Python Decal: Python and Computing for Astronomers

Pauline Arriaga, Baylee Bordwell

January 12, 2016
### Contents

1 UNIX  
1.1 What is UNIX? ................................. 1  
1.2 SSH-ing and SCP-ing .......................... 2  
  1.2.1 For Linux and Mac Machines ............... 3  
  1.2.2 For Windows Machines .................... 3  
1.3 Changing Your Password ..................... 3  
1.4 Basic Navigation ............................ 4  
1.5 Summary .................................... 8  
  1.5.1 Important Commands ...................... 8  
1.6 Tutorial .................................... 8  
1.7 Problems ................................... 9

2 The Basics of Python  
2.1 Why Program? ............................... 11  
2.2 Python as a Calculator ...................... 12  
2.3 Data Types .................................. 16  
2.4 Strings ..................................... 17  
2.5 Lists ....................................... 18  
2.6 Packages and NumPy ......................... 18  
2.7 Summary .................................... 19  
  2.7.1 Definitions .............................. 19  
  2.7.2 Important Functions and Procedures .... 19  
2.8 Tutorial .................................... 20

3 Writing Simple Programs  
3.1 Basic Functions ............................ 24  
3.2 More Advanced Functions ................. 25  
3.3 Writing Programs ............................ 27
## CONTENTS

### 4 Data Structures
- 4.1 Array Operations .......................................................... 32
- 4.2 Creating Arrays .............................................................. 32
- 4.3 Tutorial ...................................................................... 33

### 5 Control Statements
- 5.1 For Loops ................................................................. 36
- 5.2 If Statements .............................................................. 36
  - 5.2.1 Conditionals .......................................................... 36
  - 5.2.2 Using Conditionals .................................................. 37
  - 5.2.3 Elif and Else Statements ......................................... 38
- 5.3 While Loops ................................................................. 39
- 5.4 Returning to normal flow ............................................... 40
- 5.5 Tutorial ................................................................. 40
- 5.6 Problems ................................................................. 44

### 6 Advanced Arrays
- 6.1 Multidimensional Arrays ............................................... 45
- 6.2 Array Sub-scripting ...................................................... 46
- 6.3 Array Concatenation ..................................................... 46
- 6.4 Other NumPy Array Functions ....................................... 47
- 6.5 Tutorial ................................................................. 48
- 6.6 Problems ................................................................. 48

### 7 Good Programming Practice
- 7.1 Formatting ................................................................. 51
- 7.2 Comments ................................................................. 52
- 7.3 Debugging Tips and Tricks ........................................... 54

### 8 Reading and Writing Data
- 8.1 Reading Data ............................................................. 57
  - 8.1.1 Reading Data with numpy.loadtxt .............................. 57
  - 8.1.2 Reading .npz files ................................................ 58
- 8.2 Writing Data ............................................................. 58
  - 8.2.1 Writing to a .txt (or text of any kind) file ................ 58
  - 8.2.2 Writing to a .npz file ............................................. 59
- 8.3 Tutorial ................................................................. 59
- 8.4 Problems ................................................................. 60
## CONTENTS

9 **Plotting** 61
   9.1 Basic Plots .............................................. 61
   9.2 Advanced Plotting ..................................... 68
   9.3 Important Functions/Syntax ............................ 68
   9.4 Tutorial .................................................. 68
   9.5 Problems ................................................ 68

10 **Images** 69
   10.1 FITS files: Headers and Images ....................... 69
   10.2 Important Functions/Syntax ........................... 70
   10.3 Tutorial ................................................ 70
   10.4 Problems ............................................... 70

11 **\LaTeX** 71
   11.1 Why \LaTeX? ............................................. 71
   11.2 \LaTeX Headers and Packages ......................... 71
   11.3 Basic Math and Formatting ............................ 73
   11.4 Good Writing Practice ................................ 73
   11.5 Tables ............................................... 73
   11.6 Figures ............................................... 73
   11.7 Some Useful Templates ............................... 73
   .1 Using Windows to SSH and SCP ....................... 75
Chapter 1

UNIX

1.1 What is UNIX?

If you have never used a UNIX command line before, it may seem rather strange and arcane, like something out of a hacker movie or a science fiction scene. However, UNIX is simply an operating system, like Windows, Mac OSX or Ubuntu. Whereas in other operating systems, you interact with files and folders by dragging, dropping, clicking and double clicking icons on your screen, in UNIX you interact your files through a command line. Instead of moving a file from Documents to Music by dragging its icon from the Documents folder to the Music icon, in UNIX you type a command saying, in essence, “move file to Music”. However, like every programming language, you have to use very specific syntax so that the interpreter of UNIX understands what you’re trying to say. You can imagine that if the language did not have this specificity in its syntax that you may have ambiguities in the language. For example, you would have to specify in your syntax exactly which file you are trying to move. Additionally, you can have different folders named Music, only in different places, so you will have to completely specify which Music folder you are referring to.

Therefore for every operation (moving a file from here to there, copying a file from there to here, deleting a file), there is a very limited number of ways to give the command to the computer. From the UNIX terminal, we enter in our commands, and the UNIX system will interpret them and carry them out. In addition to commands to the UNIX file system, we can also open and control installed programs, much like we would from the Applications
folder or the Start menu. Python and \LaTeX{} are two such programs that we will be interacting with through the command line.

We will be accessing and storing files and programs on the UGAstro server. A server is basically like a system of computers all hooked together so that we can share and store these files. Most servers, including the UGAstro servers, utilize UNIX systems and so in order to access and manipulate files on these servers, we will have to use the language of UNIX. Though there is a sharper learning curve to a UNIX system, we will have the extra perk that since we are directly giving commands to the computer, we have a wider range of abilities than we had with our drag-and-drop or Guided User Interface (GUI) operating systems.

1.2 SSH-ing and SCP-ing

In order to access the UNIX operating system that has our distribution of Python, we will need to access a remote server, or computer hosted by Berkeley. From there we will be able to access a UNIX command line and will be able to access our remote file systems. In order to do so you will need to be on a decent internet connection. If you are on a Linux or Mac machine, you will have your own local UNIX command line, and can directly connect with the UNIX command line of the server. However, if you are on a Windows machine you will need to some programs to connect to the server. What we are doing when we connect is a process called SSH (Secured Shell Host). When we give commands to the UNIX command line we are giving commands directly to the remote computer. Sometimes we will want to open up programs and interact with them through the remote computer. In order to do this we will need our computer to do X11 forwarding, which allows the remote computer to launch programs to our computer. If you are SSHing in a Linux or Mac you will see this through the “-X” option, whereas if you are SSHing on a Windows computer, you will need a separate program to do this.

\footnote{Unless of course you have your own distribution of Python, but storing and manipulating data on the UGAstro server may be much easier on UGAstro}
1.2.1 For Linux and Mac Machines

In order to access your own UNIX command line, you will need to find a program called Terminal. On Mac machines you will find this in your Applications folder, and on Ubuntu you can access one by pressing Ctrl+Alt+T. From this command line you can enter in our first UNIX command: ssh, shown below. In the following example where “username” is your assigned username (on the UGAstro server this will usually be your first initial and last name), and “hostname” is the host of your server (on ugastro, it is @ugastro.berkeley.edu)

```
ssh -X username@hostname
```

So for example, mine would be

```
ssh -X parriaga@ugastro.berkeley.edu
```

1.2.2 For Windows Machines

SSHing and doing most programming is not recommended on Windows machines and if you do have one, I highly recommend installing Ubuntu on dual boot. This will allow you to use both Ubuntu and Windows on the same computer. However, if you do need to use a Windows system, you will need two programs, PuTTY and some form of an X11 forwarder. PuTTY will allow you to access the UNIX terminal. See Appendix A for information on how to configure this.

1.3 Changing Your Password

On the UGAstro system, you will need to change your password from the default (you should get your initial password from your instructor). To do this you will need to ssh into the Vega server. By default when you SSH to UGAstro you are on the Aquarius server. You change servers by on UGAstro by typing:

```
ssh -X username@computer.ugastro.berkeley.edu
```

So in this case

```
ssh -X username@vega.ugastro.berkeley.edu
```

Before we actually change your password, let’s look a little more in depth as to what we’re doing when we type all of that in. SSH is a UNIX command. A command, as the name suggests, is an instruction that we give to UNIX, and they’re carried out by typing their name. The “-X” is an option that
we give to SSH. There are other options we can give as well which we can find in the documentation (we’ll discuss this later). These affect how the SSH command runs, but are not necessary to run (thus option). If we do not provide the “-X” option then the command will work differently. In this case, omitting the “-X” will not allow windows to pop up. Finally we have the “username@etc.”. This part of the command is called the argument. Arguments is the part of the command where you give the specifics of what you want it to do. Whenever you SSH, you will type “ssh” every time. But when you want to SSH to a different computer or a different server, you will change the argument.

So now back to changing your password. To do this we will run the passwd command:

```
passwd
```

and follow the instructions. After that you can log out of vega with

```
logout
```

We will not need to log into Vega for anything else in this course.

### 1.4 Basic Navigation

Now that we’re in a UNIX terminal, let’s see what’s in our directories. For this we use the `ls` command

```
ls
```

If we’re on the ugastro server, we’ll see a few directories (folders) that are set up for us by default. Otherwise, nothing might come up. Either way, let’s make a new directory using the `mkdir` command:

```
mkdir pydecal
```

Then, if we run the `ls` command again, we should see a new addition to the contents of our directory, which looks like:

```
pydecal/
```

We can tell that this is a directory because it ends in a `. So now this directory is all ready for us to put files and other directories into. The `mkdir` command, like the `ssh` command takes one argument. In the case of the `ssh` command, it took the name of the server we wanted to `ssh` into. In this case, our argument is the name of the directory that we want to create. The `ls` command does not have any arguments. If we try to call `mkdir` without any arguments as we do the `ls` command, our UNIX terminal will give us an error

```
mkdir: missing operand
```
1.4. BASIC NAVIGATION

This means that we do not have the correct number of arguments. An error is what occurs if we try and give the terminal a command that it does not understand or cannot process. This often happens if we make typos in our commands:

> mdkif pydecal
mdkif: command not found

or if we provide too many or too few arguments for any given command. These are the most common error messages that we will see for UNIX commands. In general, if we find an error message that we have not seen before and do not understand, we can, in general, do a quick Google search of what the particular error message means.

