Linux, vi, and Python PharmSci 275

This week covers the computing introduction which will provide the foundation for our assignments. It was originally aimed at a more basic course, so I include it here as background information.

Virtually all scientific computing takes place in a Linux/Unix environment, so we need to cover a few Linux/Unix essentials. Editing of text is essential, so we introduce a common text editor, vi (vim). And we will use the programming language Python to do most of our assignments.

Why do we need all this computer stuff?

In this class, we plan to *do* some calculations. I want you to have some understanding of how the techniques you've learned about actually work, and there is no substitute for actually using them. But, for you to actually do anything with computational techniques, you need a bit more computer background than you pick up doing e-mail and browsing the web.

The first key ingredient we need is some Linux background. Virtually all scientific computing takes place in the Linux environment (not Mac OS X or Windows, though both can be used to interface with it; Mac in particular is built on Linux foundations). This environment provides a lot of power, but it takes a bit of getting used to. You will normally be working with a command-line text prompt that asks you to enter what you want to do, rather than working with windows and drag-and-drop of files. This means you need a bit of training.

Beyond being able to move and manipulate files and folders or "directories" in Linux, you need to be able to edit files, typically text files. A key editor here is the program "vi" (also "vim"), which is a command-line text editor. Again, this won't look like your typical word processor with drop down menus and so on. Rather, it relies on commands you type. Thus, again, there's some background to learn.

Finally, many of the tasks we do are extremely repetitive. A molecular dynamics simulation, for example, consists of calculating the forces between the same atoms over and over again and, once each set of forces are calculated, moving the atoms a tiny amount and then repeating. Computers are great at doing these kinds of repetitive tasks, but we need to write instructions to tell them how to do it, which means a (simple) computer program. Similarly, in the pharmaceutical industry, you might need to look through a database of a million compounds with a computational method to find the ones with a particular set of properties or shape that you are interested in. Who wants to look through a million compounds? A computer is great at this -- but you have to tell it what to look for. So, most of the tasks we want to do throw us back at needing to write instructions to the computer, which means we need a bit of programming. Here, we will use Python -- a programming language about as simple and natural as it can get. It’s relatively easy to learn, but does have lots of real world applications outside this class.

This document gives a crash course into Linux, vi, and Python. The class itself gives a crash course in Python, but if you need to pick up vi and Linux you’ll need to draw on this document and materials online. You’ll get lots more chance to work with these (and some additional training in them) in the context of your assignments.

Set Up Your Computer

Get ready for this introduction with the right software

To get ready for this introduction, make sure you have access to a computer where you can install software. Macs or Linux computers are ideal, but we can get by just fine in this class with Windows as well (though I could use your help updating these documents with information on Windows, as I haven’t used Windows in a number of years).

I’m going to try to have you install all of the software you need for this assignment on your own computer, as it has gotten much more straightforward in recent years.

Installing what you need on your computer

These days, Python installation has been made a lot easier by some large, nearly all-inclusive Python distributions. Install the free scientific Python distribution Anaconda: https://store.continuum.io/cshop/anaconda/ . Please Anaconda3. Also please note that Windows users may need to use the Anaconda2-2.4.x version, rather than the latest, if they encounter problems with f2py subsequently.

Once Anaconda python is installed, make sure you can open Python and type simple commands (i.e. “x = 3”, “print(x)”). In Windows 8 you should be able to use IPython (Py 3.5) or IPython (Py3.5) QTConsole as your Python interface (from the MetroUI start screen once Anaconda is installed). In Mac/Linux, I prefer to do this from the command line (i.e. Terminal application on Mac, found under Utilities) by typing “python”. If this does not work, you will need to update your environment variables to add Anaconda to your environment. On my Mac, I had to use vi (discussed more below) to add /Users/dmobley/anaconda/bin:$PATH to my environment.

Once you have Anaconda Python up and running, visit the course website (or if you’re not in a course, contact Dr. Mobley) for a copy of the oe_license.txt file which contains the OpenEye license file. Then run from the command-line (Windows 8 - the “Anaconda Command Prompt”; Linux or Mac - your normal command-line):

```
pip install -i https://pypi.anaconda.org/openeye/simple openeye-
toolkits
```

This requires you to be online; it should download and install OpenEye Scientific Software's python toolkits.

Once done, you will need to place the oe_license.txt file somewhere safe AND set an environment variable indicating where it is put. To put it somewhere safe, for Mac/Linux, the normal location would be in your anaconda/lib/python3.5/site-packages/openeye directory within your home directory. For Windows this would normally be C:\Users\USERNAME\Anaconda\Lib\site-packages\openeye (where *USERNAME* is replaced by your username). On any platform you can also put it in some safe alternate location of your choosing.

Once you place it somewhere, you need to update your OE_LICENSE environment variable to point to where you have placed this file (see also "Your shell and environment", below). On Mac, assuming you have not changed your default Terminal shell, you need to edit your .bash_profile file (i.e. "vi ~/.bash_profile" in Terminal) and add a line like: 'export OE_LICENSE="/Users/dmobley/anaconda/lib/python3.5/site-packages/openeye/oe_license.txt"' (no outer singlequotes, and modify the path to include your username and where you placed the file, rather than the path to MY file as in the example). In Linux you need to do the same. On Windows you can do this by choosing Open Control Panel > System and Security > System > "Advanced system settings". Hit the "Advanced" tab and then "Environment variables" at the bottom right. Create a new entry with the settings: Variable Name: OE_LICENSE. Variable value: C:\Users\*USERNAME*\Anaconda\Lib\site-packages\openeye\oe_license.txt (replace "*USERNAME*" with your username). Again, you can modify this to use an alternate path if desired.

Once the above are done, you should be able to successfully open Python and type:

```
from openeye.oechem import *

mol = OEMol()
```

without getting any errors.

You will also need to `conda install nb_conda -c conda-forge` to make IPython notebooks be aware of the conda virtual environments.

On Mac OS X, you will also need to install the Xcode Developer Tools available through the Mac App store. Once installed you need to find Xcode in your Applications folder and run it once before the software will be configured. You ALSO then need to go into your command prompt and type 'xcode-select —install' to install the command-line developer tools. FINALLY, you need a fortran compiler, such as the gcc package, or gfortran only. You should be able to `conda install gcc libgfortran`. Alternatively, you can get these from http://hpc.sourceforge.net/, though the conda install route is likely to be easier.

A crash course in Linux and Vi(m)

As noted above, the following instructions are going to assume that either you are already connected to GreenPlanet via ssh, or you are working in a Linux/Unix environment on your own computer (such as on a Linux machine, or in Terminal under Mac OS X).

A brief mention of formatting is warranted. When lengthy commands are given here they will often be included in a gray box like those seen above, including the command prompt and in some cases the output. At other times, commands will be given in line with the rest of the text in a `special font`, usually just when the commands are short. Occasionally, quotes will be used to help items stand out from the text; these quotes are not intended to be typed.

What follows is a *very brief* intro to some absolute essentials on these topics. Plenty more is available via Google.

What is Linux

Linux is just another way to interact with and manipulate files and run programs and commands. It works based on a command-line prompt and text commands that are typed into this prompt. Commands usually follow the format (command name) (things to act on) as will become clear below. Often, options can be specified as well, usually with a dash notation indicating what kind of option is being provided.

