L`.
`L.`*`append`*`(x)`	Adds an element containing `x` to the end of `L`.
`L.`*`extend`*`(t)`	Adds elements containing an entire list `t` to the end of `L`.
`L.`*`insert`*`(i,x)`	Adds elements `x` to `L` at index position `i`.
`L.`*`remove`*`(x)`	Removes the first occurrence of an element containing `x` from `L`.
`L.`*`pop`*`(i)`	Removes the element at position `i` from `L`. If `i` is not specified, the last element is removed.
`L.`*`sort`*`()`	Sorts, in place, the values of `L`, lowest to highest.
`L.`*`reverse`*`()`	Reverses, in place, the order of values of `L`. The combination of `sort` and `reverse` can do highest-to-lowest sort.

Numeric Python Arrays

The Python universe contains a wide range of "add-on" modules that are not part of the default Python setup but that can be obtained and installed, often for free. For arrays, we will be using **Numeric Python** which is imported via the `numpy` module. Documentation for `numpy` can be found at

`http://docs.scipy.org/doc/numpy/reference/routines.html`

In the examples that follow, we assume usage of

import `numpy` **as** `np`

after which references to the functions use the alias name `np`, as in

```
x = np.zeros(50)
```

Alternatively, restricted parts of a library can be brought in to a program, with the **from** keyword:

from `numpy` **import** *`zeros, arange`*

allows you to subsequently create an array of 50 zeros with

```
x = zeros(50)
```

Arrays are a modification of **list** objects that use only numerical information using numbers all the same type (integer or floating point). Individual numbers within the array, or subsets of the array, can be accessed by the combination of the array name and index numbers that point to individual positions or to slices. For example, instead of making 365 variable names to list all the temperatures in a year, we might fill an array `temperature` that holds 365 numbers, and refer to the temperature on the 15th of February as `temperature[45]`, the 46th temperature in the array.

The functions below are mostly used to *create* new array objects. The underlined argument names are optional, such that default values are used if the argument is not included.

`arange``(`start`, stop, `step`)`

Create a numeric array of evenly spaced values. Default starting point is 0 and default step is 1.

```
x = np.arange(20)
```

makes `x` into a list of 20 integer numbers (type inferred from the type of the argument) whose values are 0 through 19. (I.e., the arguments act in the same manner as the built-in list function *`range`*.)

`zeros``(shape, `dtype`)`

Create an array where every element is initialized to zero. Dimension `shape`, which is an integer size for a one-dimensional array and may be a list of integers for a multi-dimensional array. Default data type is `float`.

`ones``(shape, `dtype`)`

Create an array with elements initialized to ones; other arguments as with *`zeros`*.

empty(shape, dtype)

Create an array with elements uninitialized; other arguments as with *zeros*.

zeros_like | *empty_like* | *ones_like*(a)

Create an array, initialized to zeros or ones, or uninitialized, with the size and data type of the previously defined array a.

```
y = np.zeros_like(x)
```

using x from the *arange* example, makes y into a list of 20 integer zeros (because x was 20 integers).

copy(a)

Create a copy of a.

array(object, dtype)

Create an array from the array-like object given; usually used to directly construct an array from a list of elements.

Two-dimensional Arrays

All the NumPy arrays discussed in the previous section are one-dimensional, in that they look like a simple list of numbers and have a single index in square brackets to pick out individual values. If we give the array-creation functions a *list* of numbers instead of a single size number, we will create a multi-dimensional array. In practice within this course, we do not need more than two dimensions.

```
w = np.empty ( [n,m] )
```

defines `w` as a two dimensional array of `n` rows and `m` columns, and any reference to it would require two index numbers in square brackets. For example if we were to have a set of `n` latitudes and `m` longitudes forming a grid and we wanted to have a sinusoidal projection of the coordinates, we could do

```
xg = np.empty ( [n,m] )
yg = np.empty ( [n,m] )
for k in range(n):
    for l in range(m):
        xg[k,l] = lon[l] * math.cos(lat[k])
        yg[k,l] = lat[k]
```

Another example is the code required to read a file of numbers into a two-dimensional data array. As an example, imagine you a climate data file with 507 lines, each representing a different weather station. On each line is an integer station number followed by 12 monthly average temperatures, all space-separated. Putting those data into an array might look like this:

```
tdata = open('datafilename', 'r').readlines()
station = np.empty ( 507,dtype=int )
temp = np.empty ( [12,507] )
for k in range(507):
    dlist = tdata[k].split()
    station[k] = int(dlist[0])
    for l in range(12):
        temp[l,k] = float(dlist[l+1])
```

