# Contents

# Chapter 1

# Getting Started

## 1.1 Basic Skills

To do the labs and the problem sets,you will need some basic computer skills. I will outline these briefly here. The instructions below assume that the exercises will be done using what I'll call the *default setup*. In the default setup, the course software and the necessary datasets reside on a centralized server; the student logs on to the server from a workstation that supports the ssh protocol and the X windowing system. The X window system is needed to allow a remote server to write graphics (e.g. a plot, or a graphical user interface) to the screen of the local workstation. The specific instructions below apply most closely to Unix workstations. The necessary skills for using the default setup are:

- Logging in to a Linux server from a workstation on the network.
- Setting things up for the Linux machine to display its graphics on the workstation you are sitting at, using the X windowing system.
- Working with Linux directories and files (commands `cd,ls,mv,rm,mkdir`).
- Starting up the Python interpreter and using the Python Integrated Development Environment, `idle`.

### 1.1.1 Using xterms and logging in to the server

The software that you will be using, as well as the data you will be looking at, resides on a server running the Unix operating system. In the examples, we will suppose that the server is `climate.myUniversity.edu`; your own server will have a different name, which will be provided by your instructor. To use the software, you will need to log into `climate`, which you can do from any machine anywhere in the world, as long as the machine has an `ssh` program. The first step is to get an X terminal window ("xterm" for short) on the screen of the workstation at which you are sitting. If your workstation is a Unix computer the standard window you get when you request a "shell" or a "terminal" window is already an xterm, assuming the system has been started up into a graphical user environment, as is generally the case these days. To get a new xterm, you just need to click on the appropriate icon on the desktop. The specific icon varies somewhat from system to system, but will generally look like a scallop shell or a computer screen.

Macs running OSX are actually running a form of Unix, but the default graphical interface does not use the X windowing system. This will be less confusing if you recall that the "X" in "OSX" is actually pronounced "10". The standard terminal, or shell, window you get with the OSX terminal tool is *not* an xterm. While you can

issue Unix commands and log onto remote systems for text-based applications by issuing the ssh command in this window, the OSX terminal window does not handle graphics. Further, OSX does not come with the X windowing system installed by default. Fortunately, Appole provides an excellent implementation for X on OSX, which can be installed from the system install disk. If you have your own OSX Mac, or have administrative privileges for some OSX Mac you can use, you can install `X11` yourself very easily. All Macs set up for this course should, in principle, already have `X11` installed. To get an xterm, you just click on the `X11` icon in the toolbar and wait for X to start up. The default windows X puts up on the screen are all xterms. You can make a window go away by typing `ctrl-d` in the window (meaning hold down the `ctrl` key and type `d`. If you want a new xterm, just type `xterm&` in any existing xterm window, or choose `terminal` from the `Applications` menu, and a new one will pop up. Your instructor can show you how to move and resize windows or turn them into icons.

Once you have an xterm on your screen, click the mouse in its window to activate it. You are now ready to log in to the course server. If you happen to be on a Unix workstation with the course data and software installed locally, you can just skip the login step. This is one of the beauties of X and Unix – the system doesn't really care which computer is actually doing the calculation. This remark applies equally to OSX Macs, provided that Unix versions of the course software have been properly installed.

Now let's assume that you need to log on to the course server. You'll need an account to go further: a userid and a password. If you already have an account on `climate`, you can use that. If not,you can get one of the pre-assigned accounts from the TA. Once you have this data, you can log in. To log in, just issue the command `ssh -X -l<USERID> climate.myUniversity.edu` from an xterm, where `<USERID>` is the userid for the course account. Then give your password at the prompt. The `-X` option tells the server to forward graphical commands to the local X windowing system for handling. On many Linux systems, this option is turned on by default, but it never hurts to include it explicitly.

### 1.1.2 About the Python Shell and idle

Python is an interpreted language, which means you just type in plain text to an *interpreter*, and things happen. There is no compilation step, as in languages such as `c` or `FORTRAN`. To start up the Python interpreter,just type `python` from the command line on `climate`. You'll get a prompt, and can start typing in python commands. Try typing in `2.5*3+5.` and see what happens. To exit the Python interpreter, type `ctrl-d`.

Eventually, you'll probably want to put your Python programs, or at least your function definitions, in a file you create and edit with a text editor, and then load it into Python later. This saves you having to re-type everything every time you run. The standard Unix implementation of Python provides an *integrated development environment* called `idle`, which bundles a Python interpreter window with a Python-aware text editor. To start up `idle`, log in to the server from an xterm and type `IDLE`. You will get a Python shell window, which is an ordinary Python interpreter except that it allows some limited editing capabilities. The real power of idle comes from the use of the integrated editor. To get an editor window for a new file, just choose `New Window` from the `File` menu on the `Python Shell` window. If you want to work with an existing file instead, just choose `Open` from the `File` menu, and pick the file you want from the resulting dialog box. You can type text into the editor window, and cut and paste in a fashion that will probably be familiar to most computer users. You can have as many editor windows open as you want, and cut and paste between them. When you are done with your changes, select `Save` or `Save as` from the `File` menu of the editor window, and respond to the resulting dialog box as necessary. Once you have saved a file, you can run it by selecting `Run module` from the Run menu.

You can actually use the integrated editor to edit just about any text file, but it has features that make it especially useful for Python files. For example, it colorizes Python key words, automatically indents in a sensible way, and provides popup advice windows that help you remember how various Python functions are used. As an exercise at this point, you should try creating and saving a short note (e.g. a letter of gratitude to your TA), and then try opening it up again in a new editor window. To exit from `idle` just choose `Exit` from the `File` menu of any window.

An especially useful feature of the `idle` editor is that it allows you to execute the Python script you are working on without leaving the window. To do this, just choose `Run Script` from the `Edit` menu of the editor window. Then the script will run in the Python shell window. When the script is done running, you can type additional Python commands into the shell window, to check the values of various quantities and so forth.

`IDLE` has various other powerful features, including debugging support. You can manage without these, but you should feel free to learn about and experiment with them as you go along.

Once you have written a working Python script and saved it,say, as `MyScript.py`, you can run it from the command line by typing `python MyScript.py`. There is no need to start up `IDLE` just to run a script.

### 1.1.3 Running Python Locally

Note that many of the Python-based exercises given in the problem sets do not need the data stored on `climate`, or the special Python extension modules written for this course. If you have a computer of your own, you can download your own copy of Python from the web site `python.org`. Implementations are available for Macs, Linux and Windows PC's. The MacPython implementation, available for both OS9 and OSX Macs provides an excellent integrated development environment that in some ways is superior to `IDLE`. You can use your own stand-alone machine for any of the exercises that need only straight Python programming using the standard modules. You can also use your own machine for any exercises involving reading and writing of text data files, if you first download any needed data from climate to your own machine. Also, any Python extension modules that are written as ordinary human-readable Python scripts (e.g. `phys.py` ) can be just downloaded and put in your `python` directory, regardless of what kind of machine you are using. However, compiled extension modules, with names like `veclib.so` need to be compatible with your specific hardware and Python implementation.

In the rest of this workbook, when we say "Start up the Python interpreter," the choice is up to you whether you use the simple command line interpreter or `idle`, or perhaps some other integrated Python development environment you might have (e.g. MacPython). For results that produce graphics, and for the use of `idle`, you must be connected to Python in a way that can display graphics on your screen (e.g. via an xterm). You won't be reminded of this explicitly in the text. Exercises that don't produce graphics can be done over any kind of link. "Write and run" a script could mean that you enter it using your favorite editor and run it from the command line, or it could mean using `idle`.

In general, I have tried to avoid referring to implementation-dependent details in the rest of this Workbook.