Crucial Linux commands

Key operations include listing files and navigating between directories; moving, copying, deleting, and renaming files; making new directories (folders are called directories in Linux), listing what files are present, and copying files between computers, among other things. Editing files is also important, and will be discussed in the context of the vi editor, below.

Please see also the Linux Tutorial linked at the end of this section for another take on an introduction to Linux, in Tutorial format.

One major thing to know in the context of all of the commands below is that items not in the current directory can be accessed using the "/" notation. For example, if I am in my home directory which contains a directory "bananas" containing the file "potatoes.txt", I can refer to "bananas/potatoes.txt".

Another common feature is that many commands allow optional arguments to be specified that modify the behavior of the command. See the "man" command discussed below for more information.

Navigational and informational operations:

- To list the contents of a directory, use the "ls" command, which prints names of things (files, other directories, programs...) found there. For example, in my directory on GreenPlanet:

```
[dmobley@gplogin1 ~]$ ls

all_small_molecules           equil_nvt.0.log      projects
analyze_umbrella_mol.py       gyrase               readxvg.py
asymmetry_erratum             ibuprofen            readxvg.pyc
```

  This also accepts paths, such as "`ls  all_small_molecules`" to list the contents of that directory. A common additional option to ls is "-l" which provides a long list including file sizes and dates modified.

- To navigate between directories, use the "cd" (change dir) command, such as "`cd all_small_molecules`". But what if you want to change to a directory that is NOT within the current directory, such as the directory you were in before you typed "`cd all_small_molecules`"? Linux provides a way to do that -- ".." means the parent directory, so "`cd ..`" tells Linux to take you back to the directory containing the one you are in.

- Sometimes, it is useful to orient yourself and find out where you are. "ls" helps you do this by looking at your surroundings, but even that isn't enough always. "pwd" (print working directory) is another useful command that specifies exactly what set of folders you are in, all the way back to the base of the file system:

```
[dmobley@gplogin1 ~]$ pwd

/home/dmobley
```

  This is saying that I am in the dmobley directory within the home directory. It turns out this is MY "home" directory, the name for a special directory that I automatically start off in every time I log in.

- Sometimes, documentation is essential. "man" is a manual command which accepts the name of another standard Linux command and provides usage information. For example, "`man ls`" will tell you all about how to use ls and what options it takes.

File operations:

- Moving files: Moving files in Linux can really be renaming them, or putting them somewhere else, or both. The relevant command is "mv" and the basic format is "`mv` `file1 directory1/file2`" or variants of this. Here, file1 is moved to directory1 and renamed as file2.

- Copying files: This is the "cp" command in Linux, and the format is just the same as "mv" but the outcome is different -- a copy is made rather than moving the file. For example, "`cp file1 directory1/file2`" copies file1 to file2; note that directory 1 must already exist. Sometimes, one wants to copy a directory; in this case, the copy must be done recursively with the -r option: "`cp -r directory2 directory3`". This will result in a copy of directory2 being placed within directory3.

- Deleting files is done via the "rm" command, i.e. "`rm file1`". Again, directories must be removed recursively, such as "`rm -r directory1`"

- Making new directories is done via "mkdir", such as "`mkdir directory4`" to make a new empty directory by that name.

- Copying files between computers via scp (only if you are NOT using a graphical copying tool such as Cyberduck or WinSCP) is somewhat similar to the copy command, except with a special set of characters to specify the target or source computer. For example, in Terminal on my Mac, I might want to retrieve "test.txt" from my home directory on GreenPlanet:

```
[Pro:~] dmobley% scp dmobley@gplogin1.ps.uci.edu:test.txt test.txt
test.txt                          100%  195     0.2KB/s   00:00
```

  Here the typed command was rather like a copy command (I just typed the line involving scp), but I got some extra output indicating progress of the file transfer (100%) and the file transfer rate and elapsed time. In this case, the format is just like "cp" except that the files can be on other computers, and if they are, I must specify the username and computer name followed by a colon before the file names -- here "dmobley@gplogin1.ps.uci.edu:". In general, this command would be like this: "`scp username@computername:directory1/file1.txt username2@computername2:directory3/file2.txt`"

Useful shortcuts

Linux has loads of useful shortcuts. Learning as many of these will mostly be up to you, but here are a couple of my favorites:

- Use the tab key to autocomplete words. For example, if I am typing the command "`mv plants.txt new_plants.txt`" in a directory containing two text files, "`plants.txt`" and "`pianos.txt`", I can type "`mv  pl`" and then hit tab to get "`mv plants.txt`", then type the rest of the command. This is especially useful for long file names involving directory names (paths).

- Use the up and down arrows to get back previous commands you typed -- for example, if you mistyped a command and need to change something and do it again, use the up arrow to get it back and then edit it
- When editing a line of text, use ctrl-e to go to the end of the line and ctrl-a to go to the beginning
- Many directories have shortcuts, such as "~" for your home directory, "." for the current directory, ".." for the directory containing your current directory, "../.." for the directory containing THAT directory, and so on.
- Wildcards: Often, one might want to do something only to certain files. For example, I might have hundreds of files in a directory and want to see only files that end with the characters .txt. Wildcard characters provide an easy way to do this. The "*" character means "match any character, so "`ls   *.txt`" will list all files beginning with any characters and ending with .txt in the present directory. Multiple wildcard characters are acceptable, so "`ls   directory*/b*.txt`" will look in all directories with names beginning "directory" and list all text files beginning with the character b and ending with the letters ".txt". Some other commands aside from `ls` accept wildcard characters as well.

Other essentials

- Absolute versus relative paths: When a location of a file or directory is specified, if the location does *not* begin with a "/", this means it is specified relative to your current directory (as printed by "pwd"). This is called a relative path, and they are tricky -- to use one, you have to know where you are. An absolute path is one that begins with a "/" -- it works regardless of where you are (where you've "cd"'d to) so in that sense they are preferable, though they do get much longer and more cumbersome and there are some special cases where relative paths are preferable. An example will help -- let's say you tell a classmate to look in "bio/calculations", a relative path. This assumes that they are going to be looking within the same directory you're referring to -- a directory containing the "bio" subdirectory. That's fine, but if you don't tell them you're assuming that, they may get back to you and ask, "There's no 'bio' directory!". So, it may be better to tell them, "Look in /home/dmobley/bio/calculations". This is an absolute path (begins with '/') and will work without you having to explain where the bio directory is (since the path itself tells where it is). It is, however, more cumbersome, so you if you wanted to refer to other subdirectories of 'bio', you might switch back to relative paths: "Within that same 'bio' subdirectory, you can find X in the 'analysis' subdirectory."

Your shell and environment

Linux's command prompt environment is called a shell. It is actually possible to use different shells. While all of the basic commands are the same, some commands are different, and if you wanted to write a script to perform a complicated, repetitive series of commands, the way this would be written depends on the shell.

In this class, GreenPlanet should already be configured so that you use the shell "bash", which is one of the more common shells ("tcsh" is another common alternative). The main thing to know for the purposes of this class is that shells exist -- but if you are trying to do things on a different computer system and they just aren't acting quite like they do here, check what shell you're using.