The inquiry function *len*, when used on a multidimensional array, produces only the first dimension. The NumPy inquiry function *shape* produces a list of integers that gives all the dimensions. Using the preceding example:

```
print np.shape(temp), len(temp)
```

will produce as output:

```
(12, 507) 12
```

Input/Output

A common reason to need programming is to be able to read data files, calculate simple summary values from the data, modify the data, and rewrite data in a form suitable for input to another program, such as GIS, a spreadsheet, or a graphics package. In Python, these tasks usually require the ability to read or create files on the attached storage systems. To access a real-world file, create a file object to point at the file, using the built-in *open* function.

file_object_name = *open*(*file_path_and_name*, *access_mode*)

where *file_object_name* is the name a programmer makes up to refer to the file object in the rest of the program, *open* is a built-in function, *file_path_and_name* is a character string (either a character-string object name that has been given a value, or often a literal character string) containing the name and location of the file on the disk systems available to the program, and *access_mode* is typically either `"r"` for a read-only (input) file and `"w"` for a writeable (output) file.

Examples, using a Windows file reference and a Macintosh/Unix file reference respectively:

```
infile = open( "D://ds/census/counties_2000.csv", "r" )
outfile = open( "/Users/johan/census/pchange.txt", "w" )
```

After those two declarations, `infile` can be used as the file object name with a *readlines* function, and `outfile` can be used as the file object in a **print** statement.

The first argument of the *open* function is an *operating-system dependent* file reference that uses the same file name and path elements as you would see in the Finder, Explorer, or Unix shell, whereas `infile` and `outfile` are programmer-defined local Python object names that refer to these files for the remainder of the Python program.

Writing to the terminal screen To write a message to the screen, use the **print** command with a list of literal character strings and object names. The literal character strings will be reproduced literally, and the object names will be replaced with their values.

```
a = "North Dakota"
b = 25.3
print "Average temperature in", a, "is", b
```

will produce Average temperature in North Dakota is 25.3, replacing variable names with values and deleting commas between output list items and quotation marks around strings.

Write to a file. To write a line in file, just add a file object name reference to a **print** statement using `>>` to indicate a direction of flow.

```
print >> outfile, a, b, temperature
```

in which `a`, `b`, and `temperature` are object names of variables whose values will be written to a line of the file. The `>>`, made by typing a greater-than sign twice, is called a chevron.

The previous example will write a line containing the values of its three variables along with an **end-of-line** marker, so that the next **print** statement will start on a new line. A feature encountered by mistake more often than actually needed is to put another comma at the end,

```
print >> outfile, a, b, temperature,
```

which prevents an end-of-line mark and means the next **print** will continue on the same line. (The exact form of the end-of-line marking is different between Macintosh and Windows systems, causing occasional problems in transfering a text file from one system to another.)

Formatting output. In the **print** statements above, the information sent to the screen or file does not include the Python object names `a`, `b`, or `temperature`, but rather the values contained in those objects, turned into visible character strings. The conversion of binary information actually contained in the objects into character strings a human can easily read is called **formatting**, and the system has some very good default procedures for this, which will be used in the above examples.

When the output produced by the default formatting is not acceptable, such as when a floating point number is written with decimal places beyond the actual significant digits, string formatting can be used to take complete control of horizontal spacing and how many digits are included in a number. This is actually a form of character-string "arithmetic" and thus is discussed below in the String Handling section.

Reading from the keyboard. To read a value from the keyboard, use the built-in function *raw_input*. The following will prompt a user with the string inside, go down one line (the `\n` requests a linefeed), and then wait for the user to type something and press return. Whatever the user types will become the value of `dd`.

```
dd = raw_input( "Enter a degree-day threshold\n" )
```

String Handling

The most common data object in Python is a string. Output to files and the screen consists of strings; numbers are converted into string representations before printing. Input from files consists of line-long strings that can be *split* into component strings and then converted to numbers if necessary.

The *split* method is used extensively to break strings into more useful components. Consider a long line of data, probably read from a file but set directly in this example:

```
line = "2854 2014 500864 59.2 New Castle Delaware"
```

`line.split()` separates the line into its components based on the default **separator**, which is any string of blank spaces. Applying `p=line.split()` means that `p` becomes the list `[ "2854", "2014", "500864", "59.2", "New", "Castle", "Delaware" ]`. The individual elements can be accessed using **slice** notation, with element positions **starting from 0**: `line.split()[1]` is `"2014"`, the *second* item in the list.

Items produced by *split* are still character strings. Character strings of digits can be converted to numbers with a function.

```
q = int( line.split()[0] )
r = int( line.split()[2] )
s = float( line.split()[3] )
```

will produce `q` with integer value 2854, `r` with integer value 500864, and `s` with floating-point value 59.2.

The county name is a problem: `"New"` and `"Castle"` are two separate list elements. They can be easily put together because the `+` operator is **overloaded** to work as a **concatenation** operator for strings, as in

```
countyname = line.split()[4] + " " + line.split()[5]
```

but this will only work in general when every county has two-word names. Space-delimited files can be problematic if place identifier names are included; nearly every kind of place identifer has some two-word names, from Kennett Square to North America.