## 1.2 Fun with Python

This is a very simple lab designed to help you get used to programming with Python. Throughout this and the rest of the Python labs, it is expected that you will try out all the examples in the Python interpreter window, and make up additional examples on your own until you feel you understand the concept being introduced. For the most part, you won't be bothered with any further reminders of this expectation.

First, start up the Python interpreter. For this lab, you can type your input directly into the interpreter, if you wish. As you begin to do more complex programs, however, you will want to write your programs using a text editor, and then save them before running. This way, you won't have to retype everything when you need to correct a mistake in just one or two lines, and you can re-run the program or a modification of it very easily. Although none of the exercises in this lab are complex enough to really require the text editor, you can use this lab as an opportunity to become familiar with the use of the `idle` editor.

### 1.2.1 Basic operations

Once you're at the Python interpreter prompt, try some simple statements using Python like a calculator, e.g.:

```
52.5*51.2+37.
a = 7.
b=10.
a/b
a*b
a = 7
b = 10
a*b
a/b
2**1000
1717%3
```

and so forth. This is so nice,you'll probably want to load Python onto your laptop and use it in place of a pocket calculator, especiallly once you learn how to import the standard math functions into your Python world. These examples illustrate the use of floating point numbers, multiplication and addition ("*", "+" and "/) assignment to variables, integers, and exponentiation "**". The final example illustrates the use of the "mod" operator, `%` which is a binary operator applied to integers. The expression `n%m` yields an integer whose absolute value is less than `m`, which is the

result of subtracting off the maximum possible multiples of `m` (if you are familiar with clock arithmetic, this is the operation that turns regular arithmetic into clock arithmetic *modulo* $m$). The assignments to the variables $a$ and $b$ illustrate that Python is not a *typed* language. You do not have to declare the variables as being of a certain type before you use them. They are just names, which are used as long as necessary to refer to some value. This extends not just to numbers, but to all the object which Python can deal with, including arrays, lists, functions, strings and many other entities which will be introduced shortly. In the example above, first a and b are floats, and behave like floats on division. Then they are integers, and behave like integers. The last line illustrates the exponentiation operator, denoted by "**". The large number you get as a result has an "L" tacked on the end, signifying that the result is a *long* integer, which can have arbitrarily many digits (until you run out of memory). Python automatically creates this type of integer whenever necessary. The standard Python floating point number has double precision, though Python extensions are available which allow you to specify arbitrary precision for floats as well.

Python also has floating point complex numbers as a native data type. A complex number with real and imaginary parts $a$ and $b$ respectively is written as $a + bj$. All the usual operations apply. After setting $z = 7.5 + 4.j$ try $z + 1$, $z * z$, $1/z$, $z * *1.5$ and $z * *z$. If you need to make a complex number out of two real variables, say `x` and `y`, the easiest way is to use the `complex` function, e.g. `z = complex(x,y)`. Python does not have complex integers (known as *gaussian integers* to mathematicians) as a native data type, but you will learn how to define these, and virtually any other specialized type you need, in Section 1.4.1

### 1.2.2 Lists, tuples and strings

Tuples and lists are among the most basic and versatile data structures in Python. Lists contain any kind of data at all, and the elements can be of different types (floats, int, strings, even other tuples or lists). Many functions return tuples or lists. Try out the following examples in the interpreter

Heres an example showing two ways defining a list and getting at an element:

```
a = [1,'two']
a[0]
a[1]
b = [ ]
b.append(1)
b.append('two')
b[0]
```

```
b[1]
```

In the second part of the example, note that a list, like everything else in Python, is in fact an "object" with actions (called "methods") which you can perform by appending the method name to the object name. The mod operator is useful for making circular lists, which begin over from the first element when one reaches the end. For exammple, if `a` is any list, `a[i%len(a)]` will access the list as if it were bent around in a circle. The same trick works for any integer-indexed object.

Python distinguishes between lists and tuples. These are similar, except that lists can be modified but tuples cannot. Lists are denoted by square brackets, whereas tuples are denoted by parentheses. The above example is a list rather than a tuple. You can define a tuple, but once defined you cannot modify it in any way, either by appending to it or changing one of its elements. There are a very few cases where Python commands specifically require a tuple rather than a list, in which case you can turn a list (say, `mylist`) to a tuple by using the function `tuple(mylist)`.

Strings are also objects, with their own set of useful methods. For example:

```
a = 'Five gallons of worms in a 3 gallon barrel!'
a.split()
b = a.split()
print b[0],b[3],b[4]
```

Note that the split() method returns a list,whose elements are strings. By the way, in Python, you can use either single quotes or double quotes to enclose a string, as long as you use them consistently within any one string. There is no difference in the behavior of single quoted and double quoted strings. For strings, the `+` operator is concatenation, i.e. `a+b` is the concatenation of the two strings `a` and `b`.

It is very often useful to be able to build strings from numerical values in your script. This need often arises in formatting printout of results to look nice, or in generating filenames. Suppose `a = 2` and `b = 3`. Then, the following example show how you can insert the values into a string:

```
s = '%d + %d = %d'%(a,b,a+b)
print s
```

note that the "input" to the format string must be a `tuple`, not a list; recall, however, that if `L` is a list, the function call `tuple(L)` will return a tuple whose elements are those of the list input as the argument. If the tuple has only one element, you can leave off the parentheses. The format code `%d` (or equivalently `%i`) converts an integer into a string. You use `%f` for floating point numbers, and `%e` for

floats in scientific notation. There are other options to these format codes which give you more control over the appearance of the output, and also several additional format codes.

Now make up a few examples of your own and try them out.

### 1.2.3 Modules

To do almost any useful science with Python, you will need to load various libraries, known as "modules." Actually, a module can be just an ordinary Python script, which defines various functions and other things that are not provided by the core language. A module can also provide access to high-performance extensions written using compiled languages.

To make use of a module with name `myModule`, you just type: `import myModule`. Members of the module are accessed by prepending the module name to the member name, separated by a ".". For example, if `myModule` contains the constant `r_earth`, and the function `sza`, these constant is accessed using `myModule.r_earth` and the function is evaluated at `t` using `myModule.sza(t)`. If you don't need to keep the module's members separate, you can avoid the need of prepending the module name by using `from myModule import *`.

The standard math functions are in the module `math`, and you make them available by typing `import math`. To see what's there, type `dir(math)`; this works for any module. Now, to compute $sin(\pi/7.)$ for example, you type `math.sin(math.pi/7.)`. To find out more about the function `math.sin`, just type `help(math.sin)`. If you don't like typing `math.sin`, you can import the module using `from math import *` instead, and then you can just use `sin`,`cos`, etc. without the prefix.

### 1.2.4 Getting help

Python has extensive built-in help functions, which make it possible to learn new things and avoid programming errors without frequent recourse to manuals. Given that so much of Python is found in various language extensions the Python community has written, the availability of embedded documentation is beholden to the good behavior of the programmer. Python fosters a culture of good behavior, and tries to make it easy for developers to provide ample help and documentation integrated with the tools they have developed.

The main ways of getting help in Python are the `help()` and `dir()` functions. For example, you have learned about the `split()` method that is one of the methods available to strings. Suppose you didn't know what methods or data attributes went

along with a string, though? Rather than going to a handbook, you can use the `dir()` function to find out this sort of thing. For example, if `a` is a string, you can type `dir(a)` to get a list of all its methods, and also all its data attributes (e.g. its length). Then, if you want to know more about the `split()` method you can type `help(a.split)` (Warning: don't type `help(a.split())`, which would look for help items on the words in the content of the string!). Both strings and lists have many useful and powerful methods attached to them. Many of these will be illustrated in the course of the examples given in the rest of this Workbook, but you are encouraged to explore them on your own, by finding out about them using `dir()` and `help()`, and then trying them out.