Another major factor in using Linux is your "environment" and your "environment variables". Think of this as an analogy to an office environment -- your environment is the stuff that's around you that you can easily use. It's the same thing here. The environment specifies what programs and software you can use, and where they are found. This is usually specified in a particular file, in terms of a set of places Linux will look to find programs and commands. For tcsh this is in your home directory (~) as a hidden file, ".bashrc". (Any filename beginning with a period is by default hidden.) You shouldn't need to do anything with it in this course, but know that it exists, and again, if you find yourself using a different Linux computer system and installing new software on it, you may need to think about editing your environment variables (the things in this file that actually specify where software is installed) to make them point to the software you need.

Linux tutorial

Tons of Linux tutorials are available and probably come and go faster than I can maintain the links here, so I refer you to Google. However, this tutorial *currently* seems reasonable: http://ryanstutorials.net/linuxtutorial/

It covers a lot of the same material discussed here. Also, this quick reference guide provides a list of a lot of common (and not so common) commands with very brief descriptions and usage examples : http://www.pixelbeat.org/cmdline.html.

Using Vi to edit text -- the basics

Vi is an incredibly powerful command-line text editor. However, since it works from the command-line, it isn't menu driven like most Word processing or text editor programs you are used to. This, combined with its power, can make it a little bit tricky to get started on.

Here, I provide some of the absolute essentials, but I also recommend going through a tutorial. Lots more info is available via Google. And, know that vi (or its slightly more advanced colleague "vim", which usually is what you end up using when you run the vi command), is probably one of the most advanced text editors in the world (as addressed by this "Why Use Vi" column: http://www.viemu.com/a-why-vi-vim.html), so if you're after a powerful text editor, this is a good one to learn.

So, what are the essentials? First, how to open vi -- simply type "vi" from the Linux command prompt, or, to edit a specific file with it, "`vi  myfile.txt`", for example. If the myfile.txt file exists, it will be opened in vi; if it does not, it will be created.

Once in vi, you will typically see a terminal screen with the contents (or partial contents) of your file in it, and a single line at the bottom listing the name of your file. You can navigate this file using the arrow keys or a variety of different keyboard commands. If the file is empty or mostly empty, you may also see a lot of "~" characters denoting empty lines.

It's important to know that vi has two primary working modes, "command mode" and an "edit mode" which actually comes in several different flavors, the most common of which is "insert mode", for inserting new text or deleting old text. Depending on which mode you are in, what you type will get different behaviors. In command mode, for example, characters you type are understood to be commands (things like "copy this line" and "paste what I copied here" or "delete that line" and so on). In insert mode, most of what you type is understood as characters to insert or delete -- that is, what you normally expect when you type into a text file.

One of the most important things to know, then, is how to switch between these modes. When you open a file, you begin in command mode, so if you want to start typing into the file, switch to insert mode using the command "i". You should notice that now the text at the bottom of your window should say "---INSERT---" indicating that you are in the insert mode. Type whatever you want, using the delete keys as normal to delete things you don't want, etc. Then, when you are done editing, use the escape key to go back to command mode.

That's fine, but what if you want to save your work? Or exit? No menus are apparent, but it turns out saving is done with a command from command mode. Specifically, in command mode (which you enter by typing escape), type "`:w`" -- yes, a colon followed by a w. Then hit enter, and your work gets saved. You should see a message to this effect at the bottom. What about quitting? "`:q`". However, vi won't let you quit without saving unless you tell it you really mean it, so to quit with unsaved work, "`:q!`". Or, if you want to save and quit at the same time, "`:wq`".

Those are the absolute basics of vi, but it is an incredibly advanced text editor. For example, it can easily indent blocks of computer code, perform powerful search and replace operations, repeat previous commands to prevent repetitive typing, and all kinds of other things. If you find yourself doing some repetitive editing task, odds are vi can help you do it much faster. Tons of help is available online with Google.

One other feature that is incredibly useful is that vi understands programming languages such as Python, which we'll be working on in this class, and it is able to actually highlight the syntax, helping to make your code more legible, showing you where you have ended parentheses, and so on. You DO have to be using an ssh or Terminal program which supports color for this to work, but most of the recommended options above should meet these criteria. (it also has to be turned on).

Setting some recommended options for vi

Go ahead and edit a sample text file with vi, something like "vi test.txt" from the command prompt (If you're going to be using GreenPlanet, you should do this there). Type some text in using insert mode and save it (escape, then ":wq").

Next, I want you to edit some of vi's default settings to turn on syntax highlighting as just discussed so you can take advantage of it, and also turn on a couple other useful features. Type "vi ~/.vimrc" (which edits a hidden file vi uses to specify its settings) and add the following to the file in insert mode:

```
:set term=ansi
:syntax on

:set tabstop=4
:set shiftwidth=4
:set expandtab
```

Save and quit. (If you don't want to know what this does, skip to the next section).

What this does is sets the terminal to one vi recognizes for displaying colors and turns on syntax highlighting. It also sets vi to autoindent computer code using 4 spaces before each indented block of code, and sets it so tabs are automatically expanded out in code to four characters. These are important when writing Python code (which relies heavily on indentation) if you ever use the tab key, as otherwise you will end up indenting some lines with spaces and other lines with tabs, which can cause Python to have problems in some circumstances (and also cause problems for anyone else reading your files who has vi set to display tabs differently than you). Short version: Turn this on, and you can use tab OR space to indent your code, and vi and Python will both properly understand your indentation. This will make more sense after you learn some Python.

A First Crash Course in Python

Here, I again aim to provide some absolute essentials on Python. You will, however, need to do additional reading, either now or as you work on your assignments, so I will provide a couple of additional references. You will likely want to refer to the second source below as you write assignments.

Additional resources

One useful reference is the "Non-Programmer's Introduction to Python", which some of my students have found useful. It is written geared towards someone who has done no programming before, and does not provide a complete overview of the language.

Another useful reference is this Python Tutorial or Introduction which provides lots of examples and gives a much more complete introduction to the language, but is more geared towards people who have some idea how programming works.

One word to the wise: Both of these tutorials are oriented towards Python 2.7. One significant syntax difference you will have to pay attention to is that Python 3.x uses "print(x)" rather than "print x" to print things.

Much more on Python can be found online, including at the official Python website, Python.org.

Introduction to programming and Python

Programs are a sequence of instructions to the computer (really, a "compiler" or "interpreter") that run from top to bottom.

All programming languages, including Python, have certain basic features. These include: decisions or loops, which have the ability apply conditions (if x, do y) or loop over a sequence while a condition is met (while x is true, do z; for every member of q, do w). Data structures store information -- for example, numerical information (integers or floats (decimal numbers)), text (also known as strings), or (at least in Python) generic information stored in a sequence (in lists) or by keys (in dictionaries). Variables are names used to access a particular instance of a data structure. For example, x=32 stores the value 32 in the variable x. Programs also have commands -- things that do stuff, such as "x+y" or "print(x)" and so on.

In Python, punctuation and formatting can be crucial to meaning. Your set of instructions simply won't work if it's missing a close parenthesis after an open parenthesis, or missing a colon where there needs to be one, and so on. Also, spacing can be key -- in Python, it is used to indicate pieces of the program which are subordinate to or controlled by other pieces, as we will see.