A separator character other than space may be specified as an argument in *split*. In a **comma-separated values** (CSV) file, the previous line of input might look like

```
line = "2854,2014,500864,59.2,New Castle,Delaware"
```

and we may use `a = line.split(",")` to produce the list `[ "2854", "2014", "500864", "59.2", "New Castle", "Delaware" ]`, with the obvious improvement in handling of the county name. Other potential applications of a different separator character should be clear from these examples, where the resulting value is shown at right.

```
line = "10/29/2007 11:43:09"
date = line.split()[0]                  "10/29/2007"
time = line.split()[1]                  "11:43:09"
hour = int(time.split(":")[0])          11
day = int(date.split("/")[1])           29
```

Invisible characters. Every character set has some characters that indicate formatting or actions. To put these into character strings, use them in *split* statements, or send them to output, we need a way of making them visible in code. All such characters in python are indicated using the backslash character as an "escape" character, followed by a letter that has some mnemonic sense. Here are a few of the most useful ones.

`\f` Form feed (new page)

`\n` New line

`\r` Carriage return

`\t` Tab

`\b` Backspace

`\'` `\"` Quote marks as characters (used when single or double quotation marks are part of a string and not delimiters indicating the beginning and end of a string).

`\\` Backslash (used when a backslash is actually needed as a character rather than an escape indicator).

Formatting numbers into strings. Formatting refers to the conversion of internal binary representation of a number into a sequence of digit characters that a human can read. Any **print** statement must perform this conversion (see the Input/Output and Files section), and often the default formatting used by the system is adequate. However, format control characters can be used to control spacing between items, how many significant figures to include on a number, how many digits after a decimal point, and so on. Each item of an output list can have a string conversion specifier whose form is a percent symbol, %, followed by a conversion character that indicates what kind of output is expected, optionally followed by a decimal point and a number to further modify the conversion. The conversion characters include

`d`	Signed decimal integer	`u`	Unsigned decimal integer
`f`	Floating point, decimal form	`E`	Floating point, exponential form
`c`	Single character	`s`	Character string

The following block of code shows some usage examples. If a number immediately follows the %, it is the width (in spaces) of the field in which the object will be written. A second number following a decimal point controls the number of decimal places to the right of the decimal point for ***float*** and number of nonblank digits, using leading zeroes for ***int***. Blank spaces in the output are noted with ␣.

Python statement	*Output*
`a = 30.0; b=3.1416; c=5.67E-3`	
print `a, b, c`	`30.0␣3.1416␣5.67e-03`
print `'%8.2f %8.2f %.4f' % (a, b, c)`	`␣␣␣30.00␣␣␣␣␣3.14␣0.0057`
`i = 2; j=-3; k=40`	
print `i, j, k`	`2␣-3␣40`
print `'%4d' % i, '%4d' % j, '%4d' % k`	`␣␣␣2␣␣-3␣␣␣40`
print `'%4.3d'% i, '%4.3d' % j, '%4.3d' % k`	`␣002␣-003␣␣040`

Both the **print** statement and the string formatting procedures are changed in Python 3, so working at learning a lot about the ugly % syntax is a waste of time.

Python Block Structures

The next sections deal with **control structures** which allow statements to be repeated by looping structures or skipped by branching structures. Unique among computer languages, Python uses **significant indenting**, in which a loop statement (**for**) or branching statement (**if**) applies only to those statements that are indented underneath it. In the first **for** example shown below, the three indented statements are repeated over and over, once for each line of the file. The last **print** statement is executed only once. The *only* distinction between the two **print** statements that causes one to be excecuted repeately and one only once is the horizontal position on the lines.

Loops and Iteration

Iterated *for* loop. To loop one-item-at-a-time through data of various types that are arranged in sequence, commonly in a list, use a **for** loop.

for *sub_object* **in** *iterable_object*:

Iterable objects are those which consist of an ordered set of subobjects that can be gone through in sequence. There are several classes of iterable objects in Python, but the **list** is the simplest to understand, in which iteration through the list involves going through the individual items in the list one at a time. Other objects can be turned into lists: a **file** becomes a list of lines (each of which is a character string) with the *readlines* method. The *split* method turns a character string into a list of shorter character strings by specifying a **separator** character, and a character string can be converted to a list of its individual characters using the *list* function.

The other *iterable_object* we will use is lists of numbers generated by the *range* or *arange* functions. The *sub_object* will be defined by the type of the iteratable object: if the *iterable_object* is a list or array, then the *sub_object* will be same class of object as the list elements.