So when in doubt, try *help* and *dir*. One or the other will give you some useful information about just about anything in Python. If the system you are working on has the Python HTML documentation files installed, you can even get help on Python syntax and Python key words online. For example, to find out what the Python keyword `for` means, you just type `help("for")`.

Further, since Python is interpreted rather than compiled into machine language, if you have some Python programs written by somebody else, you can almost always "look under the hood" to see how they work. That is not generally possible with compiled languages where you often don't have access to the original source code.

### 1.2.5 Program control: Looping, conditionals and functions

Now we're ready for some more involved programming constructions. The basic technique for writing a loop is illustrated by the following example, which prints out the integers from 0 through 9:

```
for i in range(10):
     x = i*i
     print i,x
```

Note that in Python, indentation is part of the syntax. In the above example, the indentation is the only way Python has to identify the block of instructions that is being looped over. Indentation in a block of code must line up, and you need to be cautions not to confuse spaces and tabs. The use of indentation as a syntactic element in Python enforces code readability and reduces the need for special identifiers to terminate blocks.

The construct `range(10)` is actually shorthand for the 10-element list

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

In fact, one of Python's many charms is that a `for` loop can loop over the elements of any list at all, regardless of what the elements of the list may be. Thus, the following example sums up the length of four strings:

```
myList = ['bob','carol','ted','alice']
n = 0
for name in myList:
  n = n + len(name)
```

This can be very useful for looping over file names with data needing to be processed, or data arrays which need something done to them, and all sorts of other things that will occur to you once you get accustomed to the concept.

An alternate to looping over a list is to use the `while` construction, as in:

```
x = 1.
while x < 100.:
    print x
    x = 1.1*x
```

Now, for practice, write a loop to compute 52 factorial (i.e. 52*51*...*1). Note that Python automatically starts using long integers when it needs to.

In doing computations, it is typical that one needs to test for the satisfaction of a condition at some point before proceeding. For example, one might need to test whether a temperature is below or above freezing to decide whether to form ice or liquid water. Programming languages generally provide some conditional control to handle this situation, and Python is no exception. The following illustrates the use of an `if` block in Python:

```
if T < 273.15:
     print "Too cold!"
```

and an extended `if` block:

```
if T < 273.15:
     print "Too cold!"
elif T > 373.15:
     print "Too hot!"
else:
     print "Just right!"
```

An `if` block can have as many `elif` blocks as you need, and the conditional being tested can be anything that reasonably evaluates to a truth value. To distinguish from assignment, the equality relation is expressed by the symbol `==`. The symbols `<=` and `>=` have the obvious meanings. The exclamation point negates a relation, and Python also provides the operator `not` to negate the truth value of an arbitrary logical expression. For example `1 != 0` and `not (1 == 0)` mean the same thing. Compound expressions can be built up from the Boolean operators for "and" (`&`) and "or" (`|`, the vertical bar). Python also provides the keywords `True` and `False` for logical values, but regular integers 1 and 0 generally do just as well in conditionals.

The organization of almost any program can benefit from the subdivision of the labor of the program into a number of functions. This makes the program easier to debug, since functions can be tested individually. It also allows the re-use of code that is needed in many different places in the program. The basic means of defining a function is illustrated in the following example, which returns the square of the argument:

```
def f(x):
    return x*x
```

From the command line, you would invoke this function, once it is defined, by typing, e.g. `f(3)`.

Python can even handle recursion in functions. That is, functions can be defined in terms of themselves. As an example, a function to compute the factorial of `n` could be written:

```
def factorial(n):
   if n == 0:
      return 1
   else:
      return n*factorial(n-1)
```

Functions can return multiple arguments, as in:

```
def powers(x):
      return x,x*x,x*x*x
```

This returns a tuple containing the three values. It can be very nicely used with Python's ability to set multiple items to corresponding items of a tuple, using constructions of the form:

```
x1,x2,x3 = powers(2)
```

Python functions work only on a *copy* of the arguments. It is important to keep this in mind, since it means that any changes made to these arguments ("side-effects") do not affect the variable's value in the calling program. Try this:

```
def f(myValue):
     myValue = 0

x = 1
print x
f(x)
print x
```

In this example, `x` is unchanged because functions work only on a local copy of their arguments. However, if the argument is a name which points to the location of some data, the data pointed to can be modified in the function. This may seem somewhat arcane, but the following simple example modifying an element in a list should illustrate the general principle:

```
def f(myList):
     myList[0] = myList[1]

L = [1,2,3,4,5]
f(L)
print L
```

If you want to replace the list with a completely new list, based on the old one, the right way to do this is to return the new list, rather than doing anything to the argument:

```
def bump(myList):
    newList = [ ]
    for item in myList:
        newList.append(item+1)
    return newList
```

Then, if the user really intended to replace the old list, he or she would use `L = bump(L)`

Often, the evaluation of a function will require a number of constants or parameters which you might not need to change very often, or which are common to a great many different functions. These could always be added to the parameter list. If there are many of them, that could become tedious. Soon you will learn how

to create objects which provide a way to package data in a way that allows you to keep the argument list of functions under control. There is yet another technique, though, that can be useful if used with discretion, namely the *global variable*.

Global variables are set outside functions, and can be accessed by any function that needs them. In many languages, global variables need to be declared explicitly, but Python simply makes an intelligent guess about what you intend to be global. Basically, if a variable is used within a function, but is neither set in the function nor is in the argument list, Python assumes it to be global. Note that the same variable name can be local in one function (if it is set internally), but global in another. As an example of the use of global variables, consider the function computing the acceleration of gravity towards the center of a planet at the origin:

```
def grav(r):
     return -G*M/(r*r)
```

This has a problem, because the gravitational constant `G` and the mass of the planet `M` have not been defined anywhere. It's bad practice to hard-wire their values in the function, because then it's inconvenient to change them later. If you try evaluating the function after you've defined it, you'll get an error message (try it). However, if you type:

```
G = 6.6742e-11 # In mks units
M = 6.4185e23 #Mass of Mars inkg
grav(1.e12)
```

everything will work fine. You do not need to define the globals until you want to evaluate the function, and you can change their values at any time.

This is convenient, but if you have too many globals being set in too many places, it can be hard to keep track of what is going on. It is also bad practice to use globals for things you will change a lot, which would more appropriately be arguments. The behavior of globals can become even more confusing if you have functions spread across several different files. Nonetheless, you will encounter many cases where using a few globals does the job nicely.

## 1.3 Progressing in Python

Having covered the basics, we now introduce some Python techniques that will be of use in writing programs to do more complex tasks.

### 1.3.1 Writing your own modules and executable scripts

A module should be thought of as a library of useful definitions, including values of constants,lists, definitions of functions, objects,definition of object types, and what have you. A module can also contain executable commands of any type, which are executed by the interpreter when the module is imported. In fact, Python doesn't make any real distinction between a module and any other executable script. Generally, you should think of modules as a place to store things that you will use repeatedly. Although you can execute a module any way that a Python script can usually be executed, a module is intended to be used by importing it into some other script that needs the entities defined by the module.

Generally, when developing code in Python, you always have an interpreter window open and an `idle` editor window. You try things out in the interpreter window, and if things work, you move things into the editor window, which you save for use as an executable script or as a module.

Let's say that you've written a Python script and saved it as `myModule.py`. This script might execute a bunch of calculations, or it might just define functions that you want to use interactively from the Python command line, or it might do both. You can run this Python script from within the Python interpreter by typing `import myModule`. This will run all the executable statements, and also load any functions and so forth that you've defined in your file. This way of running is especially useful if you've defined a lot of nifty functions in your file, and then want to load them in so that you can try them out interactively in the interpreter. Note that all variables and functions imported in this way must be referred to with `myModule.` stuck on the beginning of it's name. Thus, if your file had the statement `radius = 6.0e6` in it, once you imported the file, you would use `myModule.radius` to get at the value of `radius`. Suppose now that you want to change the Python program you've just tried out. For example, you might want to change the definition of the radius to `radius = 6.0e07`. You can edit `myModule.py` in the editor window and save it, but to get the interpreter to recognize the changes, it doesn't work to simply import it again. Instead, you need to type `reload(myModule)` into the interpreter.

