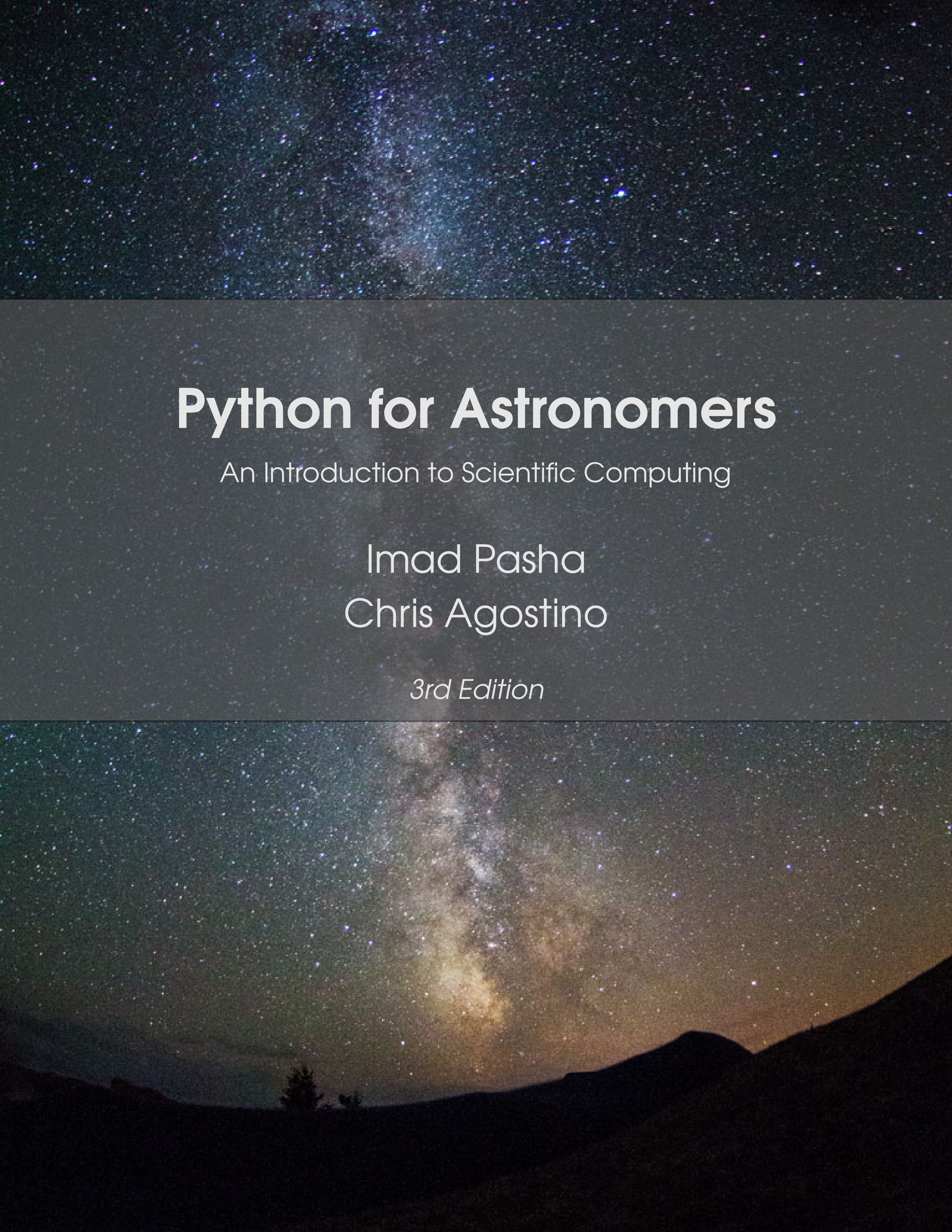


Python for Astronomers

An Introduction to Scientific Computing

Imad Pasha
Chris Agostino

3rd Edition

DEPARTMENT OF ASTRONOMY, YALE UNIVERSITY

Original template by Mathias Legrand
Image Cover Credit: Imad Pasha (all rights reserved)
Third Edition, June 2020

Contents

Preface

Over the course of the last 50 years, programming has become increasingly essential to the research of an astrophysicist and/or astronomer. In the early days, when Hubble discovered the existence of other galaxies, observational astronomy was performed by eye, with drawings, handwritten notes, and manual calculations and plots. Theory was a pencil-and-paper endeavor. Fast-forward through the invention of glass plate exposures to the modern era of CCD (charge-coupled device) telescopes, full-sky surveys, and supercomputer simulations, and the importance of a solid foundation in computing becomes clear.