In a first example, we read and sum the numbers in a file.

```
infile = open("2010_weather.txt", "r")
outfile = open("2010_precip.txt", "w")
obs_list = infile.readlines()
precipsum = 0.0
for obs in obs_list:
    precip = float(obs.split()[2])
    precipsum += precip
    print >> outfile, precip
print "Total precip in ", len(obs_list), " days is", precipsum
```

Note the use of the *len* function to determine how many items there were in the list of lines in the file – the *len* function can be used on arrays, any kinds of lists, and on characters strings.

The next example does some of the same things as the previous one, using an array instead of an output file. Note the usage of *readlines* that skips defining a file object.

```
obs_list = open("2010_weather.txt", "r").readlines()
n = len(obs_list)
for k in range(n):
    obs = obs_list[k]
    precip[k] = float(obs.split()[2])
precipsum = sum(precip)
print "Total precip in ", n, " days is", precipsum
```

Both of these examples read the data file, find the precipitation values on each line, and print a sum and number of days to the screen. The difference is in where the precipitation data reside when the block is finished. The first example creates a new data file for the extracted precipitation data. The data in the file are no longer available to this program without reading the output file.

The second example loads all of the precipitation data into an array. The `precip` array can be iterated through again, subjected to other array methods, or fed into graphics routines for plotting.

Branches and If

Many file processing tasks involve deciding whether a line should be included in output or not, or deciding how to process a number based on its size. These **logical decisions** are handled in Python with **if** statements.

```
if logical_expression_1:
    ⋮        # do this block when logical_expression_1 is true
elif logical_expression_2:
    ⋮        # do this block when logical_expression_2 is true
else:
    ⋮        # do this block when neither logical expression above is true.
```

As many **elif** blocks as needed, or none, can be included, and one does not need to include the **else** block as a "default" unless needed.

Logical expressions are comparisons of one variable to another or to a constant, using **comparison operators**:

`==`	equals	`<`	less than	`<=`	less than or equal to
`!=`	not equals	`>`	greater than	`>=`	greater than or equal to

and we also have `and` and `or`, with their symbolic logic meanings, to combine two logical expressions.

In simple data analysis problems, the most common **if** structure is to test information from a list item inside a **for** loop. Here is a data-trimming program that selects points that fit inside given latitude/longitude box and creates a new file of the subset, while also counting how many points are included in the subset.

```
infile = open("all_stations.csv", "r")
outfile = open("box_stations.txt", "w")
ltmin = 30.0; ltmax=49.0; lgmin=-90.0; lgmax=-60.0
stations = infile.readlines()
outcount = 0
for station in stations:
    lg = float(station.split(",")[0])
    lt = float(station.split(",")[1])
    elev = float(station.split(",")[2])
    if lg >= lgmin and lg <= lgmax and lt >= ltmin and lt <= ltmax:
        outcount += 1
        print >> outfile, lt, lg, elev
print outcount, " points included in output."
```

For a second example, split a weather data set into snow and rain based on the simple (inaccurate) assumption that precipitation falling in temperatures above 0°C is rain and is otherwise snowfall. Initialization of variables and file objects is left off to just show the loop and branch structure.

```
for day in days:
    temp = float(day.split()[0])
    precip = float(day.split()[1])
    if temp >= 0.0:
        raincount += 1
        print >> rainfile, temp, precip
    else:
        snowcount += 1
        print >> snowfile, temp, precip
print raincount, " rain days and ", snowcount, " snow days."
```

Nesting. Note that the previous examples have two levels of indenting: one for the **for** loop and a second level for the **if** block. This is more deeply demonstrated in the following example, which sorts sieve measurements based on size classes, using loops through lists of two different lengths. An additional item needed for this loop to work is a **break** statement, which terminates the innermost loop that it is inside.

```
buckets = open("sediment.dat", "r").readlines()
sizeclasses = [ 0.004, 0.063, 1.0, 16.0, 256.0 ]
classmasses = [ 0.0, 0.0, 0.0, 0.0, 0.0 ]
classcount = [ 0, 0, 0, 0, 0 ]
nclasses = len(sizeclasses)
for bucket in buckets:
    size = float(bucket.split()[0]
    mass = float(bucket.split()[1]
    for k in range(nclasses):
        if size < sizeclasses[k]:
            classmasses[k] += mass
            classcount[k] += 1
            break
for k in range(nclasses):
    print sizeclasses[k], classemasses[k], classcount[k]
```

While

A ***while* loop** is an alternative to **for** that controls the exit via a logical test rather than a count or iteration. The simplest form of the block is

while *logical_expression*:
    statements to run as long as logical_expression stays true

The following example repeats the same function as the outer loop section of the previous counted **for** example, using **while** instead of the inner **for**.

```
for bucket in buckets:
    size = float(line.split(",")[0]
    mass = float(line.split(",")[1]
    k = 0
    while size > sizeclasses(k):
        k += 1
    classmasses[k] += mass
    classcount[k] += 1
```