Now let’s look around in this directory. First, we’ll want to go into that directory using the cd (short for change directory) command:

> cd pydecal

If we look at the contents of this directory with:

> ls

We’ll see there’s nothing in it. So let’s copy a file over to your directory by the following into your command line:

> cp /home/parriaga/ex.txt /home/yourname/pydecal

Now if we use the ls command in our home directory, we see that the file ex.txt is now there. This is a command that uses two arguments. The first is the file that we want to copy and the second is where we want to copy it to. Our arguments are always separated by spaces. Let’s practice moving this file around by moving out of our decal folder. We can do this in one of two ways. First we can cd by specifying the whole path:

> cd /home/yourname/

Or we can use the special path specification of “..”:

> cd ..

Which always specifies the directory in which the current directory is contained. In this case, the current directory was

/home/yourname/pydecal/

while “..” is equivalent to

/home/yourname/

In other words, our command would have the same effect as if we had typed

cd /home/yourname/

---

**Problem 1.0** What would happen if we input the following commands? What directory do we end up in?
> cd directory1
> cd directory2
> cd directory3
> cd ..

Now we’re in our home directory, so let’s try to move the file ex.txt using the mv command. Like the cp command, the first argument of the mv command is the file we want to move, unlike when we were copying the file before, we don’t have to specify the full path, we just specify how to get to the file from the home directory.

**pydecal/ex.txt**

One important distinction to note is that when we specify the whole path from home, we always begin with a slash, whereas when we’re specifying the path from where we are, we do not have this slash.

The second argument represents where we want to move it to, which is the current directory we’re in. We can specify that we want the current directory with a single period:

```
mv pydecal/ex.txt .
```

and if we use the ls command we’ll see that the file ex.txt is now there.

Now our task to prepare for the tutorial will be to move all of the files from `/home/parriaga/public_html/python/week1/`

However, you might despair at the task since in the folder we have given you there are many files and it would be time-consuming to transport all of them. For this task we use a **wildcard**. A wildcard works the same way as it does in many card games: a wildcard can stand for any suit or any card value. In our case, a wildcard can stand for any string of characters.

For example, in the directory which holds the example files, the command:

```
> ls *e
```

yields

```
Blue Circle Joe Lace Nose PaulinePickleShoe Solid State Square White
```

In this case, the wildcard is standing for “Blu” and “Circl” and “Jo” and “Lac”. In plain English, this command is really saying “list all of the contents of this directory ending in e”. Note that our command specifies anything that ends in e, and not merely anything that has an e in it. This is because our star is before the e and so our wildcard can only stand for the contents before
the e and after it. **Problem 1.0** Run the following ls commands. What do they output?

```
> ls *te
> ls P*
> ls *
```

Note that this wildcard does not care how many characters it stands for. There is a different wildcard which is a “?” that will only stand for one character. For example,

```
> ls file?
yields
file1  file4  file5
```

whereas

```
> ls file*
gives us
file1 file11 file14 file299 file4 file5
```

**Problem 1.0** How would you list all of the files with 6 characters in them?

So now back to our original problem, copying all of the files:

```
cp /home/parriaga/public_html/python/week1/* .
```

This puts them all in the pydecal directory.

The last command we’ll cover is the **remove or rm** command. Be careful with this command as removing a file using rm means it’s gone forever. There is no recovery file created or recycle bin. It takes one argument which is whatever.

There is much more to learn about UNIX and its options, but we have shown you enough so that you can Google any new commands or options you’ll need and understand its documentation. If you ever want to look up the options for a command, you need only look at the manual page for it by running:

```
man commandname
```

which will show you the options and arguments for any command as well as some helpful examples. Here are some commands that you should look up the usage on your own and figure out: grep, find, ps and kill.

Let’s give an example on how to read these man pages with the ever useful command grep. Typing man grep gives us a document that tells us a lot of information. Looking under synopsis we find:

```
grep [OPTIONS] PATTERN [FILE...]
```

The options in square brackets
are things that are optional, while the things that are not in brackets are necessary. The ellipses “...” indicate that you can include more than 1 file. A short read through the synopsis shows that grep is a command that will search through a file for the “pattern” or just a word/phrase. Usually a short google search will give us examples of how to use it, but sometimes man pages also give examples. Let’s

Under the options, the listings will give two ways to pass the option, one which has one a single dash and a letter such as “-V” and one which is longer with two dashes such as “--version”. Either will work; use whichever you remember better.

1.5 Summary

1.5.1 Important Commands

- **mkdir [arg1]** - Creates a new directory with the name arg1.
- **cd [arg1]** - Changes into the directory arg1. Like all arguments that are directories, it may be specified with respect to the current directory or from /home/username/etc...
- **ls** - Lists all of the contents of a directory
- **scp [arg1] [arg2]** - Shifts documents from arg1 to arg2 in which either arg1 or arg2 are on a remote server
- **mv [arg1] [arg2]** - Moves the file arg1 to location arg2
- **cp [arg1] [arg2]** - Copies file arg1 to location arg2
- **rm [arg1]** - Permanently deletes arg1
- **man [arg1]** - Gives the documentation for arg1

1.6 Tutorial

1. Make a directory in your home directory called "pydecal"

2. Make a directory in /pydecal called /homework
3. Copy every file from:

/home/parriaga/ay98/unix_homework/

into a directory called "week1" in your ay98homework directory (you will have to make the directory "week1" first\(^2\))

4. Navigate to your week1 homework directory

5. Create a "tarball" called "e_files.tgz" that contains all the files that contain a lower case "e" in their titles.

6. Of the remaining files, find out which one contains the string "You found me!" and on which line in the file the string is located. \(^3\) \(^4\)

7. Rename the file found in number (6) "line_?", but replace the question mark with the line number "You found me!" was on.

8. Create a directory in your week1 directory called e_files.

9. Extract all of the files in "e_files.tgz" into the directory "e_files"

10. Now, there is a text file entitled "Emacs" which is a tutorial for using the text editor in the following directory:

/home/parriaga/Decal/

Copy it to your week1 directory, open it, and follow its instructions.

### 1.7 Problems

\(^2\)Hint: Use a wildcard
\(^3\)Remember how to type spaces on the command line.
\(^4\)Hint: use grep, and use an option that outputs the line number. Find the option using the "man" command)
Chapter 2

The Basics of Python

2.1 Why Program?

A lot of what we’ll use Python for fall under the category of using Python as a glorified calculator. By glorified calculator, I mean that we’ll be using it to do arithmetic and statistics. However, as we’ll see, Python is a lot smarter than our average calculator. Think about your basic calculator, the kind you get from the dollar store with about twelve buttons on it. In order for this calculator to be particularly useful in any context, the user has to know exactly what they want to use it for and how to do it. What do I mean by that? Say you had a list of numbers and you wanted to find the average of the numbers. In order to use this calculator, you would have to type in a long line of things that look a lot like

\[(5 + 3 + 4 + 2 + 4) / 5\]

all at once. The user has to know before even picking up the calculator the formula for calculating the average, how many elements they’re calculating the average for, how to add up all of the elements and how to place the parentheses. Also notice how what we type into the calculator is very dependent on our data set (our list of numbers).

To see this, let’s look at what we would type into Python (never mind the specifics of the syntax, we’ll take a close look at all of that jazz later).  

1

In[1]: list = [5,3,4,2,4]
In[2]: total = np.sum(list)

1If you’re trying to type this into Python, you’ll need one extra line before doing so: import numpy as np

11
In[3]: number_of_elements = len(list)
In[4]: average = total / number_of_elements

This is radically different from what we typed into our dollar-store calculator. Notice how much more readable our Python code is. If you were to look the dollar-store calculator input, you might be able to guess that we were calculating an average. With our Python code, every step is immediately apparent and readable. Obviously, when we get into programming much more complicated functions, it will be much more a challenge to keep this readability, but will be a constant goal that we will strive for.

A second thing to note about our Python input is that the only line of our syntax that depends on the actual data set is the first line. I can make the rest of the Python code work perfectly well with any other list of numbers (so long as np.sum and len “work” with any other list, which they do) simply by swapping out the first line with anything else. We’ll come back to this idea in the next chapter. A final thing to note is that we used a couple unfamiliar commands (np.sum and len). We’ll get into the mechanics of how these work, but the essence of them is in their names. This is as opposed to what we did with our dollar store calculator which is to type in each of the numbers ourselves to get the total of the list or to count up the elements to get the number of elements.

Of course, we are, in doing so, masking other lines of code that go into these functions np.sum and len. Writing code so that Python does counts up the number of elements for us or totals up a list for us may be, in general, more difficult and time consuming than simply doing it by hand. But once we do have these functions in Python, we have them ready to use. Luckily for us, most of these functions have already been written and already built into Python\(^2\), much like cd, ls, and mv were already built into our UNIX system. Python already has the tools for us to use. What we aim to teach throughout the course of this book is how to use them for our purposes in an efficient manner.

### 2.2 Python as a Calculator

We start Python the way we start any program we’ll use on the UGAstro server, by typing its name into the UNIX command line. For all purposes,

\(^2\)In fact, average itself is already a defined function in Python, called np.average
we’ll be using *ipython* rather than *python* since the ipython has extra features we’ll be using later.\(^3\)

```bash
> ipython
```

Now we’ll notice that the command line prints out some information about our distribution of python and that at the end we see that instead of the path that the UNIX interpreter normally has before any UNIX commands:

```bash
aquarius>/home/parriaga%
```

That we have a number

```python
In [1]:
```

This indicates that we are using a different interpreter. Instead of the UNIX interpreter, this is the Python interpreter. Only the UNIX interpreter understands the UNIX commands we’ve learned and only the Python interpreter will understand the Python commands we’ll learn.

We can start using Python as a calculator by typing math commands into our interpreter:

```python
In [1]: 5 + 5
Out[1]: 10
```

As per the example above, this is greatly helped by our ability to create variables, which allows us to carry out operations without being clunky with numbers. For example:

```python
In [2]: m = 10.
In [3]: a = 9.8
In [4]: f = m * a
```

However, we see that when we carry out operations like that, that the interpreter does not give us an “Out” or output as it does when we did 5+5, so we don’t know what f is equal to. This is in line 4, we are simply setting the value of f, which has no output. In line 1 where we carried out “1+1”, we are carrying out an operation which is meaningless without an output. Setting \(f = m*a\) is an operation in which we’re creating a new variable and setting its value, of similar nature to lines 2 and 3, which does not necessitate Python interpreter telling us what we just told it to do. In order to find out what f is equal to, we will use the print command:

```python
In [5]: print f
```

98.0

\(^3\)Certain servers on the UGAstro network do not have ipython. You may need to “ssh -X nemesis” or “ssh -X sirius” in order to start ipython
One subtlety to note about our operations with these variables is the ordering of the variable, equals sign, and the value that we would like the variable to be. Let’s illustrate this with some examples

---

**Example 2.1**

What would happen if we typed the following into Python?

```python
In [16]: x = 3
In [17]: y = 5
In [18]: x = y
```

What does x equal? What does y equal?