You can import many different scripts into your Python interpreter session, and also use the `import` command in scripts you write yourself. Indeed, this is the way that extensions to Python are handled.

If you've written a script that performs some one task (e.g. making a plot of a temperature profile), it is not good practice to execute it by importing it as a module. Rather, you should save the file (say, `myprog.py`, then execute it from the command line by typing `python myprog.py`, or by loading it into an editor window

in `idle` and executing it from there.

### 1.3.2 List comprehension

*List comprehension* refers to the ability to create new lists by processing elements of old lists. It is one of the powerful features of Python that allows one to write compact, readable code, often doing without multiline loops. As a simple example, say we want to generate a list of 10 real numbers equally spaced by increments of 1. Instead of writing a `for` loop and appending the values to a list, one can simply write `[.1*i for i in range(10)]`. Combined with the ability of Python to loop over any list at all, this is very versatile. For example, one can write `[ f(.1) for f in [math.sin,math.cos,math.exp]]`. Suppose we have an averaging function `avg` which returns the average of the elements of a list. Then, if `L` is a list of lists we want to average, we can create a list of average values by writing `[avg(list) for list in L]` The power of list comprehension is further enhanced by Python's ability to do multiple assignments on a single line. For example, suppose we want to open three files, with names `data0`, `data1`, and `data2`. This can be done on a single line using

```
file0,file1,file2 = [open(name) for name in ['data%d'%i for i in range(3)] ]
```

The `open` statement which appears in this example is a built-in Python function that opens a file and returns a file object that can be used to read the contents of the file, or write new contents to the file.

### 1.3.3 Using objects in Python

An object is a collection of data and functions that act on the data. The functions in an object are known as *methods* Almost everything in Python is an object, and you have in fact working with many objects all along. For example, if `mystring = "Use the force!"`, then when you split it using `mystring.split()` you are invoking the `split` method of a string object. The elements of an object are referenced by separating the name of the object from the element by a period, as in the string example. A function is called with parentheses, like any other function, and a value is simply referred to without parentheses. Objects can, if the designer so provides, do many other things. Objects can be called like functions. They can be indexed like lists or arrays (as in `myobject['frodo']`, and they can even be used in arithmetic expressions.

The designer of an object does not actually define the object itself. The designer defines a `class` of objects. When you use an object, you create an *instance*

of the object, just like an individual guinea pig is an instance of the general type of object known as `GuineaPig`. Suppose somebody has defined a class `GuineaPig`, which when instantiated has to be given a name and a weight in kilograms. You create an instance by typing `myPiggy = GuineaPig("Fluffy",1.2)`. Thereafter, you can get the weight by typing `myPiggy.weight` and the name by `myPiggy.name`. The object might also have various methods, such as `myPiggy.squeak("loud")`, or `myPiggy.purr()` or `myPiggy.eat()`. The `eat` method might do something like increment the weight when it is called. You can create as many `GuineaPig` objects as you like, and each will keep track of the data belonging to itself.

In Python, objects can be dynamically modified. That means that new methods can be added to an existing instance of an object at any time. This can be very handy for packaging up functions and parameters for handing off as an argument to another function. The `ClimateUtilities` module written for this Workbook provides a `Dummy` class for this purpose. It creates an object with nothing in it, which you can modify as you like. The following example shows how you can create an object with the gravitational constant, the mass of the Mars, and a function as members.

```
from ClimateUtilities import *
info = Dummy()
info.G = 6.6742e-11
info.M = 6.4185e23
#
def f(x):
   return x/(1.+ x)
#
info.function = f
```

The following shows how you might define a function using this information:

```
def g(r,input):
  return input.G * input.M * input.function(r)/r**2
```

You would call this function with a statement like `g(1.e7,info)`. Try it.

This brief discussion is intended to provide enough background to allow you to work with classes others have defined. The usage of objects will become clearer as you gain more experience working with them. In the Advanced Topics section to follow, we will take up the matter of designing your own objects.

### 1.3.4 The `Numeric` array package

Lists look like arrays, and are versatile objects, but they are not very efficient at fulfilling the functions one expects of the kind of arrays that appear in moderate to large scale scientific computation. It would be possible to write a 2D matrix as a list of lists, and even implement matrix multiplication and vector addition for such an object. However, it would be very inefficient, because lists provide for a very general and mutable data structure. Python does have a native array data type, which is somewhat more efficient, but is still not very good for scientific computing.

This is where the real power of the extensibility feature of Python comes in. When the language is missing some feature some community really needs, the community gets together and writes an extension module which fills the bill. Sometimes this is a cooperative process. Sometimes it is an evolutionary process, with many competing extensions co-existing until one comes to dominate. For scientific arrays, the solution that has come to the fore is the `Numeric` module, which provides highly efficient Matlab-style array objects.

`Numeric` is not written in Python. It is written in very highly optimized `c++`, which is why it is so efficient. This does not mean the developers of `Numeric` had to learn a lot about the internal structure of Python or spend much time at all turning their compiled library of objects into commands that could be accessed within Python. In fact, once such a library is developed, it can be turned into a Python module more or less automatically using a preprocessor known as `swig` (see `swig.org` for more details). Compiled FORTRAN libraries can be similarly spliced into Python using `pyfort` or `f2py`. For this reason, a great variety of numerical analysis libraries are already available as Python modules. Moreover, if you know how to program in `c`, `c++` or FORTRAN, you can very easily learn to build your own customized Python modules. The general strategy is to do as little as possible at the compiled level, building tools there that are very general and of broad applicability. One seeks to isolate the computationally intensive work in a few compiled toolkits, and build more complex models out of these building blocks at the Python level.

`Numeric` provides one of the most fundamental building blocks for scientific programming in Python, and most other Python modules doing numerical analysis or data analysis deal with `Numeric` arrays. The `Numeric` module is imported like any other module, using `import Numeric`. Numeric arrays can have however many dimensions you need.

The first step in using a Numeric array is to create it. `Numeric` provides various ways to do this. To create an array, you need to specify its dimensions and its datatype. Commonly used data types are default float (`Numeric.Float`, usualy double precision), default complex (`Numeric.Complex`), and default integer

(`Numeric.Int`, typically a 32-bit int). `Numeric` does not currently support 64-bit integers unless they are the default integer type on the machine you are using. It also does not support the unlimited precision Python integers, though other modules are available which do. Dimensions are specified as a tuple or a list of integers. For example, the dimensions of a 3 by 5 array are specified as `(3,5)` or `[3,5]`. If the array is one-dimensional, you can just use an integer in place of a list or tuple, if you wish.

One way to create an array in Python is to call a creation routine which makes an array of the desired dimension and fills it in with default data of some particular type. For example, the following lines create a 5 by 10 floating point array of zeroes, a one-dimensional integer array of ones of length 100, and a 10 by 10 complex identity matrix. You can see the values of an array, if it is not too big, by just typing its name.

```
A = Numeric.zeros((5,10),Numeric.Float)
B = Numeric.ones(100,Numeric.Int)
C = Numeric.identity(10, Numeric.Complex)
```

A typical thing to do would be to create an array of zeroes, then fill in the values you want in a loop, as in:

```
A = Numeric.zeros((5,10),Numeric.Float)
for i in range(5):
   for j in range(10):
      A[i,j] = .1*i*i + i*j/10.
```

This example also illustrates the way one refers to elements of an array in Python. Python arrays are zero-based, i.e. `A[0,0]` is the first element in the case above. The designers of `Numeric` provided a very versatile indexing handler, so in fact you could equally well refer to the `i,j` element as `A[i][j]`. In essence, multidimensional `Numeric` arrays act like a list of lists. We'll return to this shortly, in our discussion of array cross section notation. Another important thing to know about `Numeric` array indexing is that it conforms to Python list usage with regard to negative numbers. For example, if `B` is a one-dimensional array, `B[-1]` is the last element of the array, `B[-2]` is the next to last, and so forth. Try this with a 2D array, to make sure you understand the indexing convention.