In fact, the need for a strong background in computer programming has drifted younger in the past decade or two — now, undergraduates are expected to be proficient in at least one language when conducting research and applying to graduate schools. Between 2014 and 2018, I taught the entry-level Python programming course in the UC Berkeley Astronomy Department. This is the text that I wrote for that course — but which I am continuing to update and revise.

What this Book Is

This text is designed to be an introduction to the Python programming language — which is now used nearly ubiquitously in astronomy — with applications to the types of tasks an undergraduate (or beginning graduate student) might have to tackle. It is not, in any way, meant to be comprehensive; my focus is on bringing you up to speed as efficiently and quickly as possible. This text might have useful information for those with a working knowledge of Python outside of a research setting, but is primarily designed for those with no prior programming experience. By the end of this text, I hope to have shared enough to make you feel comfortable taking the first steps into research-type problems (of which this book will contain several examples), whether in an astronomy course, a research internship, or on-campus research.

Chapter 1 focuses on UNIX/Linux, and can be skipped initially if you want to jump into Python. More advanced topics can be found near the end of the book. The course accompanying this text is generally taught in 1 semester, and interactive tutorials for this text are available online.

1. Essential Unix Skills

1.1 What is UNIX, and why is it Important?

UNIX is an ***operating system*** which was developed at Bell Labs research center and was designed to be a multi-user system which could multitask more efficiently than previous systems. Many modern operating systems such as Mac OSX, Ubuntu, GNU, and many others are considered to be UNIX-like in the way they are designed–specifically in terms of their filesystems. To be clear, Unix is an operating system that handles files (and where they are stored within folders/***directories***) the same way your own computer does. The principle user-end differences between UNIX and other operating systems are the interface by which one interacts with the file system, and the way filesystems can be stored on servers that are accessible from any computer in the network (we'll talk more about this later).

1.2 The Interface

It follows that an understanding of UNIX can be extremely useful as many of its basic tenets can be applied to a variety of scientific systems. Alternatively, most current operating systems on personal computers offer what is called a ***graphical user interface (GUI)***.

Definition 1.2.1 A **GUI (Graphical User Interface)** is the type of system most people are accustomed to, in which the primary means of interacting with the file system is via a mouse which can click to open up windows of different folders, etc.

While GUI systems are usually more intuitive by nature, they are also inefficient (for example, to move a file from one folder to another one has to open two windows separately, usually by clicking through multiple other folders). In comparison, command line interfaces have a steeper learning curve, because they require knowledge of ***syntax***.

Definition 1.2.2 **Syntax** refers to the phrases and commands that can be interpreted properly by a computer.

However, the advantage of learning this syntax is that the command line is an extremely efficient way of navigating an operating system. As a counterpart to the previous example, a single phrase in the command line (“mv filename newlocation”) can accomplish what took many clicks and drags in a GUI.

R Note: Because these GUI’s on personal computers usually operate over UNIX, one can usually find ways to access a command line interface for these computers as well, it just isn’t the primary interface by which most people interact with the operating system.

Perhaps most importantly, the majority of supercomputers and telescopes are operated by systems which utilize Linux operating systems which makes it especially useful for astronomers to learn. For example, one can often only access these systems through the use of the terminal’s “secure shell” service, or more simply, ***SSH***, which provides a remote, secure login. In utilizing the command line, or ***terminal*** as it will be referred to from here on, one can accomplish many tasks by simply typing a few commands rather than having to make several mouse clicks.