Let’s check the values with the print command:

```python
In [19]: print x
5
In [20]: print y
5
```

Both of the variables have assumed the value of y, rather than the value of x. This is because of the way that **variable assignation** works. This is to say, that when we assign a variable in Python (and in math, usually) that we will always tend to type:

```python
In [21]: x = 5
```

Rather than

```python
In [21]: 5 = x
```

In fact, the latter assignation gives

```python
In [21]: 5 = x
```

File "<ipython-input-21-94ca2abeea2f>" line 1

SyntaxError: can’t assign to literal

Which is our first **error message**. These indicate that something has gone wrong, in this case a **Syntax error**, which indicate that we have misspelled something or typed a command in the wrong order. The rest of the error message does not mean much to us yet, but we’ll get to that later.

---

**Example 2.2**

Let’s say we had two variables a and b:

```python
In [1]: a = 5
In [1]: b = 3
In [1]: print a
```

5
2.2. PYTHON AS A CALCULATOR

In [1]: print b
3

Now let’s say we wanted to swap the values (i.e. make a equal 3 and b equal 5). We can, of course, do it the easy way and just set:
In [1]: a = 3
In [1]: b = 5

But that’s not very general. What if we didn’t know the values of a and b to begin with (we’ll see cases later where we can’t just print the values and set them equal when we write procedures and functions). Well we can start out with:

In [1]: a = b

which sets a equal to the value of b, but that’s now we’ve lost whatever the value of a was to begin with, so we can’t set b equal to it. All we have now are two copies of b. To illustrate this, let’s introduce a silly analogy. Our variables are special boxes that can only hold one object. In our box a, we have a green circle, and box b we have a blue square. [NTD:some MS-paint figures] Now, say we have a special 3-D printer that can make an exact copy of anything inside a box. So we make a 3-D printer copy of what’s in Box B and we want to put it into Box A. Box A can’t hold more than one object, so in order to put our blue square into Box A, we’ll have to toss out our green circle.

So how do we remedy this? Well, the answer now, I hope, is clear. Before putting our copy of the blue square into Box A, we take our green circle and put it into a new Box C. That way when we toss out our green circle to put the blue square into Box A, we have a green circle to spare to put into Box B. So let’s do exactly this with some Python. First we make a copy of what’s in Box A and put it into Box C:

In [1]: c = a
In [1]: print a
3
In [1]: print c
3

Now we’re free to put the contents of Box B into Box A:

In [1]: a = b
In [1]: print a
5
In [1]: print b
5
Finally, we give \( b \) the previous value of \( a \), as stored in \( c \):

```python
In [1]: b = c
```

So now

```python
In [1]: print b
3
In [1]: print a
5
```
as desired

### 2.3 Data Types

If you play around enough with the arithmetic in Python, you may notice that Python seems to get it wrong sometimes. For example, if we try dividing a couple numbers

```python
In [1]: 1 / 2
Out[1]: 0
In [2]: print 5 / 2
Out[1]: 2
```

Python will round down, giving a pretty useless answer. Why does Python round? This has to do with the **data type** of the variables we’re looking at. The data type of a variable determines what values it can take, how much space it takes up on our computer, and what kind of operations we can do with it. In the case above, we are looking at a data type called an **integer**, which, like integers in math can only take whole number values (and negative whole number values). When we do operations between two variables of the same data type (the division is our operation in this case), we get a result that is of the same data type. Therefore, Python can’t give us back \( .5 \) for the first operation and \( 2.5 \) for the second because these numbers aren’t integers!

We can remedy this fairly easily, by adding a decimal point after:

```python
In [1]: print 1. / 2.
Out[1]: 0.500000
```

This converts our numbers into a different type of data called a **float**. In general, we’ll want to use floats to avoid **floating point errors**, in which we lose precision in our calculations because our data types are not precise enough for our needs.
Problem 2.2 Check the outputs of the following operations:
In [1]: print 2 / 3.
In [1]: print 2. / 3

Sometimes, though we’ll need bigger numbers than we have available in floats (especially if we’re dealing with astronomical quantities, which we’re fairly likely to do!). The next step up is a data type called a double. In order to tell Python we want a number to be a double, we have to cast it as this particular data type:
In [1]: x = double(5)
In [1]: print x
5.0000000
Our variable x does not look terribly different from a float. We could count the number of zeros in each case and notice that the double has more precision that way, but we can also use the help procedure
In [1]: x = 5
In [1]: help, x
In [1]: y = 5.
In [1]: help, y
In [1]: z = double(5)
In [1]: help, z

Problem 2.2 Check the data outputs of the following operations:
In [1]: x = double(4) / 3.
In [1]: y = double(1 / 2)

2.4 Strings

We’ll soon see that it’s useful to get Python to print words as well as simply printing the outputs of our variables. However, we can’t just have Python print words
As you see, this will result in an error. This is because Python thinks that we’re trying to print out a variable called “words”, but can’t find one. We indicate to Python that we’re trying to print out the actual words by putting quotes around the words we’re trying to print out:
In [1]: print 'What do you read, my lord'
What do you read, my lord
In [2]: print 'Words words words'
In light of our discussion before, when we put quotes around words, we’re turning it into a different data type called a string:

In [1]: x = ‘hello, world!’
In [2]: print x
hello, world!

### 2.5 Lists

Our final data type for this section, is actually not a data type, but a collection of other data types. This collection, which will show its infinite utility in coming chapters, is called a list or array:

In[1]: list = [5, ‘hello’, 12., 6, ‘goodbye’]

In this list, we have 5 elements and we can reference each one using its index. *In every programming language, array indexing begins at 0.* Therefore if we print

In [7]: print list[0]
5
it prints out 5. We will work more on lists and arrays in the following chapters.

### 2.6 Packages and NumPy

As of now, our functions with Python are rather limited by the number of functions that we have. We basically only have operations that we can do on a dollar store calculator. However, the utility of programming languages is that many of the functions that we would like to carry out on a fancier calculator are already written; we just need to bring them into Python. We do this by importing libraries and modules from other sources. The first that we will be using is numpy, which has math functions. In order to import numpy, we simply use the command:

In[8]: import numpy

Now, for example, we can use:

In [4]: numpy.sqrt(2.)
Out[4]: 1.4142135623730951

or
2.7 Summary

2.7.1 Definitions

- **Integer** - A number variable without decimal points
- **Float** - Another number variable with decimal points
- **Array** - A collection of several objects with the same variable type
- **Data Type** - Describes the properties of a variable (e.g. integers, floats, arrays are different data types)
- **Casting** - Turns a variable from one data type to another

2.7.2 Important Functions and Procedures

- **print**arg1 - Takes arguments of any data type and prints them in the command line
- **float**(arg1) - Casts arg1 as a float.
- **fix**(arg1) - Casts arg1 as an integer
- **string**(arg1) Casts arg1 as a string
2.8 Tutorial

In this tutorial, we will be practicing doing basic math operations and getting used to the syntax that we’ll be using in Python. The first thing that we’ll be doing is defining some constants within Python. Since we are astronomers, we’ll be using some fairly large numbers, and therefore using a lot of scientific notation. Define some variables by typing the following lines:

In[1]: c = 3.e8
In[2]: m_hydrogen = 1.67e-27

The rest energy of an atom is gained by:

\[ E = m_0c^2 \]  \hspace{1cm} (2.1)

Define a new variable \( E \) that has the value of the energy of hydrogen using:

In[3]: e = m_hydrogen * c**2.

Remember that the “c**2” means take \( c \) to the 2nd power.

Create a directory for the week 2 tutorial. In this directory open up an emacs file called “tutorial” with:

`username>/week2/tut/% emacs -nw tutorial`

Remember that this creates a text file with the name “tutorial”. Whenever you open up emacs, follow it with the name of the file to open that file. In this file, you will write the values derived from performing the following tasks at the command line:

1. In the first line of this document, type the energy of hydrogen. Save this file with Ctrl+x Ctrl+s.

2. One of the most irritating conventions in astronomy is the magnitude system. The magnitude of a star is a description of how bright the star is. A smaller magnitude means a brighter star and vice-versa. We can calculate the apparent magnitude of a star using the following formula:

\[ m = -2.5 \log_{10} \left( \frac{F}{F_0} \right) \]  \hspace{1cm} (2.2)

in which \( F \) is the flux of the object that we’re considering, and \( F_0 \) is the flux of Vega in the V-band, which is not only an arbitrary zero point but is also a variable star so not technically a stationary one.

The flux of Vega is \( 3.636 \times 10^{-20} \text{erg cm}^{-2} \text{s}^{-1} \text{Hz}^{-1} \).
If a star has a flux of $2.1 \times 10^{-20}\text{erg cm}^{-2}\text{s}^{-1}\text{Hz}^{-1}$, then what is its apparent magnitude? Note that this uses log based 10, so numpy.log (which is the ln function) would be the incorrect function to use. Google the correct numpy function to do this operation. Put this magnitude in the second line of your tutorial document.

3. One function you’ll be using a lot no matter what field of astronomy you enter is one that reads in data. Let’s get a little practice reading in and manipulating data; we’ll be covering this in a lot more depth later on. Copy the following file into your week 2 tutorial directory:

```
/home/parriaga/public_html/python/week2/data.dat
```

.dat files are basically just text files containing our data. You can open these in emacs with

```
emacs data.dat
```

Doing so, we see a list of numbers. There are quite a few numbers in this file, but programming allows us to manipulate the data without directly interacting with it.

Now, in Python let’s read in the file using the function:

```
In[0]: dat_arr = numpy.loadtxt('data.dat')
```

This reads in all of the numbers from data.dat and puts it in an array or list. Remember for the following problems that array indexing begins with 0.

4. Find the number of elements in this array, and therefore in the data set and put this number in your tutorial document.