An array can also be created from a list. You can let `Numeric` infer the data type from the contents of the list, or you can specify it explicitly, in which case a conversion is performed. Try the following statements, and see what kind of array is produced:

```
A = Numeric.array([.1,.2,.3])
B = Numeric.array(range(20),Numeric.Complex)
C = Numeric.array( [ [1.,2.],[-2.,1.] ])
```

Another useful way to create an array is to define it in terms of a function on the indices, so `A[i,j] = f(i,j)`, `f` being some function you have defined. This is probably the most common way of creating the kind of array used in scientific computation without employing a loop. It will generally operate much faster than a Python loop, especially for large arrays. The following illustrates how to create a 10 by 10 array from a function:

```
dx = .1
dy = .2
def f(i,j):
   x = dx*i
   y = dx*j
   return x*x + y*y
A = Numeric.fromfunction(f,(10,10))
```

The two parameters `dx` and `dy` are provided to the function `f` as globals, because that is the only way `Numeric` has provided to pass auxiliary information to the function defining the array. There is an important subtlety in the use of `fromfunction`. The parameters `i` and `j` look like integers, but in fact they are arrays of the same dimension as the array being created. The above example works because the arithmetic being done on the arguments is actually array arithmetic. This allows `fromfunction` to call the function only a single time, rather than in a loop, and results in a much faster computation. If you are using other functions or list indexing inside your creation function, you must take this into account. For example, the following code will work:

```
dx = .1
dy = .2
def f(i,j):
   x = dx*i
   y = dx*j
   return Numeric.sin(x)*Numeric.sin(y)
A = Numeric.fromfunction(f,(10,10))
```

but if we used `math.sin` and `math.cos` instead, it would not work, since these functions do not operate on and return `Numeric` arrays. Similarly, an expression like `myList[i]` will not work inside the function `f` since `i` is not an integer, and so can't

be used as an index. There are workarounds, but generally speaking, `fromfunction` does not work very gracefully with table look-up operations. If you are still confused about just what `i` and `j` are, try putting a print statement in the function to print out the arguments.

There are other ways to create arrays, but the above methods should take care of any cases you are likely to encounter. We have already illustrated how to refer to individual array elements, but it is worth knowing about some of the powerful *array cross section* indexing features allowed by Numeric, which allow you to refer to subarrays easily. Python uses the colon (`:`) as an identifier to build cross sections. The colon by itself stands for the full range of values of the corresponding index. For example, if `A` is a 5 by 5 array, the subarray `A[:,0]` is the one dimensional array with elements `A[0,0],A[1,0],...,A[4,0]`. An index of the form `m:n` would denote the range of values `m,m+1,...,n-1`, so that `A[1:3,0]` would be the array with elements `A[1,0],A[2,0]`. If you leave off one of the endpoints, Python substitutes the first array element for the starting point, or the final array element for the ending point. If `B` were a 100 element array, for example, `B[:25]` would be the same as `B[0:25]` and `B[50:]` would be the same as `B[50:100]`. Finally, you can specify a stride, allowing you to pick off every $k^th$ element. Thus, `0:10:3` represents the set of indices `0,3,6,9`. You can combine subarray indices for the various dimensions of an array in any way you want, as in `A[0:8:2,5:]`.

Now we come to the most powerful aspect of `Numeric` arrays, namely that one can do arithmetic with them just as if they were scalars, avoiding the writing of inefficient and cumbersome loops. The following statements illustrate the kind of arithmentic operations that can be performed with arrays. Note that you will get an error if you try to perform arithmetic on incompatibly sized arrays (e.g. adding a 10 by 10 array to a 5 by 5 array).

```
A = Numeric.ones((10,10),Numeric.Float)
B = Numeric.identity(10,Numeric.Float)
C = (A-B)*10. #You can multiply by a scalar
C1 = (A-B)*B  #You can multiply by an array
D = A/(A+2) #You can divide arrays.
E = C**5.2  #Exponentiation
```

The expressions can be as complicated as you want. Note that `Numeric` array multiplication is element-by-element multiplication, rather than matrix multiplication (and similarly for division). If you want matrix multiplication you use the `Numeric.dot(...)` function, as in `C = Numeric.dot(A,A)`. When the two arrays are 1D, this reduces to the conventional vector dot product. There is also a function `Numeric.outerproduct` which computes the outer product of two arrays. `Numeric`

does not provide a function for matrix inversion; that can be found in various other linear algebra packages made to work on `Numeric` arrays.

`Numeric` also provides versions of the standard math functions, that work on entire arrays. For example `B = Numeric.sin(A)` returns an array of the same dimension as `A`, whose elements are the sin of the corresponding elements of `A`

Array arithmetic can be done between arrays of different types, if the operation makes sense. The result is promoted to the higher of the two operands. For example, adding an integer and a complex results in a complex, or adding a single precision float to a double precision float yields a double precision float. The default float type for `Numeric` arrays is double precision. All floating point scalars in Python are also treated as double precision, so an operation involving a Python float constant and a `Numeric` float array will yield a double precision array.

Cross sections can be used with array arithmetic to compute many useful derived quantities without the need of writing loops. For example, an approximation to the derivative of a function whose values at an array of points `x` are tabulated in the array `F` (i.e. `F[j] = f(x[j])` ) can be computed using:

```
#We assume the function f and the array x have been defined already
F = Numeric.array([f(x1) for x1 in x])
n = len(x)
dx = x[1:n]-x[0:(n-1)]
df = F[1:n]-F[0:(n-1)]
dfdx = df/dx
xmid = (x[1:n]+x[0:(n-1)])/2.
```

The final line defines the array of midpoints, where the derivative has been estimated. Note the use of list comprehension to generate the array `F`. Array cross sections can also be used to do matrix row and column operations efficiently. For example, suppose `A` is an n by n matrix. Then a set of row and column reductions can be done with the following 1D loop:

```
for i in range(n):
  A[i] = A[i] - 2.*A[0]
  A[:,i] = A[:,i] - 3.*A[:,0]
```

You can even do a fairly broad class of conditional operations on arrays, using the `Numeric.where` function. This function takes three arguments. The first is a conditional involving an array. The second is an array expression, to be evaluated and returned if the conditional is satisfied. The third argument is the expression to be returned if the conditional is not satisfied. For example, the following use of

`where` returns the array `A` where `B` is negative, and returns the value 1. where `B` is non-negative:

```
C = Numeric.where(B<0,A,1)
```

Note that the array `A` must have the same dimensions as `B`, since the conditional is evaluated element-by-element. The expressions can be quite complicated. For example:

```
C = Numeric.where(Numeric.sin(B)>0, Numeric.exp(1.+ B*B),Numeric.exp(1.-B*B))
```

The combination of array cross sections, conditionals and array arithmetic is so powerful that one should only rarely need to resort to writing a loop. This is a good thing, since large loops run very slowly in Python, as in other interpreted languages. Try computing the matrix product of two 200 by 200 arrays using an explicit loop, and compare to the time taken to do the same multiplication using `Numeric.dot`. Avoiding loops is also good practice because it makes the meaning of your code more transparent.

`Numeric` offers a rich variety of other useful array operations, such as convolution. For additional information on `Numeric`, just type `help(Numeric)` after you've imported it. Full documentation can be found at `www.pfdubois.com/numpy/`.