There are two main ways to interact with Python, which itself is a program you can access from the command-line. The first is to actually type the word “python” and access the interactive “interpreter” and type Python commands in to it. This is the best for learning Python, but the downside is that you aren’t saving what you’re doing so you can reuse it later. The second main way to interact with Python is to type things you *would* have typed into the interpreter into a separate text file with a filename ending with “.py” and store them. Then, this file itself can be run through python from the command line. This provides a powerful way to store commands to reuse or edit, and to build up complicated programs you couldn’t easily type in all at once or which you might want to modify later.

Here, the best way to get started with Python is to type “python” at the command prompt and open the interpreter, then get started typing commands. But in this class, assignments will be built on the second way of working with Python -- saving Python commands to a text file (such as “assignment1.py”) and running this from the command-line using “python assignment1.py”. Generally, as you’re working on an assignment, you probably want to try commands using the interpreter directly, then save them into a text file (ending with “.py”) to store them for later reuse. As noted in the vi section, vi can help make these files more readable by colorizing your code.

Getting going in Python and data structures

Start your Python experience by typing “python” on the command line:

```
>>>
```

This is the python interpeter waiting for you to type something. Let’s do a very simple program:

```
>>> x = ‘Hello World’
>>> print(x)
Hello World
```

Here, anything following “>>>” is something YOU type, and anything not after those characters is something Python prints out. Here, ‘Hello World’ is a Python *string*, a data structure that can store a sequence of any kind of characters (including special characters, such as tabs (‘\t’) and the newline or carriage return character (‘\n’)). Here, we create a new variable or object x by assigning it to store the string we defined. Then, we use the print command to print the contents of x.

As noted, there are many different data structures or types available in Python. We just saw strings. Integers are a common one used for storing whole numbers:

However, we also often need numbers which have decimal places. In Python these are called floating point numbers or “floats”. One major difference is precision -- an integer, 12, is exactly

```
>>> y=12
>>> y/4
3
>>> y/5
2
```

12, while a floating point number, “12.0”, is approximately 12 (to the precision of the computer).

```
>>> z = 12.
>>> z/5
2.3999999999999999
>>> print(z/5)
2.4
```

Notice that conversion between types is done using the name of the target type. It is not needed very often, but occasionally -- most often when converting between things that are significantly different in type, such a a string and an integer:

```
>> z
12.0
>>> int(z)
12
>>> y = “2”
>>> int(y)
2
```

Python also handles special data types called “Boolean” data, which have two values, either True or False. (These are not in quotes as they are not strings -- Python recognizes these as special keywords).

Lists are generic Python data structures that can contain pretty much anything. They are defined using square brackets, and items can be added to lists by appending them:

The above code creates an empty list named a, and adds the items 1, ‘test’, and 1.43 to the list by appending them (adding them at the end), then prints it. It then creates a second list with some strings in it, prints it, and creates a third list out of a combination of a & b.

It’s worth noting at this point that common operations, such as +, have different meanings in Python depending on what type of data they are operating on. 1+2 has the common mathematical meaning, but adding two strings together means something different, and adding two lists together means something else entirely. Adding a string and a number together will result in an error, as Python doesn’t know what you intend in this case.

```
>>> a=[]
>>> a.append(1)
>>> a.append('test')
>>> a.append(1.43)
>>> print(a)
[1, 'test', 1.4299999999999999]
>>> b = [0,2,'another list', 'final entry']
>>> print(b)
[0, 2, 'another list', 'final entry']
>>> c=a+b
>>> c
[1, 'test', 1.4299999999999999, 0, 2, 'another list', 'final
entry']
```

Objects and decisions in Python

Everything in Python is an object. An object is something with properties and functions (actions which can be performed) that can be accessed via dot notation. For example, we already saw the append function, which is a function attached to a list object which allows adding to the list. But lists have several other useful functions:

```
>>> c
[1, 'test', 1.4299999999999999, 0, 2, 'another list', 'final
entry']
>>> c.index('test')
1
>>> c.remove('another list')
>>> c.reverse()
>>> c
['final entry', 2, 0, 1.4299999999999999, 'test', 1]
```

Python provides information on these modules and properties via the “dir” command, which accepts a Python object and gives you back a list of things that can be applied to the object.

```
>>> dir(c)
['__add__', '__class__', ..., 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> help(c.count)
count(...)
    L.count(value) -> integer -- return number of occurrences
of value
```

Ignore those entries beginning with '__', and focus on append, count, extend, index, insert, and so on. These are all functions which act on a list. You can find out more about any of them using the `help` command, i.e. '`help(c.insert)`' for help on what the insert function does and what it needs in order to work.

Decisions are another very important part of any programming language. Often we want to do something only if certain conditions are met. An 'if' statement provides a means of doing this, running statements within a following indented block of text (program) only if the condition is met. It can be supplemented by elif and else statements -- elif (standing for "else if") providing a means to do something else only if an additional condition is met, and else provides a fallback -- if no conditions are met, what's inside 'else' is done. For example:

```
>>> a=1
>>> if a==4:
...     print a
... elif a==2:
...     print a*a
... else:
...     print a*a+1
...
2
```

Here, indentation (extra spaces) is used to designate "blocks" of code -- that is, sections of code which are inside the if statement and only run if the condition is met. When the indentation ends, the blocks end. (In interactive mode, as here, there is an extra carriage return to end the block as well).

Decisions are based on comparisons, and a variety of different tools for making comparisons are available. In Python, the "=" sign is used to set one thing equal to another, and a double equals sign is used to compare two things for equality -- hence "`if  a==4`" in the example above. Other common comparisons include "<" for less than, ">" for greater than, "!=" or "<>" for not equal, "<=" for less than or equal to, and so on. Logical and mathematical operations are also permitted -- for example, "`if (a==b and c!=2)`" or "`if not (a or b):`" or "`if (a-b < 2):`"

A 'while' statement is another common kind of decision (and the simplest loop, or repeated task), running an indented block of code as long as a certain condition is true. Here, you can see that 'ct' is initially set to 5. The while loop runs as long as it is larger than 1, so we then enter the while loop. 'ct' is printed, and then we compute "ct - 1", which is 5-1 or 4. This is then stored back to 'ct'. Now, ct = 4, and we go back to the top of the while loop to see if the condition (ct > 1) is still met. It is, so we print ct again, subtract one again, and so on. The countdown continues as long as ct is greater than 1.

```
>>> ct = 5
>>> while (ct > 1):
...     print(ct)
...     ct = ct - 1
...
5
4
3
2
```

This example uses another common feature in many programming languages -- a seemingly impossible equation such as "`ct = ct -1`". These kinds of statements are always executed beginning with the right, so, as noted, we first evaluate the right hand side to get a value, and then store this value back in ct. So, despite its algebraic impossibility, it makes perfect sense in a computer program.

'While' loops are the simplest loops, but 'for' loops are the most common. 'For' loops take a sequence of entries (such as a list of numbers or values) and run a series of steps for each entry in that sequence. For example:

```
>>> elements = ['hello','world', ', ', 'again']
>>> for elem in elements:
...     print elem
...
hello
world
,
again
```

Here, `elements` is a list of entries. The 'for' loop looks at each entry in turn and prints it. Since the print statement switches to a new line every time it prints, our entries get spread across lines.