5. Find the 401st element of this array and put its value in your tutorial document.

6. Find the sum of the 60th through 190th element of the array and put its value in your tutorial document.

7. One important skill for writing programs for pipelines is the manipulation of filenames by manipulation of strings. Take the following filename:

```
1040_20111013_Aldebaran_0.fits.gz
```
Extract the first 4 numbers and assign them to a variable `init`, the next set to `date` and Aldebaran to `star` using slicing and indexing of the strings. Concatenate these strings together along with the string 'processed.sav', separated by dashes, and save this string to the last line of your tutorial.
Chapter 3

Writing Simple Programs

While the Python interpreter in tandem with our imported libraries give us a powerful tool to do mathematical operations, we are limited by the fact that all of the operations we carry out are erased from the computer’s memory when we exit the program. Additionally, if we mess up and make a mistake, there is no way to “undo” our operations. True programming involves writing our commands for Python to a saved file that Python will then read and carry out, rather than working solely from the command line. Now, if we make a mistake in writing our commands, we can alter them and rerun them with very little effort.

These files are called Python scripts and they are marked by the file extension .py. We can start a new script in any text editor:

```bash
emacs hello_world.py
```

Remember to always begin your text file with a filename ending in .py. Otherwise, we won’t have font color highlighting. Here, we write our commands:

```python
print 'hello world'
x = 5
print x + 10
```

Now, we want Python to run this file. We do this by going into Python and using the run command:

```
In[1]: run program.py
hello world
15
```
3.1 Basic Functions

Another utility of scripts is that they allow us to write our own functions, like those in the numpy library that we used before. This is very useful to us as scientists because our goals usually require far more specific functionalities than most of the functions found in a package provide. For example, we may want to write a function that takes an image of a star and spits out the flux of this star. Whereas functions like log or sqrt are useful in many applications, our function has limited applicability to the whole community of Python programmers, and therefore likely does not come with a pre-existing library.

In order to know how to go about writing a program to solve a research problem, we need to truly understand what is meant by a function in Python. As can be ascertained from the previous chapter, functions in programming work in close analogy to functions in mathematics. Take, for example, the function:

\[ f(x) = x + 5 \]  

(3.1)

From this function, we can evaluate it, for example, for the value \( x = 5 \):

\[ f(5) = 5 + 5 = 10 \]

In the language of programming, the function \( f(x) \) takes \( x \) as an argument, and returns a value of 10. We can have a more complicated function such as:

\[ f(x, y) = x + y \]  

(3.2)

For this function, we have two inputs, \( x \) and \( y \). Just for good measure, we evaluate it for \( x = 4 \) and \( y = 3 \):

\[ f(4, 3) = 4 + 3 \]

Going simpler, we can have a function that takes no arguments at all

\[ f() = 5 \]  

(3.3)

which will always be 5.

The properties of mathematical functions will be similar to the properties of our functions in Python in a number of ways. First, Python functions can take any number of arguments, mess around with them in some fashion, then return a value. Second, notice that when we write our functions such as “x
3.2. MORE ADVANCED FUNCTIONS

+ y” or “x + 5”, we don’t refer to what we’re going to plug into it until after we have defined our function. This may not make sense until after seeing a function or two, so let’s try writing a function for Equation 3.1.

We begin our functions with the following syntax:

```
def addfive(x):
    "def" stands for definition and tells Python that we’re starting a function. "def" is always followed by the name of our function. This is the name that we will be using to call the function when we use it on the command line or in a program. The final component is the argument, which is always listed between parentheses (Even if your function has no argument, you still must include these parentheses). Finally, there is a colon to indicate the beginning of the function.

    After this line, we write the steps describing what we want the function to do. In this case, we want the function to add 5 to x and return the result back to the user. The syntax is as it sounds:
```

```
def addfive(x):
    result = x + 5
    return result
```

Note the indentation after the “def” line of the function! In Python, indentations are used to separate out blocks of commands and to relate them to the lines above and below. By indenting the commands following the “def” line, we are indicating that these commands should be executed when the function addfive is called.

3.2 More Advanced Functions

Similarly to our example in the last section, if we tried writing a function for Equation 3.2, we would write:

```
def addy(x, y):
    result = x + y
    return result
```

Note that in the above function, we separate our arguments by commas and include both arguments within the parentheses. It is important to remember that when we call our functions, the order of these arguments matters. In this function, if we provided y for x and x for y, no damage would be done,

```
but what if instead we had been writing a function in Python for the function $f(x, y) = x + y^3$?

Now it’s your turn, how would you write Equation 3.3?

We could also decide that for our uses of the function, we would like to use a more general function like addy, but have its default behaviour be to act like addfive. How would we do this in Python? Luckily, Python provides us with the ability to define a function with default arguments. To write the function as we just described it, we would write:

```python
def addy(x, y=5):
    result = x + y
    return result
```

As written, when two arguments are given to addy, the result will be their sum, and when one argument is given, the result will be the sum of that argument and 5.

Finally, one other useful ability that python provides us with for writing functions is the ability to include keywords. One way to use keywords is similar to how we defined a default argument above. If we set a variable equal to some value in the function’s arguments, that particular argument becomes optional, and we can have the function respond to whether or not that value is changed when the function is called. When we set keywords in this fashion, we can include keywords in any order (or not at all), but we must call (calling in programming is equivalent to evaluating a function in math) the function as:

```python
In[1]: y = function(argument, keyword = value, keyword2 = value2)
```

In this situation, if we just include another argument after “argument”:

```python
In[1]: y = function(argument, value)
```

“keyword” will be set to that value, because the arguments are taken in the order of their position in the “def” line. If we wanted to set the value of “keyword2” without changing “keyword”, how would we call the function?

Alternatively, we can include the ability for a user to include any number of additional arguments and keywords by defining their function as follows:

```python
def function(x, *arguments, **keywords):
```

where “**keywords” must always follow “*arguments”. By using these special kinds of arguments in your functions, you will have all additional arguments given by the user included in a tuple called “arguments”, and all additional keywords (which are differentiated from arguments by being called
3.3. WRITING PROGRAMS

As “variable = value”) set to a dictionary called “keywords” with keys that match the keyword names given by the user. This alternate method of providing the user with the ability to provide optional input goes beyond the scope of our class (at least at this point in time), however, so if you are interested in this topic, we refer you to the next chapter for definitions of tuples and dictionaries and the literature (and internet) for examples of this method in action.

3.3 Writing Programs

As mentioned in the introduction to this chapter, we write scripts because usually the problems that we want to solve benefit from us being able to save the results, interactively edit our commands, and write our own functions. Another benefit of writing scripts is that they perform our commands much more efficiently by executing them in a linear order, with each line being executed as soon as the previous command is finished. For very simple problems, this is much faster than having you type each command into the command line one after the other, while for extremely complex problems that require a large number of calculations to be done, this may be useful because it frees you from waiting for each step to finish (which may take from minutes to days).

For more complex problems, it is often necessary to define more than one function in your script, and to use a combination of functions you have written and functions included in pre-existing libraries generated by the greater Python community.

Example 3.1
If I wanted to find the exit angle and the change in speed of a ray of light after transitioning between two media based on the entrance angle and the indices of refraction for both media, I would need one function that used Snell’s law to calculate the exit angle, and another function that calculated the speed of light in each media and found the difference between the two. If it was useful to me to perform these calculations often, I might write a script like this:

```python
import numpy as np

angle1 = np.pi / 3.
```
n1 = 1.2
n2 = 1.5

def exit_angle(theta_1, n_1, n_2):
    sin_theta_2 = n_1 * np.sin(theta_1) / n_2
    theta_2 = np.asin(sin_theta_2)
    return theta_2

def speed_dif(n_1, n_2):
    c = 2.998e8
    v_1 = c / n_1
    v_2 = c / n_2
    delta_v = v_2 - v_1
    return delta_v

print 'Exit angle is: '
print exit_angle(angle1, n1, n2)
print 'Change in speed is: '
print speed_dif(n1, n2)

With this script, I could change the values of my inputs directly in my file before I ran my program.

---

**Example 3.2**

Alternatively (and more conventionally), I could write a **main-level function** in place of the print statements in the example above that I could provide a call to directly at the command line.

```python
def main(angle1, n1, n2):
    print 'Exit angle is: '
    print exit_angle(angle1, n1, n2)
    print 'Change in speed is: '
```

```
print speed_diff(n1, n2)

Note that if I was running my script as defined above, it would be very important for my print statements to follow my function definitions, because otherwise I would be calling my functions before they had been defined. In contrast, if I wrote a main-level function, I would have to import the compiled module in a manner similar to when we use numpy, prior to use, so the functions would already be defined, and the order in which I wrote them would not matter.
Chapter 4

Data Structures

For this chapter, we’ll be using a lot of numpy functions, so be sure that every time you start your session of iPython that you

```python
import numpy
```

In general, a data structure is simply a collection of variables. In Python, we have many different data structures available to us. We’ll be concentrating on lists, dictionaries, and NumPy arrays. We’ve already seen lists and arrays, so let’s review them quickly. A list is a collection of any kind of variable. We can have lists of integers:

```python
In[1]: a = [1,2,3,4,5]
```

Lists of strings

```python
In[1]: b = ['spam', 'baked beans', 'spam', 'eggs']
```

Lists of both

```python
In[1]: c = ['spam', 3, 'hello', 4.]
```

And even lists of lists

```python
In[1]: c = [['spam', 3, 'hello', 4.], [1,2,3], [4,5,6]]
```

```python
In[2]: c[1]
Out[2]: [1,2,3]
```

Numpy arrays are fairly similar. We create numpy arrays in the same way that we make lists except we declare it to be a numpy array using the function `numpy.array()`:

```python
In[1]: array = numpy.array([1,2,3,4,5])
```

```python
In[1]: print array[0]
Out[2]: 1
```

Let’s look at some of the similarities and differences between lists and numpy arrays
4.1 Array Operations

An operation between two arrays is similar to an operation between two numbers. The operation $2 + 2$ gives us the result 4 and the plus sign indicates that we add the numbers. In python, the plus sign has two different meanings, one for operations between numpy arrays and one for operations between lists. If we have two lists:

```python
In[1]: a = [1,2,3,4]
In[2]: b = [5,6,7,8]
```

and we add them using the plus sign, the result is the **concatenation** of the two. Concatenation is a fancy term for sticking two lists together. Our result is

```python
In[3]: a + b
Out[3]: 1 2 3 4 5 6 7 8
```

Numpy arrays behave a little differently because they’re designed to hold numbers and to allow us to manipulate numbers. If we add two arrays

```python
In[1]: a = numpy.array([1,2,3,4])
In[2]: b = numpy.array([5,6,7,8])
In[3]: a + b
Out[3]: [6 8 10 12]
```

This results in an array which is the same size as the original array. This operation goes element and element adds the pairs together.