### 1.3.5 The Curve object and its uses

The `ClimateUtilities` module written for use with this Workbook provides a `Curve` object, which is intended to simplify the process of reading, and writing plain-text tabular data, and of plotting data which has either been read in from a file or generated by some calculation in your script. In essence, a `Curve` object is a set of data columns, each of which must be the same length, together with optional auxiliary information describing the data. The auxiliary information also allows you to specify certain things about how the data will look when it is plotted.

To use a `Curve` object, you create one, and then "install" data columns using the `addCurve(...)` method. Since `Curve` objects are intended to represent data sets in which any element of a column can be regarded as a function of the corresponding element of another column, all data columns installed must have the same length. You can install any one-dimensional indexable object, including lists and `Numeric` arrays. If you install a list, it will be converted automatically to a `Numeric` array, so you can do arithmetic with it.

The following example creates a curve object containing values of `x`, `sin(x)` and `cos(x)`.

```
import math
from ClimateUtilities import *
x = [(i/10.)*2.*math.pi. for i in range(101)]
y1 = [math.sin(xx) for xx in x]
y2 = [math.cos(xx) for xx in x]
c = Curve()
c.addCurve(x,'x','x axis label')
c.addCurve(y1,'y1','sin(x)')
c.addCurve(y2,'y2','cos(x)')
```

The first argument is mandatory, since it defines the data you want to install in the `Curve`. The second argument of `addCurve` defines a variable name, which you will use to refer to that data column. It is optional. If you omit it, the `Curve` object will create a variable name of the form `v0,v1,v2,...`. If you specify a variable name yourself, you can also optionally specify a "long name," or label, which is used to label the corresponding data when plotting, and also to provide more long-winded information about what the variable represents. The label can be a good place to record the units of a variable.

`Curve` objects are indexed. You refer to a data column by its variable name. In the above example, `c['x']` returns a `Numeric` array of the `x` column, and `c['x'][30]` would return item 30 of that array. You can get a list of variable names in a `Curve` object by using the `listVariables` method, as in `c.listVariables()`

A good general technique for putting data from a calculation into a Curve object for saving, plotting or further analysis is to accumulate the data into a set of lists, and then install the lists in a Curve object. This way, there is no need to know in advance how long the data objects will be. When data from a list is installed in a Curve object it is automatically converted to a `Numeric` array. The following code snippet provides an example:

```
xlist = [ ]
ylist = [ ]
x = 1.
xfact = 1.1
while x < 100.:
     xlist.append(x)
     ylist.append(x*x)
     x = x*1.1 # or, equivalently x *= 1.1
c = Curve()
c.addCurve(xlist,'x')
c.addCurve(ylist,'y')
```

The `readTable(...)` function in `ClimateUtilities` will read columnar tab or space-delimited data from a text file and return a `Curve` object with the data. It is called with the filename as the argument; for example to read the file `"profile.txt"`, you would type `c = readTable("profile.txt")`. The filename can also contain the path to the directory where the data is located, if necessary. Note that you do not have to create the `Curve` object yourself; that is done for you by the `readTable` function. If the data set contains column headers containing variable names (without spaces or tabs in them), `readTable` will recognize that and use them as variable names. `readTable` will also do a pretty smart job of recognizing what is data and what is text description of the data set, such as often is found at the beginning or end of a file. Descriptive text will be separated out from the data, and returned as the `description` element of the `Curve` object.

`ClimateUtilities` also provides an easy-to-use `plot` function which produces line graphs from `Curve` objects. One of the data columns must be designated as the independent, or "x" axis; other data will be plotted as a function of this. By default, the first column installed is designated the "x" axis, but any other curve can be so designated using the `Xid` element (e.g. `c.Xid = 'y1'` in the above example.

If you want to plot the data in a `Curve` object `c`, you can simply type `plot(c)`. The `Curve` object has the following plot options, which can be set to control the appearance of the plot. If `c` is the `Curve` object, then:

- `c.XlogAxis = True` plots the x axis with a log scale, and similarly for `c.YlogAxis`.
- `c.reverseX = True` plots the x axis data with the largest values at the left, and `c.reverseY` plots the y axis data with the largest values at the bottom.
- `c.switchXY = True` switches the x and y axes. For example if you are plotting temperature `T` as a function of pressure `p` (which is designated as the x axis) and you want to make the pressure appear on the vertical axis, you would invoke this option. If you are just plotting one column, the same effect could be achieved by just changing the specification `c.Xid`, but if you are plotting multiple curves on the same graph, you would need to use the `c.switchXY` option.
- If you want any data column to be plotted as a scatter plot, i.e. with symbols at the data points but no line drawn, you can set `c.scatter(varname) = True`, where `varname` is the name of the variable you wish to affect – for example, `'y1'` in the sin and cos curve we defined earlier.

When you call `plot` it puts up the plot in a window which you can move around, but which you won't be able to get rid of until you terminate the Python shell. `plot`

actually returns a plot object which you can use to do further things with the plot. In particular, if you were to write `w = plot(c)`, then you can make use of the plot object `w`. For example `w.delete()` gets rid of the plot window. You can also use the plot object to save the plot as a postscript file, for later printing or incorporating into a lab report. You do this by typing `w.save('myplot')`, which save the plot into the file `myplot.eps`. You can of course replace the filename `myplot` with whatever name you want.

If you don't want to use the python-based graphics, you can always use `c.dump('myfile.txt')` which makes a tab-delimited text file, and then plot it using the program of your choice.

## 1.4 Advanced Python Topics

### 1.4.1 Defining your own objects

You define a new type of object by using the `class` statement. This will define the data and functions (known as "methods") that will be part of the object. The `class` statement should specify a special method called `__init__(...)` which is known as a *constructor*. This method is invoked when a new instance of the object is created, and says what needs to be done to create the object. Sometimes the creation process is very simple, but the creation process can also be very complex. Certain methods, like `__init__` have special meaning. All such methods have names with begin and end with a double underscore.

As a simple example, the following class defines an object which can be used to evaluate the gravitational acceleration as a function of distance to a planet with mass M:

```
class gravity:
   def __init__(self,M,G):
        self.M = M
        self.G = G
   def accel(self,r):
        return -self.G*self.M/(r*r)
```

The argument `self` must be the first argument to every method defined in the object. It provides a way to refer to the members of the particular instance of the object being worked on. When you actually call the methods, or create the object, you leave off the `self` argument and Python puts it in automatically. To make an

object of the type we have just defined, and use it to compute the acceleration at a certain distance, you would use

```
g = gravity(6.4185e23,6.6742e-11) #Invokes the __init__ method to create an
                                  #instance of a gravity object
g.accel(1.e12) #Computes the acceleration
```

You can improve the object by making it callable, so it acts just like a function. You do this by adding a `__call__` method:

```
class gravity:
   def __init__(self,M,G):
        self.M = M
        self.G = G
   def __call__(self,r):
        return self.accel(r)
   def accel(self,r):
        return -self.G*self.M/(r*r)
```

If you create an instance `g` of this object, the acceleration can be computed by just typing `g(r)`. This is a convenient technique for creating functions that remember the parameters needed to compute themselves.