It's worth noting that the basic syntax of a for loop is '`for (your choice) in (sequence name)`' where (sequence name) is the name of the list or sequence of items to work on, and (your choice) is a name you specify. It's usually a good idea to make this descriptive, but you can really make it anything you want -- so in our example above, we could have said, '`for banana in elements:`', and then asked it to '`print(banana)`'.

Common mathematical operations are all available in Python as expected -- plus, minus, and divide (/) all have their usual functions (though it's worth noting that 1./3. is floating point division, while 1/3 is integer division, and they yield different results because 1/3 gets rounded to the nearest integer). Exponentiation is done by `**`: "`a=x**2`" takes the contents of x and squares it. It's also possible to take remainders: `z = 13%4` takes the remainder of 13/4, which is 1. Operations of the sort x = x+1 or y = y-1 are so common that there is even a shorthand for these expressions: x+=1 or y-=1 means the same thing. This also works for division: z/=3, for example. Other operations, such as sqrt for square root, log for natural logarithms, and log10 for logarithms in base 10, as well as common trig functions, are available in standard numerical libraries such as numpy or math. (Libraries are extra toolsets you can 'import' into Python to get additional functionality).

Slicing for lists and strings

Often we will want to work with just part of a list or a string, perhaps a group of elements. For example, we might want just specific characters from a particular line. (Some homework tracks will see this problem in an assignment). This can be done with a technique called slicing:

```
>>> line = "This is the line we're working on right now."
>>> line[16:20]
" we'"
>>> line[1]
'h'
```

Slicing takes just certain characters from a list or string (here, a string). This particular case first requests characters 16 through (but not including) character number 20. Slicing always works this way -- the first entry number is included, but the endpoint is not included. It's also worth noting that line[1] points to character 'h', which is the second character on the line. This is because for historical reasons Python starts counting with character 0, so line[0] would point to 'T'.

Slicing can get a lot more complicated (and powerful) as highlighted in these examples, which you're encouraged to try out for yourself:

```
>>> line[:20]
"This is the line we'"
>>> line[20:]
're working on right now.'
>>> line[:-5]
"This is the line we're working on right"
>>> line = line[7:20]
>>> line
" the line we'"
```

In the first two cases, when no character number is specified before or after the colon, it is assumed that you mean either from the start of the list or string (in the case that the left entry is missing) or to the end of the list or string (in the case the right entry is missing). Negative numbers can also be provided, and these count back from the end. Lastly, we reset 'line' to be just a subset of the original string.

While all of these examples are for a string, they work in essentially the same way on lists.

Using Python noninteractively

As noted above, using Python noninteractively (from the command line) is more common for big tasks. This allows us to avoid retyping things when we need them again, allows us to reuse our code, and allows for easy troubleshooting. Using it noninteractively simply amounts to typing Python commands into a text file (with name ending .py) and running it from the command prompt:

```
x = [0,1,2,3,4,5]
y=[]
for i in x:

    y.append( i*i-1 )

z = x+y
print(z)
```

Running from the command prompt:

```
[dmobley-Pro:~] dmobley% python test.py
[0, 1, 2, 3, 4, 5, -1, 0, 3, 8, 15, 24]
```

Functions in Python are reusable tools

Python functions are tools you can think of as a black box. They take something in, do something with it, and give something back, and you shouldn't have to know what goes on inside the box in order to be able to put things into it and get them out. Python has many built in functions, but it's also possible to write your own, and we will see quite a bit of this. Think of it as making your own reusable, general purpose tools.

It's also worth noting that functions can take in any number of items, and give back any number of items, including none.

Here are a couple (somewhat silly) functions to illustrate by way of example:

```
>>> def multiply(a,b):
...     c = a*b
...     return c
...
>>> multiply(77,43)
3311

>>> a = [1, 2, 3, 4, 5, 6]
>>> def modifylist(list):
...     list.append(33)
...
>>> modifylist(a)
>>> a
[1, 2, 3, 4, 5, 6, 33]

>>> def printstuff():
...     print("WARNING: Executed the printstuff function.")
...
>>> printstuff()
WARNING: Executed the printstuff function.
```

Here we have three examples -- multiply, which takes two things in and gives back one; modifylist, which takes one thing in and gives back none (simply modifying the thing put into it) and printstuff(), which takes no things in and gives back nothing (simply printing a message).

A function always starts with a 'def' statement, which defines it. It also takes zero or more 'arguments', given within the parentheses after its name. When it is first defined with a 'def' statement, these arguments are just names -- for example, multiply is said to take two arguments, which for the purposes of the definition are called 'a' and 'b'. What the function does is defined in terms of these names (multiply a and b together), but the names are not used outside the function definition. Think of the names as describing things that go on inside the black box -- if you're going to use the black box, you don't need to know the names, you just need to know what goes in and what comes out. That's why when multiply is actually used, it's '`multiply(77,43)`'. If we can see inside the box, we can see that 77 is going to go into the name 'a' and 43 is going to go into the name 'b' and they'll get multiplied together. But we don't have to know that to use the box. We just need to know we put two values into it and it gives back the product.

If a function is to give anything back, it does so by the `return` statement. Here, we see it returning one value, but '`return`' can also give back multiple values, such as "`return a, b, c, d`" for example, which would give back all four items.

If you're going to use a black box, you don't necessarily need to know what goes on inside it, but you do need to know what it's supposed to do and what it takes in and gives back. Python provides a powerful way of doing this called 'doc strings'. These make it possible for Python's built in help to work for functions you have written as well. For example:

```
>>> def a(x, y):
...    """Adds two variables x and y, of any type.  Returns
single value."""
...    return x + y
... <hit return>
>>> help(a)
Help on function a in module __main__:

a(x, y)
    Adds two variables x and y, of any type.  Returns single
value.
```

This doesn't appear critical when it comes to a simple function like 'add', which there's really no point in using and which is trivial. But it becomes crucial for real functions in order for them to be useful to people other than their author, or over the long term.

Python code reuse and readability is also improved by commenting. Python allows lines marked with a "#" symbol at their beginning to be ignored. These are called comments, and should be used to explain key features of what you are doing in any Python-based assignments (and any programs you write in any other context!). For example, you might put in your program:

```
#Compose a new list consisting of the squares of elements in
the original list
b = [ elem*elem for elem in a]
```

The line beginning with "#" is informative but will be ignored by Python.

Python modules are packages of code that can be used from other programs

Python modules are extensions of a sort, providing additional functionality that you or others may need from your Python programs. Many are already provided with Python, but others are available for a wide variety of common tasks. You may even write your own. Any Python file containing Python code you have written can also be used as a module. We already mentioned several examples of modules, such as 'math' and 'numpy', which we will see again for numerical operations. But you might also have written your own module for working with prime numbers. To use this within Python, you might do this, if you already have a file "primes.py" containing some tools you want to use (such as the "nextprime" function).

```
>>> import primes
>>> primes.nextprime()
```

Existing modules already have a lot of useful functionality. Suppose we want, in Python, to get a list of what is available in the directory where we're working (like we might get from the Linux 'ls' command). We can do this:

```
>>> import os
>>> os.listdir('.')
[..., 'Applications', 'bin', 'calendar', 'Desktop',
'Documents', 'Downloads', 'Illustrations', 'Library', 'local',
'mbox', ... ]
```

This provides a list of what is available in the present directory. There are also (in the 'os' module and elsewhere) tools for doing most of the common Linux file operations, as well as running tasks directly at the command line.