---

**Example 4.3**

---

### 4.2 Creating Arrays

Typing out these arrays can be very tedious especially when we want an array that is very large. For this we have special functions that can generate which can get handy when arrays reach thousands to millions of elements. The two functions that we will use are the `numpy.arange` and `numpy.zeros` functions. Both of these functions take one argument which is the number of elements that we want our resulting array to be. The `numpy.arange` function will create an array of our desired number of elements counting up from zero to the number of elements minus one:
In [1]: x = numpy.arange(3)
In [1]: print x
0 1 2

The `arange` function creates an array of zeros:
In [1]: x = numpy.zeros(3)
In [1]: print x
0 0 0

However, at first glance these arrays do not seem very useful since they are very specific functions. By performing operations on these arrays we can mold them to our purposes. This is very much facilitated by the fact that any operation to an array is carried out over element. For example:
In [1]: x = numpy.arange(3)
In [2]: print x
0 1 2
In [3]: print x + 2
2 3 4

or
In [1]: x = numpy.zeros(3)
In [2]: print x
0 0 0
In [3]: print x + 3
3 3 3
In [1]: y = (x + 2) * 5
In [1]: print y
10 10 10

In principle, we can do anything to an array that we can do to any given variable. But let’s say we want to manipulate a single element. For this we use the array indexing we used before. Using our array `y` above and remember that array indexing starts from zero:
In [1]: y[1] = 4
In [1]: print y
10 4 10

4.3 Tutorial

For this tutorial, we will be creating some various arrays. In your tutorial document for this week, then please type out the commands by which you
would generate the following arrays. Check as you guy by generating them in Python. Then copy the output of the array and paste it under the command. This will give you a great reference document once we start generating arrays for plotting. For none of these should you have to type more than 4 numbers. Le none of your commands should look like:

In [1]: x = [1,2,3,4,5,6,7,8,9]  
And should look more like  
In [1]: z = findgen(8) + 1

1. Generate an array beginning with 0 and ending in 7 using the findgen function. Set this array equal to x:

[0, 1, 2, 3, 4, 5, 6, 7]

2. Now, let’s make a new array, call it y, equal to our x array with 5 added to each element. Remember that mathematical operations on an array are carried out on each element. Y should have the numbers:

[5, 6, 7, 8, 9, 10, 11, 12]

3. Let’s jump the gun a little bit and just use the following command. What does it output?

import matplotlib.pyplot as plot  
plot.plot(x, y)  
plot.show()

4. So now let’s make a new array of 8 zeroes. You will want to use the numpy.zeros command

[0,0,0,0,0,0,0,0]

5. In the same way that we manipulated the previous array, turn this array into an array of 7s and call it z

z = [7,7,7,7,7,7,7] 

6. Now type in the following commands:

import matplotlib.pyplot as plot  
plot.plot(x, z)  
plot.ylim([0,10])  
plot.show()  
Can you explain what’s going on in this plot?
Chapter 5

Control Statements

So far, the tools that we have learned how to use give us the power to manipulate an input and turn it into a usable output. In principle, we could stop here and replace the rest of the book with a list of procedures and functions, describing their data inputs and outputs and how to use those results. However, keep in mind that in astronomy we don’t just work with single images that we can input into a function to get some result, we have billions to run through whole series of functions. We could sit at the terminal and individually input each filename into our pipeline...or better yet, write a procedure with all of the filenames in it and copy and paste the same line with a different file name every time. As you would soon see after your fourth or fifth iteration of copy and pasting, this is vastly inefficient, and worse, leaves our pipeline vulnerable to human error. Instead, we should make Python do it for us using control statements (specifically loops).

Control statements are useful in any case where you need to control the flow of your program more explicitly than is the default. By default, when a program is run, the commands will be executed in a linear order from left to right and top to bottom (as we have emphasized many times in this text). In the interest of readability or efficiency, however, it often is useful to circumvent this strict linear interpretation by modifying the flow to repeat similar steps many times with slightly different parameters, or to execute different sets of commands based on whether your intermediate data products meet certain conditions.
5.1 For Loops

When you wish to perform the same action over and over again with minor alterations in a program, your best option is often to use a for loop. A for loop is written somewhat similarly to the beginning of a function, and also makes use of tabbing for organization. Instead of writing,

```python
def function_name(input):
```

you will begin by writing,

```python
for i in xrange(0, n):
```

where the special command `i` for designates the commands following as part of a for loop, `i` is the iterative variable that increases by 1 (by default) each time, and `xrange` is a function that specifies the numbers that `i` will iterate through, from 0 to `n-1` in increments of 1. To iterate by a different amount, you can give `xrange` a third argument that is the size of the increment `i` increases by. You should consider, however, that complicated increments (like fractions of \( \pi \)) will often slow down your program, so it may be more useful to achieve the effect of these fractional increments by different means.

5.2 If Statements

5.2.1 Conditionals

Before we get into using if statements, we first need to discuss the conditional operators that we will use to specify our conditions. At their most basic, conditional operators are symbols or words used to evaluate the relationship between two values. You actually already know most of the conditional operators (in concept if not in form) from 1st or 2nd grade math. The basic conditional operators are:

- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to
- `==` equal to
- `!=` not equal to
5.2. **IF STATEMENTS**

Note that when evaluating equality (in some languages we might say identity, but this can get a bit complicated in Python), we must use `==` and `not =`, which is specifically reserved for assignment. If you need to check more than one condition, it will be necessary to use more advanced operators, such as: To a computer, True is equivalent to 1 and False is equivalent to 0,

```
  and   true if both statements are true
  or    true if one statement is true
  not   reverses the evaluation of a statement
```

and all conditional relations are applications of Boolean algebra. With this understanding, the `and` operator can be interpreted as multiplication of the truth values, while the `or` operator can be interpreted as addition followed by another evaluation for a non-zero value.

### 5.2.2 Using Conditionals

In their most basic implementation, conditionals generate a truth value output from the evaluation of a specific set of restrictions. A simple example would be printing the output to the command line. If you chose to write, 3 < 4
to the command line, the output would be,

```
In [1]: 3<4
Out [1]: True
```

and if you chose to use the first output algebraically, it would act as a 1, even though it will read “True” when printed to the screen. For a slightly more advanced example, pretend you were given a dataset and told to zero out any region with noise below a 1σ cut. While you can use some of the other techniques from this chapter to perform this task iteratively, conditionals make it much easier. When using a conditional, you can evaluate entire arrays in a single step, and generate a Boolean mask as output as follows,

```
boo_arr = arr < cut_plus and arr > cut_min
```

`boo_arr` is now an array of True and False (1s and 0s), and can be used to zero out values outside your range as,

```
new_arr = arr*boo_arr
```
More typically, you will be using conditionals in if statements. As an example, imagine that you were constructing a pipeline to process raw data into a useable form. During this process, you might need to screen the data and throw out bad data points based on whether the datum fell within an allowable range. If we called this datum x, we could use the if statement,

```python
if x > val1 and x < val2:
```

where the statement structure is similar to the for loop, except that it begins with `if` and uses `and` to indicate that both cases must be true to proceed through to the commands indented in from the if statement.

### 5.2.3 Elif and Else Statements

Often when you are using an if statement you will have multiple possible methods for treating your data product based on the conditions it meets. In this case, you can approach the problem in a way more sensitive to the multitude of possibilities by adding more statements after an if statement to provide different options and other conditions to check. For example, if I had two different sets of commands based on whether a variable a was less than or equal to 0, I might write,

```python
if a < 0:
    #do this set of commands
elif a == 0:
    #do this other set of commands
else:
    print "Your a is positive?"
```

where `elif` specifies another condition to check, and `else` provides a set of commands to follow if none of the checked conditions are met.

---

**Example 5.1**

Let’s make sure we understand our boolean logic by thinking through an if/else tree. Consider the following program

```python
def ifelse(number, string):
    if number == 5 and string == 'hello':
        print 'Once upon a midnight dreary'
    if number > 3 and string == 'hello':
        print 'As I wandered weak and weary'
```

---
else:
    print 'Over a quaint and curious volume'
print 'Of forgotten lore'

1. What does the program print if number = 5 and the string is 'hello'?
2. What about if the number is 4 and the string is 'hello'
3. What if the number is 5 and the string is 'goodbye'
4. What if the number is 3 and the string is 'hello'

5.3 While Loops

Sometimes you will find yourself in a situation where you do not know how many iterations you will have to perform to reach your desired data product. Usually in these situations, rather than knowing an exact number of steps that must be performed, you will know a condition that must be met before the process can end. In this situation, you will need to use a while loop. A while loop is a lot like a for loop, except that rather than specifying how an iterative variable will grow each time the loop is repeated, you will specify when looping will be terminated. Every time a while loop is repeated, this condition will be checked. For example, if we wanted to have our program iterate until the result converged (as we might if we were trying to figure out the amount of ionized hydrogen in a star, say), we would use,

```python
n_HI_old = 0
n_HI = 1e6
while np.abs(n_HI-n_HI_old) < 1e3:
    n_HII = saha(n_HI, m_star, t_star)
    n_HI_old = n_HI
    n_HI = n - n_HII
```

Note that when we use while loops it is important to define your iterative variable outside of the loop to begin with, because the while loop will check the condition at the beginning of every iteration. It is also important to make sure your iterative variable is changed within the loop, otherwise you will enter an infinite loop because the condition to exit is never met.
5.4 Returning to normal flow

Altering the normal flow of a program’s interpretation is all well and good, but how do we get it back to normal? Typically, when the commands within a control statement are finished (i.e. the number of iterations specified in a for loop are completed, or a while loop breaks the condition that keeps it iterating), normal program flow will resume. There are occasions, however, where it will be necessary to force the interpreter to break out of a loop. For example, if in the while loop example above my result started deviating from the desired convergence in a runaway fashion, I would prefer to exit the loop rather than iterate uselessly. To achieve this end, I could write,

\[
\begin{align*}
\text{n.HI.old} & = 0 \\
\text{n.HI} & = 1e6 \\
\text{while} & \quad \text{np.abs(n.HI-n.HI.old)} < 1e3: \\
& \quad \text{n.HII} = \text{saha(n.HI, m.star, t.star)} \\
& \quad \text{n.HI.old} = \text{n.HI} \\
& \quad \text{n.HI} = \text{n} - \text{n.HII} \\
& \quad \text{if} \quad \text{n.HI-n.HI.old} > 1e8: \\
& \quad \quad \text{break}
\end{align*}
\]

The command \texttt{break} causes python to exit the loop its currently in. It is less efficient to have a program check two conditions every iteration, but occasionally these break statements are very important for making your program robust to error. Note that in the situation that you have nested loops (a for loop within a while loop, or anything like that), a break statement will only break out of the local (smallest level) loop, not out of the entire nested series of loops.