Now, since the gravitational constant is a *universal constant* which should never change from one planet to another, it is rather silly to have to specify it separately for each planet. Python objects allow you to specify data that is common to all objects of a given type. If this class data is changed somewhere, the change applies to all objects. This behavior is useful not just for shared constants, but it also provides a way for one instance of an object to communicate with all other instances of an the same kind of object. Class data is specified within the class definition without any identifier. It is referred to using the name of the *class*, rather than the name of an instance. The following example re-implements the class, making `G` into class data, and also implementing a class-level counter that keeps track of the number of planets created:

```
class gravity:
   G = 6.6742e-11
   nPlanets = 0
   def __init__(self,M):
        self.M = M
        gravity.nPlanets = gravity.nPlanets + 1
   def __call__(self,r):
```

```
        return self.accel(r)
    def accel(self,r):
        return -gravity.G*self.M/(r*r)
```

Now, you can create two planets using `planet1 = gravity(6.4e23)` and `planet2 = gravity12.e23`. The two masses are still `planet1.M` and `planet2.M`, and you compute the gravity for the two planets using `planet1(1.e6)` and `planet2(1.e6)` as before. Now, however, if you need to change the value of the gravitational constant, you can update it for all the planets by writing, for example, `gravity.G = 6.672e-11`. At any time, the number of gravity objects instantiated so far will be `gravity.nPlanets`.

In many applications, an object will store data in the form of arrays. In such cases, the `__init__` method typically will create the arrays and set their initial value. For example, the following class creates and initializes pressure and temperature arrays of a specified length:

```
class profile:
    def __init__(self,n,p0,T0):
        self.p = Numeric.array(range(n),Numeric.Float)*p0/(n-1)
        self.T = Numeric.ones(n)*T0
    def warm(self,dT):
        self.T = self.T + dT
```

The class provides a method which increments the temperature by an amount `dT`. Once you have imported `Numeric`, you can create an instance using `pT1 = profile(100,1000.,300.)`. If you want to find the pressure at index 4, you would type `pT1.p[4]`, and similarly for temperature. You can also change the values of the arrays as you would for any other `Numeric` array, using, e.g. `pT1.T[10] = 301.`.

The special methods `__getitem__` and `__setitem__` allow you to make an object indexable, so that individual elements of an object can be retrieved by specifying an index of some type. Array elements are indexed by specifying sequences of integers, but the object used for doing the indexing can be quite general. The following example shows how to make and use a tridiagonal array object, which can be addressed as if it were a full matrix.

```
class tridiag:
  def __init__(self,n):
      self.A = Numeric.zeros(n,Numeric.Float)
      self.B = Numeric.zeros(n,Numeric.Float)
      self.C = Numeric.zeros(n,Numeric.Float)
```

```
  def __getitem__(self,key):
     # key is a list of arguments passed within square
     # brackets when an indexing operation is performed
     # on an instance of the object. In this case,
     # key is expected to be a two element list.
     if key[0] < key[1] - 1:
         return 0.
     if key[0] > key[1] + 1:
         return 0.
     if key[0] == key[1] -1:
         return self.A[key[1]] #Below the diagonal case
     if key[0] == key[1]:
         return self.B[key[1]] #Diagonal case
     if key[0] == key[1]+1:
         return self.C[key[1]] #Above the diagonal case
  def __setitem__(self,key,value):
     # key is a list of arguments passed within square
     # brackets when an indexing operation is performed
     # on an instance of the object. In this case,
     # key is expected to be a two element list.
     #
     # value is the value to which the element is to be set.
     # It is taken from the right hand side of the equal
     # sign in an expression like M[i,j] = 1., where M
     # is an instance of the object.
     #
     if key[0] < key[1] - 1:
         print "Out of bounds"
     if key[0] > key[1] + 1:
         print "Out of bounds"
     if key[0] == key[1] -1:
         self.A[key[1]] = value #Below the diagonal case
     if key[0] == key[1]:
         self.B[key[1]] = value #Diagonal case
     if key[0] == key[1]+1:
         self.C[key[1]] = value #Above the diagonal case
```

The object could be used as follows:

```
M = tridiag(10)
for i in range(10):
```

```
    M[i-1,i] = 1.
    M[i,i] = -2.
    M[i,i+1] = 1.

print M[9,9],M[2,2],M[2,3]
```

Square brackets are used in indexing operations. If the square brackets contain only a single item, that item is passed to `__getitem__` and `__setitem__` as `key`. If the brackets contain a sequence of elements separated by commas, they are passed as a list, whence `key` has to be treated as a list, as in the example above. This example uses integers for indexing, but the indexing model in Python is completely general. Any Python objects may be used for indexing, including floats, complex numbers, strings, and even objects you have defined yourself.

One of the most powerful features of object-oriented programming is that you can define, or *overload* all the arithmetic operators so that they have the behavior you want when applied to your own objects. This allows you to design customized data types that allow you to condense very complicated operations into compact and simple statements. As a very simple example, let's create a *gaussian integer* data type, which behaves like a complex number, but uses long integers so as to allow unlimited precision. Python does not provide this as a native data type, but the following example allows us to do it ourselves:

```
class gaussInt:
   def __init__(self,real,imag):
       self.real = real
       self.imag = imag
   def __add__(self,other):
       if (type(other) == type(1)) or (type(other) == type(1L)):
            return gaussInt(other+self.real,other+self.imag)
       else:
            return gaussInt(other.real+self.real,other.imag+self.imag)
   def __mul__(self,other):
       if (type(other) == type(1)) or (type(other) == type(1L)):
            return gaussInt(other*self.real,other*self.imag)
       else:
            return gaussInt(other.real*self.real-other.imag*self.imag,
              other.imag*self.real + other.real*self.imag)
   def __repr__(self):
       return "%d + %d i"%(self.real,self.imag)
```

Note that the arithmetic methods create a new `gaussInt` object to return. Note

also the type-checking, which allows arithmetic to be performed between Gaussian integers and regular integers. The class would be used as follows:

```
x = gaussInt(3,5)
y = gaussInt(7,1)
x*3 + y*y
z = 1
for i in range(50):
   z = x*z
print z
```

The `__repr__` method in this example makes the object print out nicely when you type its name or use it in a `print` statement. Try leaving it out, and see what happens when you type the object's name, or perform an arithmetic operation.

We have defined addition and multiplication for our Gaussian integers, but attempts to use subtraction, negation, mod,exponentiation or division will cause an error message, because these operations have not been defined. Each of these operations, and many more, have their own reserved method names that can be defined. There is no need to stick to uses of the operator symbols that resemble their customary use. You're free to redefine addition as multiplication and multiplication as addition, if you want. More usefully, if you need a special symbol to represent matrix multiplication, you can redefine the mod operature `%` to mean matrix multiplication, leaving `*` to mean pointwise multiplication. Or, if you mostly use matrix multiplication, you can define `*` to mean matrix multiplication and `%` to mean pointwise multiplication.

A *commutative operator* (say, "*") is one for which $x * y = y * x$. Many operations, most commonly matrix multiplication, are non-commutative; one needs to keep track of the order of operation. All the binary operators can be made non-commutative in Python. In the Gaussian integer example, we didn't need to think much about which operand in `x*y` was "self" and which one was "other", because the operation being implemented is commutative. To implement a non-commutative operation, you only need to pay attention to which operand is "self" and which is "other". The rule in Python is that in a binary operation such as `x*y`, the first operand (`x`) is "self" and the second (`y`) is "other". As a simple example, let's implement "*" as string concatenation, which is a non-commutative operation. For strings, Python already implements concatenation as the meaning of the operator `+`, so the only point of this example is to fix in the mind the somewhat confusing matter of which operand is "self" and which is "other". We define the class `nonCommutative`:

```
class nonCommutative:
```

```
    def __init__(self,x):    #x should be a string
        self.val = x
    def __mul__(self,other):
        return nonCommutative(self.val + other.val) #Concatenate strings
    def __repr__(self):
        return self.val
```

Now, if we create some instances:

```
x = nonCommutative('a')
y = nonCommutative('b')
```

then `x*y` will evaluate to `ab` and `y*x` will evaluate to `ba`. Thus, to implement a noncommutative operation, you only need to keep in mind that "self" is the operand on the left and "other" is the operand on the right, in binary operations like `__add__` or `__mul__`.