File operations in Python are a strong point for Python

File input and output in Python is rather easy. To open a file, we do this, and help is available:

```
>>> file = open('README', 'r')
>>> help(file)
```

Opening a file takes options such as 'r' to open in read mode, 'w' to open for writing, and 'a' to open in append mode (where items can be added to the end of the file).

Once a file is open, various options are available, such as:

- file.readlines(), file.writelines(name): Read or write lines as a list of strings
- file.readline(), file.writeline(name): Read or write a single line
- file.close(): Close the file when done reading or writing (and save).

Programming requires some planning

Before you sit down and start trying to write a program, first make sure you understand your task. Then, break your task into small steps and describe them in English (if you are a beginner, I suggest actually doing this in outline format on a piece of paper, though more advanced students may do this in their heads). Write down a description of what each major step is, and then outline in detail each minor step in that process, in English. If a step is too hard, it means you don't understand it well enough or haven't broken it down enough, so break it down further.

Once you have it outlined in detail, think about how you translate each English statement into Python, and begin working on your program. Do keep in mind, as you do so, that any repeated steps or steps you may use in another program should probably be written as functions.

One goal of good programming is to make your code reusable -- both by yourself later (5 years from now when you've forgotten!), or by someone else who doesn't have you around to explain it to them. This means that good code should be commented, explaining your goals, major steps, and then a variety of intermediate steps along the way. Use descriptive variable names: 'filename' makes a better variable name for the name of a file than 'i', and so on. And, generally, write reusable functions for repetitive or general tasks -- a tool you use here could be used again in the next problem if you make it reusable.

Pointers for troubleshooting

Inevitably, programs don't work as you would like. My first suggestion is to try things in the Python interpreter (within Python itself) before trying them from the command line when possible, as this can provide useful information. Also, use print statements -- printing contents of variables throughout the code can help you see what's going on. Step through the code if necessary -- there are ways to do this using something called a debugger, or just use "raw_input()" statements at key places in your code. When a program hits one of these, it will pause until you hit enter, providing a way to pause programs.

Also, test components of your program separately. If you use a bunch of functions together, make sure each one works separately. Each step or component ought to work individually before you try and make them work together.

If your problem is too confusing, simplify by breaking it into smaller steps.

Illustrative examples

Sum of the squares of integers:

Let's imagine we want to write a really simple function to compute the sum of the squares of the first 20 integers. First, two examples of what won't work:

```
#Not this
for i in range(1,21):
   sum = i**2


#Not this either
for i in range(1,21):
   sum = sum+i**2
```

Here, “range” is a builtin Python function that gives back a list beginning at the first number and ending before the second, so we’re asking to loop over the list [1, 2, 3, ..., 20]. That part is OK. But the first example goes wrong because it simply makes ‘sum’ BE the square of i. The loop is right, and i is right, but the final value in sum will be 400, which is not what we want.

The second example won’t work at all. It’s closer, but ‘sum’ isn’t defined except inside the ‘for’ loop. The first time we get there, what will ‘sum’ be when we evaluate the right hand side of the equation? It won’t be anything, and so Python won’t understand it. Rather, we need to do this:

```
#Compute the sum of the squares of the first 20 integers
sum = 0
for i in range(1,21):
   sum = sum+i**2
```

We have to set sum=0 *outside* the ‘for’ loop so it’s already defined when we need to use it inside the loop.

Printing only lines with a certain word from a file:

Next, consider reading a text file and printing only lines containing a certain word:

```
file = open('dummytext.txt', 'r')
text = file.readlines()
file.close()

for line in text:
    if ‘comfort’ in line:
        print(line)
```

This uses the readlines function attached to the file object to read in all the lines from the file into a list we call ‘text’. It then looks at each line of text individually (naming them ‘line’) and prints the line out if the word ‘comfort’ occurs on the line.

Multiply elements of two lists together, but skip certain results:

Now, let’s take two lists and multiply their elements together, composing a new list. But let’s also stipulate that we don’t want any elements larger than 150 in the new list.

```
list1 = range(0,15)
list2 = range(15,30)

newlist = []
for (idx, elem) in enumerate(list1):
    prod = elem*list2[idx]
    if prod<150:
        newlist.append(prod)
print(list1, list2, newlist)
```

Here, we define two lists using the range function as discussed above -- list1 running from 0 to 14 and list2 running from 15 through 29. Note that these have the same number of elements. We want to multiply the first element of the first list by the first element of the second, the 2nd of the 1st with the 2nd of the 2nd and so on. One way to do this is with a ‘for’ loop. But we want to store the results into a new list, so, like with our sum a couple examples above, we set up a new list (‘newlist’) outside the for loop so it already exists when we need it.

Then, the ‘for’ loop looks a little unconventional. The thing is, now we want to track not just what value we are working on, but what element number we’re working on. That is, we want to know we’re on the first element of list 1, so we can look up the first element of list 2. So, we use another feature of a ‘for’ loop, which allows us to have two things change at the same time. ‘enumerate’ is a Python built in function that takes a list (or certain other sequential data types) and gives back the numbers of the items and their values. So, enumerate(list1) will give us back the pairs 0,0; 1,1; 2,2; and so on (kind of boring in this case since they are the same, but for list2 it would be more interesting -- enumerate(list2) would be 0,15; 1,16;, and so on). Now, inside our ‘for’ loop, we take ‘elem’, which is the element of the first list we’re looking at, and multiply it by the corresponding element of the second list (‘list2[idx]’, the idx-th element in list2). This gets stored to ‘prod’. We then check if ‘prod’ is less than 150, and if it is, we store it to our new list. And that’s the end!

On to something mildly useful -- let’s find the first N prime numbers!

Next, let’s find the first N prime numbers. Remember, a prime is a number divisible only by itself and 1.

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

#N is limit for how many primes we want to find
N = 50

#Start with 2 as the first prime; ignore 1 since every number is
#divisible by 1
primesfound = [2]
#Track which number to look at next
ct = 3

#Loop until we find N primes
while len(primesfound) < N:
    #Track whether we have found the number we are looking at NOT to
    #be a prime
    notprime = False
    #Loop over every element of our existing primes to see whether
    #our new number is divisible by one of these
    for elem in primesfound:
        #Check to see if it is divisible; if so, break out of the
        #loop and go on to next number
        if ct%elem == 0:
            notprime = True
            break
    #If our number is not divisible (that is if it is prime) track it
    if not notprime:
        primesfound.append(ct)
    #Go on to next number
    ct += 1

print(primesfound)
```

Here, we start off by specifying how many prime numbers we want -- in this case, 50. And we jump start the process by listing prime numbers we already know, beginning with just the number 2. We start a list to track every prime we've already found, and a counter 'ct' to track which number to look at next.

We'll look for prime numbers in the direct way here -- start at a low number (3) and work our way up. Every time we get to a number, we'll check whether it's divisible by any of the prime numbers we already know, and if it is, it's not a prime. If it's not divisible by one, it's a prime. That's what the while loop is about -- it is going to keep going until we have found N prime numbers. It starts by assuming a number is prime, and starts a tracking variable 'notprime' to track whether we've found it's NOT prime. Then, it looks through all the primes we already know about using a '`for`' loop, and if our current number is found to be divisible by any known prime (remainder 0) it sets 'notprime = True' and runs the 'break' command. 'break' is a command that exits out of the last loop, so in this case that means leaving the 'for' loop. Then, we come to the last 'if' statement. If we haven't found the number is not a prime by finding something it's divisible by (that is, if it IS a prime) we add it to our list. Either way, when we get to the end of the while loop, we go on to the next number.

Note that there are probably much more efficient ways to find prime numbers -- this is just the most direct way.

A simple function example -- factorials

Let's try and calculate a factorial, and let's make a reusable function to do it. We want our function to be a black box. It will take a (positive) integer, and return the number's factorial (the product of the number itself with all smaller integers, i.e. 3! (3 factorial) is 3*2*1 = 6). First, let's do this the simple (but longer) way:

```
def factorial( num ):
     """Take a positive integer and return its factorial."""
     total = num
     next = total - 1
     while next > 1:
          total = total*next
          next = next - 1
     return total
```

What happens here? Let's suppose we put in the number 4, and think it through. Total is set to 4, and we set next to 3, and then we hit the 'while' loop. 'next' is greater than 1, so we enter the while loop. 'total' now gets set to 'total' (4) * 'next' (3) which is 12. Then we subtract 1 from 'next', so we'll look at 2 next. We then check to see if 2 is greater than 1 and it is, so we go back to the top of the while loop. Now 'total' gets set to 'total' (12) * 'next' (2) which is 24, and 'next' gets set to 1. Now, we check to see if we take another trip through the 'while', but 'next' is now 1, so we are done. So our black box spits back out the result currently in 'total' -- 24.

If you don't follow that 100% and you're planning on either of the Python-based homework tracks, go back and work through it one line at a time. Start at the top filling in what each variable is until you hit the end of the while loop. Then go back to the top of the while loop and do it again, switching to a different color pen or a different spot on your paper. Keep going until you get to the end.

So, that’s not bad -- a factorial function that works, consisting of 6 lines of code. That seems pretty good, but in fact there’s an even shorter function that will do the same thing:

```
def factorial( num ):
      """Take a positive integer and return its factorial."""
      total = num
      for value in range(1,num):
            total = total*value
      return total
```

This one is a bit harder to understand (since it does the multiplication in reverse order), but the result is the same. In fact, to the user of these functions, it wouldn’t matter which one were used, since it’s only what’s *inside* the box which is different -- input and output are the same.

Common mistakes in Python

Here are some common mistakes. Python is good about telling you what mistake you've made, but at first it is hard to understand what it's saying. Try some of these out to see what error messages you get and start getting a feel for how it tells you what the problem is.

```
#Error 1
for i in range(5)
   print(i)

#Error 2
b = 2
a = sqrt(float(b+2.)


#"scope" errors -- this works but maybe not how you would think
for i in range(20):
   a = []
   a.append(i)
print a

#type errors
a = 'test string'
b = 2
c = a+b
```

A small word of caution about lists and assignment

Python often handles assignment by pointing. For example, setting "a=1" creates the number 1 in the computer's memory, and points the variable a to it. This means that copying doesn't necessarily work as one might expect, especially for lists or dictionaries. Watch out for setting '`a = [1, 2, 3]`' and 'b=a'. After subsequently setting '`b[1] = 0`', '`print(a)`' gives [1,0,3]. Somehow changing b changed a! That's because b was pointing TO a, so changing an entry of b changes what b is pointing to (a). This behavior can be avoided by making sure that b is a copy of a when that is intended ('`import copy`', then '`b = copy.copy(a)`').

This is not true for some types of data -- numbers are 'immutable' so, for technical reasons, setting a=b and then changing b simply changes what b points to.

This can be confusing. The point is, just be careful not to assume that 'b=a' makes b a COPY of 'a', though it does for numbers.

Numerical Operations in Python

Why NumPy and SciPy

Python is not a 'compiled' language, where code is 'compiled' for the computer to run in advance, a step that takes some time. This is both good and bad, and the bad is that it tends to be slower than many other languages, at least for doing large numerical calculations. One way around this is to use Python 'modules' that are built in other languages that DO work well for numerics. We will use several of these here. NumPy (Numeric Python) provides basic routines for manipulating arrays or matrices and other numeric data. SciPy (Scientific Python) extends NumPy by providing additional routines such as minimization, Fourier transformations, statistical packages and a large variety of other tools. Because these are precompiled and written in other languages, they are faster (though the user doesn't need to know about thsi part to use them).

Reminder: Import modules to access their functionality

NumPy and SciPy have to be imported into Python before you can use them:

```
>>> import numpy
>>> #Access using numpy.X
>>> #OR:
>>> import numpy as np
>>> #Access using np.X; this will be assumed in what follows
```

The code which follows will assume you have used '`import numpy as np`', so that numpy tools are accessed via 'np.X'

Working with data in NumPy

While NumPy has a lot in it, the key data structure we will see the most here is an NumPy array. It is like a Python list, but it contains numerical data, which must all have the same type (either a float or an integer (int)). However, it is much more efficient for numerical operations than a list, and many common tasks can be done very quickly on these arrays. One major difference from lists is that arrays are given a size in advance, while lists can be expanded using append.

It is easy to create NumPy arrays, and there are several ways to do it:

Here you can see example array creations by either converting a list to a (1 dimensional) array, or creating a new array from scratch out of zeros. In the latter case, the (10) specifies the size of the array.

```
>>> a = [ 1, 4, 5, 8]
>>> b = np.array( a, float )
>>> c = np.zeros( (10), int )
>>> b
array([ 1.,  4.,  5.,  8.])
>>> c
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> type(c)
<type 'numpy.ndarray'>
```

Multidimensional arrays are also possible (of arbitrary dimensionality), and different ‘axes’ or directions in the arrays are accessed using commas within the brackets for the array, as illustrated here:

```
>>> a = np.array( [ [1,2,3], [4, 5, 6] ], float )
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

Let’s suppose we want to read a bunch of numbers from a file, with one stored on each line of a text file, and put it into a NumPy array. How do we do this when we can’t append to an array, and have to set its size initially? This illustrates one approach:

```
>>> #Open a file and read lines
>>> file = open(‘tmp.txt’, ‘r’)
>>> text = file.readlines()
>>> file.close()
>>> #Allocate storage for data from file
>>> data = np.zeros( (len(text)), float )
>>> #Loop over lines in text, manipulating and storing to data array
>>> for (linenum, line) in enumerate(text):
...     #(code to parse line would go here)
...     data[linenum] = ...
```

The example is missing the code to actually get the number from each line (which would depend on the file format) but should be enough to illustrate the key idea -- first determine how many entries will need to be stored (such as by looking at the length of the file), then read the data and store it.

Arrays can also be sliced in a matter much the same as slicing of lists:

```
>>> a = np.array( [ [1,2,3], [4, 5, 6] ], float )
>>> a[0,:]
array([ 1.,  2.,  3.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:, -2:]
array([[ 5.,  6.]])
```

Note, as in lists, the use of ':'. When ':' alone is specified in a particular direction, it means all elements in that direction -- for example, 'a[:,2]' means all rows and the #2 (third) column in array 'a'. 2D arrays can be thought of as having rows and columns, so the first index is the row number and the second is the column number.