5.5 Tutorial

At this point in your Python-learning journey, you’re finally starting to reach the point where you can start to tackle real astrophysics applications. To that end, this tutorial will guide you through a relatively simple task in astronomy...finding the equivalent width of a spectral line. Write all of your functions into one file names tutorial5.py.

Equivalent widths are often used in empirical relationships relating spectral features to stellar properties. The feature you’ll be looking at in particular is the calcium triplet feature of a blue supergiant star in the galaxy IC 10.
Although this relationship is not extremely accurate for blue supergiants in particular, Battaglia et al (2008) have shown that there is a correlation between the equivalent widths of the calcium triplet, the g, r and i photometric band intensities, and the overall metallicity of the star.

Now, you may ask, what is the equivalent width of a spectral feature? Figure 5.1 shows it best, but in words, the equivalent width is the width of a spectral feature that when multiplied by the height of the continuum will generate the same area as the integrated area of the feature itself.

![Figure 5.1: The equivalent width of a spectral feature (Credit: Szdori, Wikimedia Commons)](image)

1. To start, we will need to obtain our spectrum. Copy CaT_spec.npz from /home/bbordwell/decal/python to the directory in which you are working on this tutorial. Use the following lines to load in the data and get arrays with wavelengths and intensities.

```python
data = np.load('CaT_spec.npz')  #loading in the data
data_arr = data['data']  #getting the data in an array
lamb = data_arr[0]  #Angstroms
spec = data_arr[1]  #Photon number
```

2. To begin analysis, we will need to subtract off the continuum offset. A nice way to do this is to subtract a median-smoothed version of the
spectrum from your raw spectrum... but to do that you’ll need a median smoothing function! You can write this function as follows

- Name your function mboxcar (we’ll be using a boxcar smoothing method) and give it the two inputs arr and width (the array to be smoothed, and the width of the boxcar).
- Generate a new empty array called new_arr to hold your smoothed array
- Define a range for the for loop to iterate through. Remember that as you will be taking the median of box centered around a point of width width, you will need to leave half a box (width/2.) of room on both sides of your range. (i.e. don’t range from 0 to len(arr), but instead...)
- Start a for loop using the variable i to iterate through your range.
- In the for loop, set the ith element of new_arr to the median of a box running from the i-width/2. to i+width/2.-1 elements of arr.
- At the end of your function, return new_arr.

When you’re done, your function should look something like this:

```python
def [FITB]:
    new_arr = np.[FITB](len(arr))
    # create new array to hold result

    rng = [FITB](width/2., len(arr)-width/2.)

    for i [FITB]:
        new_arr[i] = np. [fill in the blank] (arr[ [FITB] ])
        # take the median of the boxcar around each point

    [FITB] # return the smoothed array
```

Note that wherever you see a [FITB] (fill in the blank) you must replace that item with the correct code.

3. Find the area of your spectral feature by integrating the area of that part of your continuum-corrected spectrum using numpy.trapz (look up the inputs you’ll need using help(numpy.trapz)). Your command should look something like:
area = np.trapz([FITB])

You will need to integrate only the spectral feature when you are finding its area. For this tutorial we will look at the spectral feature between 8538 and 8542 Angstroms. One way to isolate this feature is to pull the feature out of the spectrum using a single line for loop. You can use these lines in your code to do this (Don't worry too much about the syntax here unless you feel comfortable exploring it, it's rather complicated):

```
fraction_indices = [i for i in range(len(lamb))
    if lamb[i] < 8542 and lamb[i] > 8538]

lamb_feat = lamb[fraction_indices]
spec_feat = spec[fraction_indices]
```

4. Estimate the height of the feature as the difference between the maximum and the minimum of spec_feat. You can find your equivalent width by dividing the integrated area by this height.

Your final script should look something like this (imagine filling in where the comments are with your code):

```
# import statements at the top
import numpy as np
import matplotlib.pyplot as plt

# your boxcar smooth function goes here

# load in your data using the lines in step 1

# smooth your spectrum with a width of 30

# plot your spectrum to see the raw data
plt.plot(lamb, spec)
plt.show()

# subtract your "continuum" (smoothed spectrum) from your spectrum
```
#plot your spectrum to see the corrected data
plt.plot(lamb, spec)
plt.show()

#isolate the second calcium feature
#using the lines in step 3

#plot your feature
plt.plot(lamb_feat, spec_feat)
plt.show()

#find the area of the feature using np.trapz

#find the approximate equivalent width

#return the equivalent width of the spectral feature

## 5.6 Problems

1. Write a function that returns the n-th root of an input number. You will input the number and the "n" for which root you want to take. If the user inputs a negative number, return 0.

Your final output of this function should look like:

```python
print n_th_root(8, 3)
2.00000
```
since 2 is the cubic root of 8.

2. Find the sum of all the multiples of 3 or 5 under 10,000

(a) Using for and if statements.

**Hint:** In order to find the mod (remainder) of two arguments, the syntax is:

```python
a % b
```

(b) Using a while loop instead of for loop.
Chapter 6

Advanced Arrays

6.1 Multidimensional Arrays

In Chapter 4 we introduced the concepts behind working with simple one dimensional arrays and the powerful abilities they conferred upon your programs in terms of performing operations upon groups of objects. In this chapter, we shall expand upon this topic by introducing the topic of multidimensional arrays, which, through increasing complexity, enable you to perform even more complicated operations in a more efficient manner. In addition, the addition of another dimension to arrays opens up the possibility of using matrix math in your programs, which is a very powerful way to solve linear systems.

In Python, to create a multidimensional array, you may use most of the commands we’ve learned so far to generate an array (not arange, but np.array, np.zeros, np.ones, np.empty, etc.). To work with two dimensions, when providing the input for the number of elements in the array you must instead provide a tuple,

$$(\text{# of columns}, \text{# of rows})$$

To create arrays of even greater numbers of dimensions you would simply add more elements to the tuple. If you want to use a multidimensional array with values like those produced by arange, you can achieve the same effect by first initializing a single dimensional arange array (call it $x$) of $(\text{# of desired columns})*(\text{# of desired rows})$ elements, and then using the command reshape with a tuple input,

$x$.reshape$((\text{# of columns}, \text{# of rows}))$
6.2 Array Sub-scripting

Subscripting a two-dimensional array in Python is analogous to the way you learned to refer to elements within a matrix in linear algebra. As was the case when we worked with single dimensional arrays, the indices also still will begin with 0. If you provide only one index, you will receive the one column of your array,

\[
x = \begin{bmatrix}
1 & 2 \\
0 & 3 \\
\end{bmatrix}
\]

(6.1)

\[
x[0] = \begin{bmatrix}
1 \\
0 \\
\end{bmatrix}
\]

(6.2)

If you specify two indices, you will first specify the column number, then the element within the column (the row number), as,

\[
x[0,1] = 0 
\]

(6.3)

If your arrays are larger than two dimensions, your indices will be similar, except that your first index will now specify the slice chosen out of the outermost dimension, and your last index will still specify the elements within a column. For example, suppose you had a cube of information as a stack of these 2 arrays (\(x_0\) as the bottom of the stack, \(x_1\) as the top),

\[
x_0 = \begin{bmatrix}
1 & 2 & 9 \\
0 & 3 & 9 \\
\end{bmatrix} \quad x_1 = \begin{bmatrix}
4 & 5 & 99 \\
8 & 7 & 10 \\
\end{bmatrix}
\]

(6.4)

to specify the 99 in the three-dimensional array, you would provide the indices \(x[1,2,0]\).

6.3 Array Concatenation

If I was provided the two two dimensional arrays \(x_0\) and \(x_1\) defined in the last section, I could concatenate them into the three dimensional array \(x\) using the numpy command vstack and a multidimensional list input,

\[
x = \text{np.vstack (} \text{[[x_0],[x_1]]})
\]

To concatenate within the dimensions already present in the array, you can also use the concatenate command,

\[
x = \text{np.concatenate (} ((x_0,x_1), \text{ axis } = 1)
\]
6.4 Other NumPy Array Functions

As mentioned at the beginning of this chapter, one of the powerful opportunities that multidimensional arrays offer you is matrix math. In NumPy you will find that there are whole sets of functions designed to this end such as np.dot, np.transpose, etc. Generally, you should always use NumPy when working with arrays, and should try to use np.linalg functions when performing matrix operations. There are many references that will go deeper into this material, and we will refer you to the documentation on these functions in lieu of rephrasing it in depth here. The one caution we will offer, however, is that it is important to carefully evaluate the way each function operates, as your initial assumptions as to what might be being done could be incorrect.

Three examples of this are the np.dot function, np.invert function, and the np.transpose function in the case of one dimensional arrays. In the case of the np.dot function, to perform the operation,

\[ a = X \cdot Y \]  

where \( X \) and \( Y \) are two matrices, the Python command would be,

\[ a._t = \text{np.dot}(Y,X) \]

where \( a._t \) is the tranpose of the \( a \) in the above equation. Although you would expect the dot product to act analogously to normal matrix math, Python fails you in this regard. Similarly, when taking the inverse of a function, it is necessary to use np.linalg.inv, rather than np.invert, to achieve the linear algebra version of an inverted matrix. Finally, although your mathematical...
intuition would lead you to suppose that taking the transpose of a column vector would produce a row vector, in Python it is necessary to use reshape to achieve this transformation, as np.transpose will simply return your input unchanged.

6.5 Tutorial

One very important function in any scientific work is a regression to data. If you have not yet taken a linear algebra class, you may not know how to do this automatically, but you know the general idea of it. If we have a set of data such as in Figure NTD, we can see that the data is pretty linear and try to eyeball the general trend of the data and use a ruler to draw a line through where we think it should be. This is easier if we have a data set that has a pretty tight trend such as Figure NTD, but can be very ambiguous in noisy data such as Figure NTD. Our ruler and eyeball are good for data such as Figure NTD but can be very subjective for data such as Figure NTD. To make matters worse, our ruler and eyeball method only works in the case that we have linear data. For a scientific work

6.6 Problems

1. Write a function called swap_em. This function should take a 2-element array. It should return an array with the two elements swapped. For example, if you input the array [1,5], swap_em will return the array [5, 1].