Since matrix multiplication is non-commutative, you would need to pay attention to this if you wanted to overload multiplication of `Numeric` arrays so that `*` meant matrix multiplication instead of point-by-point multiplication. You don't need to pay *much* attention, though, since the basic lesson is that if you keep the operands in the "natural" order, everything will work out fine. The following defines a new class of matrices, for which `*` is matrix multiplication and `%` is pointwise multiplication:

```
class BetterArray:
   def __init__(self,array):
       self.array = array
   def __mul__(self,other):
       return BetterArray(Numeric.matrixmultiply(self.array,other.array))
   def __mod__(self,other):
       return BetterArray(self.array*other.array)
```

The array handed to `__init__` should be a `Numeric` array, though this class doesn't check to make sure that is the case. To try this out, let'x multiply the matrices

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{1.1}$$

To create and multiply these arrays, and print out the result, type the following:

```
import Numeric
```

```
X = BetterArray(Numeric.array([[0,1],[1,0]]))
Y = BetterArray(Numeric.array([[0,1],[-1,0]]))
Z = X*Y
print Z.array
Z1 = Y*X
print Z1.array
```

Multiply out the arrays by hand to make sure our implementation of `BetterArray` handles the non-commutativity properly. Note that to get the results in the above example, we needed to do things like `print Z.array`, because we didn't go through the bother of writing a `__repr__` method for `BetterArray`. Without this, simply typing `X*Y` performs the operation, but tells us nothing useful about the result. `Z.array`is a `Numeric` array, which knows how to print itself. [1]

As an exercise, complete the definition of the `BetterArray` class by defining addition, subtraction and negation, and try out the object. You can further extend the class by adding some type-checking of `other` so that the operations can handle the case of multiplication by a scalar. If you are really ambitious, you could even define matrix division, by incorporating a matrix inversion method for square matrices.

Python also provides methods for "right" versions of the binary operations, e.g. `__rmul__` and `__radd__`. If `y` has an `__rmul__` method and `x` has no `__mul__` method, then `x*y` translates into a call to the `__rmul__` of the object `y` in which `y` is "self" and `x` is "other", i.e. `y.__rmul__(x)`. It works similarly for other binary operations. As we've already seen, you do not need to use the "right" versions to make operations non-commutative. So what are the "right" versions good for? Our `gaussint` class provides an example of a case where you would need `__rmul__` and `__radd__`. If `z` is a `gaussInt` then `z*3` returns the correct answer because `z` has a `__mul__` method and this method checks if `other` is an integer and proceeds accordingly. However, `3*z` raises an error, because the integer has no way of knowing how to multiply itself by a `gaussInt`. The way out of this problem is to give the `gaussInt` class an `__rmul__` method:

```
class gaussInt:
     ... #Same stuff as before here
     def __rmul__(self,other):
        if (type(other) == type(1)) or (type(other) == type(1L)):
            return gaussInt(other*self.real,other*self.imag)
        else:
```

[1] Unfortunately, the `__repr__` method of `Numeric` arrays is buried in the compiled level of the implementation, and can't easily be gotten at.

```
            return gaussInt(other.real*self.real-other.imag*self.imag,
              other.imag*self.real + other.real*self.imag)
```

Now, `3*z` works, because `z` has an `__rmul__` method that Python can use. Note that, because we want `i*z` to give the same result as `z*i`, where i is an integer, we didn't really need to copy the whole definition of `__mul__` int `__rmul__`, as we did above. As a shortcut, we could have written simply

```
class gaussInt:
     ... #Same stuff as before here
     def __rmul__(self,other):
        return self.__mul__(other)
```

Note that we invoke the `__mul__` method as `self.__mul__(other)` *not* as `self.__mul__(self,other)`. That is because the special `self` argument is only used in Python when we are *defining* a method, and never when we are just *invoking* (calling) it. As an exercise, extend the arithmetic of `gaussInt` further by adding an `__radd__` method.

Basically, the "right" versions of the binary operations are never needed if you only want to define operations between objects of the same type. You only need them to deal with operations between objects of differing types. Now here comes the confusing part. Suppose `x` has an `__add__` method and `y` has an `__radd__` method. How is `x+y` interpreted? Is it `x.__add__(y)` or `y.__radd__(x)`? The answer is that Python will implement the operation as `x.__add__(y)`. There is an important exception to this rule. Namely, certain very well-constructed objects will go on to try `y.__radd__(x)` if `x.__add__(y)` creates an error (and similarly for other binary operations). This is how Python was able to make sense of `3*z`, in our `gaussInt` example, even though integers already have a `__mul__` method, which you'd think would raise an error, since they don't know how to multiply an algebraic integer by themselves. This kind of error trapping doesn't happen automatically. It has to be built into the object. Integers, long integers, floats, complex numbers, strings, lists and `Numeric` arrays are all well constructed objects in this sense.

In the case of the class `BetterArray`, it is a considerable nuisance that it is necessary to provide definitions of all the operations you want the object to handle, even if you don't need to change their behavior (as in the case of addition and subtraction). Object oriented languages like Python provide a powerful way to handle this issue, in the form of *inheritance*. [2] Use of inheritance to build daughter

[2]Unfortunately, you can't actually use inheritance with `Numeric` arrays in this way, since the classes used in these arrays are implemented in efficient, compiled code. These object definitions are not accessible to the Python interpreter, and so most their methods cannot be inherited by objects defined at the Python level.

classes from existing ones will not be treated here. Other advanced object concepts we are leaving out include *introspection* and *polymorphism*. The eager student who wishes to dig deeper into object oriented programming can find excellent treatments of these subjects in virtually any complete Python programming handbook.

### 1.4.2 Dictionaries

### 1.4.3 Writing text data to files

The data writing and reading capabilities provided by the `Curve` object will probably take care of all the input-output needed to get through this course. At some point, you may be faced with the need for writing or reading files of a more general form. This section and the next covers the basics of how that is done.

First you need to open the file. To do this, you type

```
myfile = open('test.out','w')
```

where `myfile` is the object by which you will refer to the file in your Python program. You can use any variable name you want for this. Similarly, you can replace `test.out` with whatever you want the file to be called. The second argument of `open` says that we want to create a new file, if necessary, and allow the program to write to it.

You can only write a string to a file, so first you have to convert your data to a string. You have already learned how to do this. The following example serves as a reminder, and also shows how you can skip to a new line at the end of the string.

```
a = 1.
b=2.
outstring = '%f %f\n'%(a,b)
outstring
```

The definition of `outstring` tells Python to make a string converting the elements of the tuple from floats to characters, with a space in between and a carriage return (newline) at the end of the line.

Files are objects in Python, so to write the string to the file, you just do `myfile.write(outstring)`.

Now you have everything you need to write space-delimited data to a file, except that when you are done, you need to close the file by executing `myfile.close()`.

Using what you have learned write a table of the functions $x/(1+x)$ and $x^2/(1+x^2)$ vs. x to a file. You can take a look at the resulting file in any text editor, or load the file into the program of your choice for plotting.

### 1.4.4 Reading text data from a file

To read text data from a file, you need to read in a line of text, and then convert the items into numbers, if that is what they represent. In order to do the conversion, it is necessary to know what kinds of items are in the file, although strings have various methods that can help you identify whether an item is a number or not. Conversion is done by string-to-number routines found in the module `string`. Strings representing integers can be converted to float numbers, but if you try to convert a string representing a float to an integer, you will get an error.

The following example reads a single line from a file, which may consist of integers or floats separated by white-space characters, and converts it into a list of values. To read the rest of the file, you would repeat the procedure until the end of the file is reached.

```
import string
f = open("myfile.txt")
buff = f.readline()
items = buff.split() # Or, do it all at once with items = f.readline().split()
values = [string.atof(item) for item in items]
```