Often, we need to determine certain things about arrays, such as their shape ('a.shape'),

```
>>> a = np.array( [ [1,2,3], [4, 5, 6] ], float )
>>> 2 in a
True
>>> 0 in a
False
```

their length ('len(a)' gives the length along the first axis), or whether certain values are present:

Arrays can also be converted (back) to lists: 'a.tolist()'. Combining arrays also is different from combining lists -- combining arrays is done by concatenation: '`c = np.concatenate(a,b)`', for example.

Working with data in NumPy

For array math, the arrays should be the same size (typically). '+', '-', '*', '%', and '**' all do operations on the arrays in an element-by-element way as you might expect. It's worth noting that for multidimensional arrays, multiplication (*) performs element-by-element multiplication. (If you've had linear algebra or worked with matrices, this might seem strange and you might expect matrix multiplication, but there are functions that will do that if that's what you want).

Arrays, unlike lists, *can* handle simple additions or subtractions (or even multiplications) of non-arrays. For example:

NumPy provides many mathematical functions and constants, including functions such as `abs` (absolute value), `sign` (the sign of a number), `sqrt` (square root), `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`,

```
>>> a = np.array([1,2,3], float)
>>> a+=2
>>> a
array([ 3.,  4.,  5.])
```

`arcsinh`... Constants include pi and e. There are also many functions for whole array operations, such as `a.sum()` to sum elements in the array. Other functions are applied in the same way and include `prod` (product of elements), `mean`, `var` (variance), `std` (standard deviation), `min`, and `max`. `argmin` and `argmax` give indices of minimum and maximum values. Application of these can also be limited to a particular axis (direction) using the optional axis argument:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
```

Standard comparisons can be performed on arrays; these return arrays of Boolean (True/False) values. For example:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
```

NumPy also provides an extremely powerful function named 'where' which can be used to easily select only certain elements of an array:

```
>>> coordinates = np.array( [0.0, 0.45, 0.75, 0.13, 0.89, 1.47,
0.14], float )
>>> interesting_residues = [ True, False, False, True, True, False,
True ]
>>> indices = np.where( interesting_residues )
>>> interesting_coordinates = coordinates[indices]  #See p. 15, NumPy
writeup
>>> interesting_coordinates
array([ 0.  ,  0.13,  0.89,  0.14])
>>> np.where( coordinates < 0.3 )
(array([0, 3, 6]),)
```

Logical operations can also be used in constructing conditions for selecting elements of an array. NumPy provides '`logical_and`' and '`logical_or`', so, for example, one can ask for ‘`where( logical_and(a > 0, a<3))`’ to obtain indices where elements of a are greater than zero but less than 3.

Additional references on NumPy

For additional information on NumPy, check out the documentation on the NumPy site (http://numpy.scipy.org/) as well as this more complete writeup from a colleague (http://dl.dropbox.com/u/3409095/Research%20Group/Python%20Intro/numpy_intro.pdf), which also includes some information on installing it if you would like to install it on your own computer.

Making plots and graphics within Python

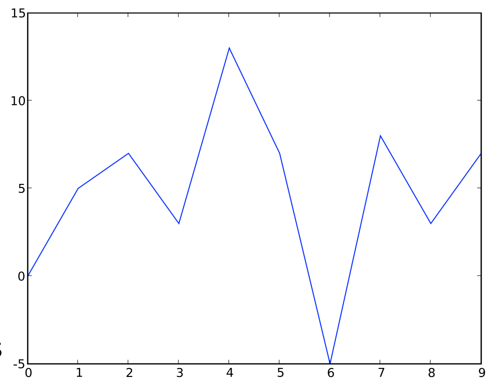
Matplotlib/Pylab plotting tips and tricks

Matplotlib and the connected library Pylab provide a lot of useful tools for making plots and graphics directly within Python and either displaying them to your screen or saving them to a graphics file in one of a variety of formats.

Making really basic plots is simple, and is highly recommended anytime you need to visualize data you already have within Python -- it will take you more time to get the data *out* of Python into some other format than it will to plot within Python, and Python can make publication quality plots using these libraries.

Here's a really basic example:

```
>>> import pylab as pl
>>> import numpy as np
>>> xvals = np.arange(10)
>>> yvals = [0,5,7,3,13,7,-5,8,3,7]
>>> pl.plot(xvals, yvals)
>>> pl.savefig('firstplot.pdf')
```



At the most basic level, Pylab plots a set of x values (in a list or array) versus a set of y values. Axis labels, legends, and all of the other typical plot options are available. Graphics can be output to various formats, including pdf (here), jpg, png, svg and so on. Axis labels can be added using `xlabel`, `ylabel`, and so on; limits can be adjusted using `xlim`, `ylim`, and similar. Symbols and line styles can be specified with an optional argument to the plot command, such as '`plot(x, y, 'mo')`' to plot magenta circles (for formatting info, see help(plot)).

By default, issuing a second plot command will simply ADD to your existing plot, but you can create a new plot using figure().

Lots of documentation and examples are available online and you can find them easily using Google. See also the help command.

You will likely end up making some plots within Python, so you may need to refer back to this section.

Random number generation

Computational science often requires 'random' numbers

Often, we want to do something with a certain probability. For example, we may have two choices for what to do next, and decide to pick one with a 50% probability. That means that we essentially have to have the computer "throw a coin" -- basically, it needs to pick a random number to determine whether or not this event happens. There are many applications of this; one example is in assigning initial velocities to objects in a molecular simulation. We know what the average velocity ought to be given a target temperature, but we are not given any information on velocities of particular particles. So, one way to deal with this is to give particles random initial velocities selected in such a way that they have the correct average velocity.

So, we need to be able to pick "randomly", but computers can't actually generate random numbers (they can't throw coins) -- if you program them to do something, they do it, which is a deterministic rather than a random process. So, what we actually do in computational science is generate "pseudorandom numbers" -- numbers that are generated deterministically from an initially starting point, but distributed in a way that is very similar to random. These can be used to make *apparently* random choices -- and many are good enough that for practical purposes the numbers generated do behave like random numbers (though there are also bad random number generators!). NumPy has a random number generator available in its Random module:

```
>>> np.random.seed(293423)
>>> np.random.rand(5)
array([ 0.40783762,  0.7550402 ,  0.00919317,  0.01713451,
0.95299583])
>>> np.random.random()
0.70110427435769551
>>> np.random.randint(5, 10)
9
```

This initializes the random number generator by providing a 'seed' number, then generates 5 random numbers stored in an array. It also generates one additional number (a float) after that, and then a random integer between 5 and 10.

Interfacing Python with Other Languages

Optional: You can write your own fast external libraries

As noted above, Python is not particularly fast at high performance numerics, so libraries like NumPy and SciPy provide routines written in other languages (Fortran or C++) which can be used within Python to provide good performance. It is actually possible to write these sorts of modules yourself without much effort, if you know some Fortran or C++.

Here, several of the homework tracks will use Fortran modules I have built, and for those homework assignments, instructions will be provided on how to use those within Python. However, if you would *like* to know more about how this works, you may want to consult this guide, which was prepared for another class but covers this topic: http://dl.dropbox.com/u/3409095/Research%20Group/Python%20Intro/f2py_fullpacket.pdf