2. ⋆ Generate the matrix. Don’t type in any numbers, just use numpy.arange and numpy.zeros

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
0 & 0 & 0 & 0 & 0 \\
10 & 11 & 12 & 13 & 14 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\] (6.7)

3. ⋆ Create a function called centroid which will find the centroid of a certain function. A centroid is found by:

\[
\sum_{i=0}^{n} x_i y_i
\] (6.8)
4. Translating quicksort

```plaintext
function quicksort(array)
    if length(array) 1
        return array // an array of zero or one elements is already
        sorted
    select and remove the first element from the array
    create empty lists less and greater
    for each x in array
        if x pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), list(pivot),
    quicksort(greater)) // two recursive calls
```
Chapter 7

Good Programming Practice

This is arguably one of the most important chapters and likely to be the most ignored chapter. However, good programming practice is one of the biggest time-savers in coding. You will find that, once you get the hang of the basics, that most of the difficulty you encounter in working with code will come from interpreting others’ code and rereading your own code long after you have written it. Rather than giving a laundry list of good programming practices, much of this chapter will be taught through examples.

7.1 Formatting

Example 7.1

Here is a silly and fairly useless piece of code. There’s an error in it somewhere:

```python
def slow():
    a = np.zeros(1000, 1000)
    s = a.shape
    for i in range(0, s[0]):
        for j in range(0, s[1]):
            if i+j < 90:
                a[i, j] = sin(i+j)
    for i in range(0, s[0])
        for j in range(0, s[1]):
            if a[i, j] != 0:
```

51
Notice how this code looks unlike a lot of the code that we’ve presented throughout. First of all, the variables have been very unhelpfully named as a, i, j, and b. There’s not much indication of what any of these variables are. Additionally, this function has no documentation, so while we may deduce a little bit about what it does based on its name, we would generally be at a loss as to why it is “slow”, what the purpose of it being slow is, and how it achieves this effect. When we attempt to run this function, we get the rather helpful message of:

```
File "<ipython-input-53-4c2febb9d238>", line 8
  for i in range(0,s[0])
```

SyntaxError: invalid syntax

Ipython (the interpreter) will generally inform you of where the error in your code comes in (by line), indicate the position with a carat and describe the specific error. While in this situation it is easy to interpret what went wrong based on ipython’s report, it isn’t always this clear cut. In every situation, it is always best to follow good programming practices and leave yourself (or later users) plenty of helpful pointers as to what is happening and why.

### 7.2 Comments

Different programmers have different conventions for using comments in their code. The general motivation is to provide occasional statements summarizing the actions of a block of commands, and to provide explanations, units, or even sources when you’re working with empirical relations, for single lines of code that may be less than clear when first reviewed. Some people choose to provide documentation above blocks of commands, others choose to write their comments below them, but all programmers hold to this general motivation.

**Example 7.2**

The following code could be used to find the exit angle of a ray from an equilateral prism on an optical bench given the entry angle and index of refraction of the prism,

```python
def theta_out(theta_in, n_prism):
    n_air = 1.001
    theta_1 = np.arcsin(n_air*np.sin(theta_in)/n_prism)
    theta_2 = np.pi - ((np.pi/2.-theta_1)+np.pi/3.)
```
The code snippet provided demonstrates a function `theta_out` that calculates the exit angle of a beam of light from a prism based on the entry angle and the index of refraction of the prism. The function takes two parameters: `theta_in` (the incident angle relative to the normal of the entering beam of light) and `n_prism` (the index of refraction of the prism). The output is `theta_fin` (the exit angle relative to the normal of the interface at which the light will leave the prism).

```python
def theta_out(theta_in, n_prism):
    
    # defining required constants
    n_air = 1.001

    # applying Snell's law for the entering beam
    # ALL ANGULAR VALUES ARE IN RADIANS
    theta_1 = np.arcsin(n_air*np.sin(theta_in)/n_prism)

    # applying basic triangle math
    theta_2 = np.pi - ((np.pi/2. - theta_1)+np.pi/3.)
    theta_3 = np.pi/2. - theta_2
```

The documentation for the function provides a clear explanation of its purpose, input, and output, along with the mathematical steps involved in calculating the exit angle. This helps in understanding the code without needing to refer to the code itself.
#applying Snell’s law for the exiting beam
theta_fin = np.arcsin(n_prism*np.sin(theta_3)/n_air)

return, theta_fin

7.3 Debugging Tips and Tricks

As I hope we have demonstrated, debugging is much easier when your code has documentation, is commented, and has informative variable names. However, often we find that errors in programs are distributed between two levels. After writing the initial draft of your program, you will likely spend some time correcting the simple syntax and logical errors that the interpreter can catch. After this initial batch of errors are caught, however, there often remain several more insidious errors that will still stop your function from running properly, but are not as immediately apparent. To deal with that second tier of errors, here is a sample order of approaches you can follow,

- Check your indexing. Remember that Python does not include the last value in a range when you index an array. Additionally, remember to keep track of what the indices used in loops actually correspond to (if you have an image, are you going through rows or columns? If you have a 2d array showing variation across two parameters, which axis corresponds to which parameter?)

- Look for infinity and NaNs showing up in your arrays, they’ll cause you a great deal of trouble. To avoid them, try to follow math (don’t take the log of 0, or divide by 0).

- Look at EVERYTHING. It is important to get a good idea of what your intermediate products in a program look like, and to get used to using imshow to do more than interpret pretty pictures. In a similar vein, it is often useful to report back or save many intermediate values produced in your code for the first run through so that you are better able to check the accuracy of your function as it runs, and have information to look at in case of failure already saved if your function is particularly slow to run and would be a pain to run again.
7.3. DEBUGGING TIPS AND TRICKS

- Create test cases. For any function you write, you should be able to provide some input for which you know the expected output. If you can’t do this, you don’t understand what you’re doing well enough.

- Incorporate checks into your code based on what you know should be true for your function. You should generally do this when you’re writing a program to start with so that you don’t have to approach a malfunction completely blind.
Chapter 8

Reading and Writing Data

Much of the programming you will have to perform in astronomy will require processing the input of raw data files of various types and converting it to info-rich, easily usable output files. We will cover the usage of fits files, a very commonly used file type in astronomy, when we go over images, and for now we shall focus on using text files (.dat, .txt, .log, etc.) and the Python-specific .npz file.

8.1 Reading Data

8.1.1 Reading Data with numpy.loadtxt

Given a file containing some number of columns of text, you can read the values contained in those columns into Python by,

```python
info = np.loadtxt('data.txt')
```

Info will be a two dimensional array where every column corresponds to a single row of information in the file. To gather all the values in one of the files columns you would index info,

```python
coll = info[:, 0]
```

Often, however, you will find that your text files are more difficult to read in due to the inclusion of comments, separation of data points by a specific character, varying datatypes for your columns, or entire introductions that do not correspond to the same pattern as the columns that you actually want to read in. In these cases, the comments, delimiter, dtype and skiprows
keywords (respectively) will be very useful. If you run into other issues, it may be worthwhile to look into the loadtxt documentation.

It is also important to note that in some cases it may be easier for you to use the native open, readlines, and read functions in python. In this case, to read all of the lines of a text file into an array with each line corresponding to a string in a list, you could write,

```python
file = open('data.txt', 'r')
lines = file.readlines()
file.close()
```

where the 'r' input specifies that you are opening the file for reading.

### 8.1.2 Reading .npz files

Given a Python-specific .npz file, you will read in the data values as,

```python
info = np.load('data.npz')
```

Info will in this case be a dictionary with keys designated during the creation of the .npz file. To see these keys, you can use the object attribute, info.itemkeys, and index the dictionary with specific keys as,

```python
value = info['value_key']
```

### 8.2 Writing Data

#### 8.2.1 Writing to a .txt (or text of any kind) file

To generate a simple text file, you can use the open, write and close commands. In this case, you would use the 'w' option to open the file for writing (or the 'a' option to append) and write,

```python
file = open('filename.txt', 'w')
file.write('some values or something')
file.close()
```

Unless a `\n` is appended to the end of a string within your write command, all printed statements will be added to the same line (even if you use multiple write commands). This is often more useful when you are simply generating a computer, rather than human, readable document.
To generate a file similar to those described in the first section, you can use the command `np.savetxt` as,

```python
np.savetxt('filename.txt', array_of_values)
```

where `array_of_values` will be saved in the text file in the same format as the info array described in the first section.

### 8.2.2 Writing to a .npz file

To save variables from a program into a .npz file you will provide the command,

```python
np.savez('filename.npz', key1 = value1, key2 = value2)
```

where your keys would correspond to `key1` and `key2` (but you can have more or less) and your variables would correspond to `value1` and `value2` (but you can have more or less, and they can be of any Python class type).

### 8.3 Tutorial

Navigate to the directory `/idl/week3/datafiles/` and take a look at the contents. We’ll see a bunch of files named as `data1.dat` `data2.dat` `data3.dat` ...

`data15.dat`

In practice, never label your data with an unhelpful name like “data”.

Our task for this tutorial will be to read through and process each of these files in an efficient manner (i.e. automatically). Let’s take a look at the first file just so we know what kind of data we’re dealing with. From command line:

```
/idl/week3/datafiles% emacs -nw data1.dat
```

Inside this .dat file we see one column of a bunch of numbers. If we look through the numbers we could get a feel for the range and values of these numbers. We can do this for all of the files and write down values for what these numbers are, but we can do better than that.
Let's play around with a file from command line. We use the readcol command to read one in:

```python
In [1]: readcol, 'data1.dat', dat
```

We can see how many numbers are in this data file:

```python
In [1]: print n_elements(dat)
```

As well as some statistics for the numbers:

```python
In [1]: print avg(dat)
In [1]: print stddev(dat)
In [1]: print median(dat)
In [1]: print mode(dat)
```

### 8.4 Problems

- ⭐⭐ Read into Python the file numbers.dat. In it are one hundred 50-digit numbers. Take the 5th to 10th digit of each number and sum all of these numbers.
Chapter 9

Plotting

One of the basic, but powerful ways to display data in science is through plotting. However, a plot is only useful if done correctly. In this chapter we’ll not only be trying to convey the code on how to display data but also the necessary information needed in order to make the plot useful.

For this chapter, we will be using the matplotlib.pyplot libraries, so we will begin each of our sessions and our documents with

```python
import matplotlib.pyplot as plot
```

### 9.1 Basic Plots

Let’s say that we wanted to plot the equation

$$y = 5x + 4$$  \hspace{1cm} (9.1)

The way that plotting works in Python is similar to how you did it when you were in grade school. Take a table of x values, plug them into the equation, fill in the y values, like in Table 9.1. Then on graph paper, match x value to y value and connect the dots.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5 * 1 + 4 = 9</td>
</tr>
<tr>
<td>2</td>
<td>5 * 2 + 4 = 14</td>
</tr>
<tr>
<td>3</td>
<td>5 * 3 + 4 = 19</td>
</tr>
<tr>
<td>50</td>
<td>5 * 50 + 4 = 254</td>
</tr>
</tbody>
</table>
Python’s `plot` procedure takes two inputs, our column of x values and our column of y values, each in an array. We then call plot with the following syntax:

```python
plot.plot(x, y)
```

This may seem like a redundant name, but all it means is that from the matplotlib.pyplot package we’ll be using the plot function which takes two arrays and plots them by connecting each successive point.

Python then matches the first element in the x values array to the first element in the y values array and plots it, then takes the second element with the second element and so on. Clearly, the x array and the y array must be the same length. Now, how do we generate our arrays for any given function we might want to plot? Well, we use our elementary school method of typing in an array of x values, computing the y values and typing those into an array, but we can do better than that. We basically want to apply our function to every element of some arbitrary array of x values. But remember that when we do operations on an array (say, 5 * x array), it does the operation to every element of the array, so if we simply apply our function to the entire array of x values rather than to each individual x value, we’ll get the array of y values out:

```python
In [1]: y_array = x_array * 5. + 4.
In [1]: print x_array
[1  2  3  4  50]
In [1]: print y_array
[ 9 14 19 24 254]
```

We’re almost there, now all we need is an array of x values. In general, this will depend on what part of the function we want to look at. We want an array with enough points to give us an accurate sampling of the function while giving us the range of the function that we want to see. Regardless of the specifics of what we want to see of our function, our starting point for our x array will almost always be a findgen array (in the problems, we’ll see a few examples where, say, a fittarr might be a better choice). Remember what a `plot.arange` does:

```python
In [1]: x_array = numpy.arange(5)
In [1]: print x_array
[0 1 2 3 4]
```

So let’s get to it already and make a plot:

```python
In [1]: x_array = numpy.arange(5)
In [1]: y_array = x_array * 5. + 4.
```
9.1. BASIC PLOTS

Figure 9.1: A plot of the function $5 \times x + 4$

In [1]: plot.plot(x_array, y_array)
We’ll notice after the plot.plot command that python doesn’t do anything.
We need one final command to make it work:
In [1]: plot.show()
which tells python to open up a plot window that displays your plot, which
should look like Figure 9.1.

Let’s try some more functions:
In [1]: x_array = numpy.arange(5)
In [1]: y_array = x_array**2. + 2.
In [1]: plot.plot(x_array, y_array)

which produces Figure 9.2. Notice how the line looks rather chunky rather
than being a smooth curve. Unlike our middle-school selves who would do a
good job of interpolating a nice curve to connect the dots, all Python can do
is use a straight-edge to draw lines between the points that we give it. How
do we get a nicer plot? Clearly, we need to add more points to our graph,
by adding more points to our arange array.
In [1]: x_array = numpy.arange(50)
In [1]: y_array = x_array**2. + 2.
In [1]: plot.plot(x_array, y_array)
Figure 9.2: A plot of the function $x^2 + 2$
Figure 9.3: Another plot of the function $x^2 + 2$
producing Figure 9.3. This looks much nicer, but notice how the plot now has a much bigger range. If we zoom into the same range as Figure 9.2 using the plot.xlim and plot.ylim:

```python
In [1]: plot(x_array, y_array)
In [1]: plot.xlim([0, 4])
In [1]: plot.ylim([0, 16])
```

we generate Figure 9.3. Notice that this has the exact same range as Figure 9.4. We see that we haven’t really fixed our problem; our plot is still as blocky as it was before. In order to make our plot look a little nicer, we need to increase our sampling, or the number of points in the range that we want to look at. Let’s make this explicit. Right now in the range of our function, we have 5 points. Let’s say we wanted to double the number of points in our range from 0 to 4. We can do this the hard way:

```python
In [1]: x_array = [0.,.5,1.,1.5,2.,2.5,3.,3.5,4.]
```

but, as usual, this is not the way to go. An easier way to do this is to notice that our array with twice the sampling is basically a findgen array up to 8 with every element divided by two!

```python
In [1]: x_array = numpy.arange(9)
In [1]: print x_array
0.00000 1.00000 2.00000 3.00000 4.00000 5.00000 6.00000 7.00000
```

Figure 9.4: Figure 9.3 zoomed into the range 0-4 on the x axis and 0-16 on the y axis
9.1. BASIC PLOTS

Figure 9.5: The parabola with proper sampling

8.00000
In [1]: x_array = x_array / 2.
In [1]: print x_array
0.00000 0.500000 1.00000 1.50000 2.00000 2.500000 3.00000 3.50000 4.00000

We can do this with an arbitrary amount of points, by applying the transformation on the axis:

x_array = numpy.arange(desired number of points)
x_array = x_array * (max value wanted) / (desired number of points)

Note that we don’t need to change the line where we apply the formula to get the y axis because nothing’s changed. Overall, here’s what it looks like:

x_array = numpy.arange(50)
x_array = x_array / 10.
y_array = x_array**2. + 2.
plot.plot(x_array, y_array)
plot.show()

Which generates 9.5.
9.2 Advanced Plotting

9.3 Important Functions/Syntax

9.4 Tutorial

Linear Least Squares part 2 Todo: finish

9.5 Problems

- * In this problem, we’ll generate some simple figures and then save them, so you should write all of your commands in a single procedure for ease of debugging
  - First, notice that there are four different plots in a single figure. Use
    \[ \texttt{!p.multi = [0,2,2,0,1]} \]
    to set up our plot window to generate all four plots in the same window
  - Now, try and generate the first plot
Chapter 10
Images

This chapter is aimed at dealing with images in astronomy. In astronomy, when we take images with a telescope, we’re taking light from an area the size of our lens and we focus it onto a small CCD (charged-coupling device) plate. The CCDs in telescopes work much like a CCD in a digital camera. We have an array of little buckets to catch photons in, we expose the array to some light, and count the number of photons that we catch in each bucket. The more photons we catch in any particular bucket, the brighter our source is in that corresponding location in the sky. Each of our buckets is called a pixel and corresponds to a certain amount of solid angle in the sky. The CCD can then send the number of counts from each bucket to our computer where we can display it. Our task for this chapter is to deal with such images in Python.

10.1 FITS files: Headers and Images

The ubiquitous format for images is one that you’re not likely to see outside of astronomy: a FITS (Flexible Image Transport System) image. They end in .fts or .fits. A FITS file is useful not only for its image content, but also because it comes packaged with a header, which is a text file that gives us information about the image such as the date and time that it was taken, some calibration information, the pixel scale, or the coordinates that were observed in this image.

The Python package used to read in images in Python is called PyFITS.

TODO: Finish
10.2 Important Functions/Syntax

10.3 Tutorial

Centroid

10.4 Problems

- Read in the FITS image
Chapter 11

\LaTeX

11.1 Why \LaTeX?

In science, one of the most important skills is to be able to communicate clearly and effectively. Part of this is having a well-formatted and aesthetically pleasing document. \LaTeX (pronounced Lay-Tech or Lah-Tech) is a programming language specifically suited for this endeavor. It is not a traditional programming language in the sense that you create functions and programs, but in that it has specific commands and codes that create certain outputs. The \LaTeX style of formatting is seen in many textbooks (including this one). While the fact that the tendency of \LaTeX is to put figures on different pages than the reference to the figure, we’ll soon see that this is actually an advantage when writing books and scientific papers. We’ll get into the advantages of \LaTeX later on in the chapter.

11.2 \LaTeX Headers and Packages

In order to start a \LaTeX document, we open a new text file in our favorite text editor and make sure that it ends in “.tex”. Our final file, usually a .pdf will be compiled from the .tex file. In other words, the \LaTeX compiler will read through our file and parse all of our commands and translate it to a .pdf output. Let’s look at what the .tex file looks like

The first line of any \LaTeX document is the declaration of the document type:
\documentclass{article}
CHAPTER 11. \texttt{\LaTeX}

Article is the ubiquitous format for most documents. There are document classes that are suited to specific types of documents such as the book documentclass which is used for this book. Additionally, many journals in physics will release their own document classes (such as Rev\TeX) which include all of the formatting specific to their individual journals.

Commands in \LaTeX, like our commands in UNIX and Python have three components: commands, arguments and options. In the above example, the command is “documentclass”. In all \LaTeX commands, commands will begin in the backslash. The argument is the type of class the document is, in this case “article” and is enclosed in curly brackets. The above command doesn’t have any options, but we can give it some enclosed in square brackets:
\begin{verbatim}
\documentclass[12pt]{article}
\end{verbatim}

There are many other options that you can add to this command, separating them by commas, but for most purposes, this is the only option you’ll realistically need.

The next part of the document is where we import packages. These packages, like packages in Python, teach the \LaTeX compiler how to do certain functions. For example, we will use a package called \texttt{geometry} which allows us to set the margin size:
\begin{verbatim}
\usepackage[margin=1in]{geometry}
\end{verbatim}

This will make our margins one inch. We’ll get into more packages to use when we need them.

Anything beyond that is part of the body of the text. This part of the document where we establish the properties of the formatting is called the \texttt{header}. In order to mark that we are beginning to write text that will appear on our compiled document, we use the most used command in \LaTeX, the \texttt{begin} command:
\begin{verbatim}
\begin{document}
\end{verbatim}

We put all of our text after this. For every begin there must be an end:
\begin{verbatim}
\end{document}
\end{verbatim}

All put together our document will look like
\begin{verbatim}
\documentclass{article}
%This is how you put comments, you precede them with a % sign
\usepackage[margin=1in]{geometry} % This allows us to manipulate the geometry
\begin{document}
\end{verbatim}
Though my soul may set in darkness
It will rise in perfect light
I have loved the stars too fondly
To be fearful of the night

\end{document}

And that's it! Already, this can be compiled by pressing the compile button on your \LaTeX editor or by typing into your UNIX command line:

UNIX> pdflatex [document name here].tex

11.3 Basic Math and Formatting

11.4 Good Writing Practice

11.5 Tables

11.6 Figures

11.7 Some Useful Templates
.1 Using Windows to SSH and SCP

You will need the following programs: PuTTy WinSCP X-11 forwarder