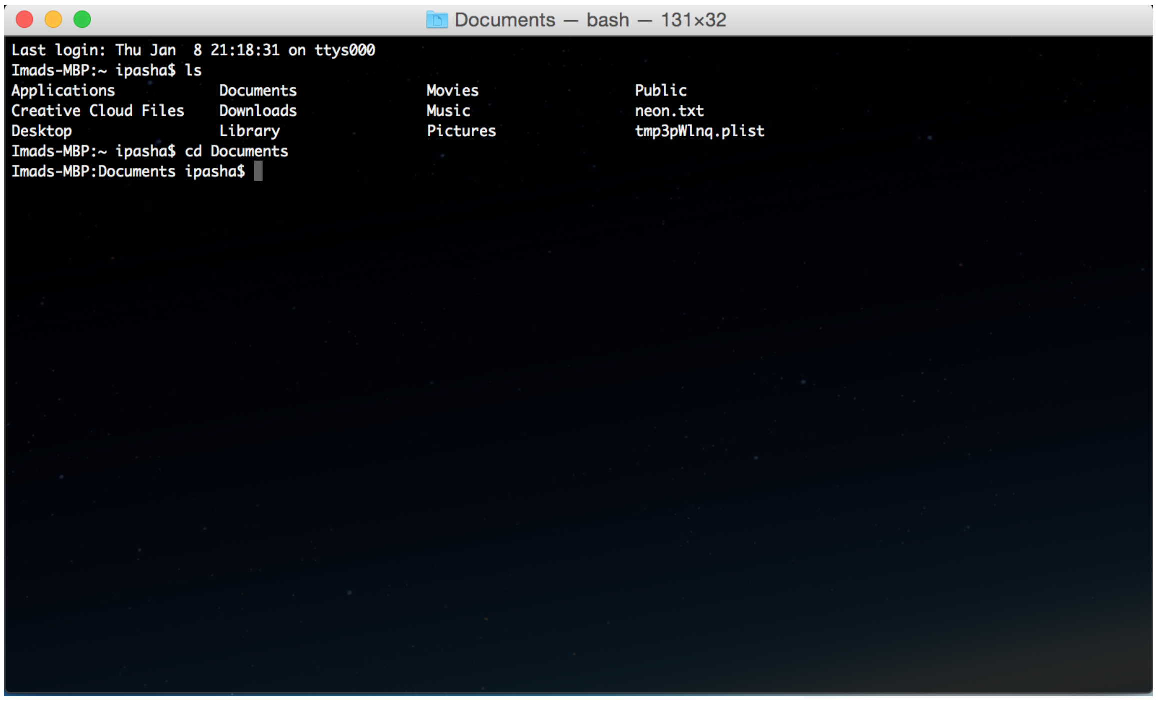
Definition 1.2.3 A **terminal** is a form of command-line interface, that is, a program by which the user enters commands in a certain syntax that the computer then executes.

Some examples which will be explicitly outlined later include removing files, creating folders, opening programs, and searching through droves of files for a specific keyword.

1.3 Using a Terminal

Most systems will have keyboard shortcuts to expedite and simplify the opening of a terminal application. In Ubuntu, one can press Ctrl+Alt+t to open a new terminal window. In Mac OSX, perhaps the simplest way to open the terminal application is to press Command + Spacebar then to type terminal and press enter. The new window should open in your home directory, which is the directory of your own user account which holds your documents, downloads, music, and other personal files. Not all operating systems have keyboard shortcuts for this but if one wishes, one could customize his/her own keyboard shortcuts to allow for this quicker functionality.

Once the terminal has been opened, there are really only two areas to understand. The ***prompt*** is a line of letters and symbols that appears on the left of the terminal window. What this prompt actually reads is different for different systems, and in fact can be customized to say whatever you like. Many systems will be set up so that the prompt indicates to some degree the current ***path***. Path refers to the description of where you are in the filesystem, beginning with it’s most basic, or root directory. For example, if you were in your documents folder, the path to your location might be /root/home/users/your_name/documents. Different systems also have different names for their root directory. In some cases your prompt may be the full path, but it is often a shortened version which only indicates the name of the current directory; such a prompt might look like this: “systemuserid:documents%” (Prompts almost always end in a % or $ symbol). Ultimately the prompt doesn’t affect what you can or can’t type; at most it can be a handy way of seeing where you are in a file system. The other area of the terminal is the actual command line, where you type the commands to be interpreted by the computer. We will cover in depth all of the commands you need to know to navigate a UNIX system. (See fig. 1.1).



```
Last login: Thu Jan  8 21:18:31 on ttys000
Imads-MBP:~ ipasha$ ls
Applications            Documents               Movies                  Public
Creative Cloud Files    Downloads               Music                   neon.txt
Desktop                 Library                 Pictures                tmp3pWlnq.plist
Imads-MBP:~ ipasha$ cd Documents
Imads-MBP:Documents ipasha$
```

Figure 1.1: A typical terminal, with a prompt on the left hand side. An example command has been sent through, which displays the contents of a folder.

1.4 UNIX Commands

UNIX terminals have a vast number of available commands which one can use but many of them are outside the scope of this course. For the majority of this course, you will need only a few simple commands which will be outlined and explored in this chapter. A more complete list of commands with shorter explanations is available in the appendix, and on the website as the “UNIX Guide.”

We begin with the commands by which one actually navigates from folder to folder within a UNIX system. In the course of this textbook, we will frequently want to indicate commands to be typed into the terminal. Our format for doing so will be to represent the prompt with “»” and to usually indent commands on separate lines: for example, to exit a terminal, type

```
»exit
```

and the shell will close. Additionally, in this section of the text it will be useful to define a “typical” series of nested directories, so that as we practice navigation, we can use self consistent examples. For now lets use this typical tree:

```
root/physics/user/sally/documents/homework/python/week1/
```

1.4.1 Changing Directories

There is a single command by which one can navigate the entire UNIX directory tree of any system, and as there are several subtleties to it, we will discuss it in some detail. The command in question is “cd.” The syntax “cd” is interpreted by the computer to mean “change directories.” Clearly though, with just this command, it would be impossible for the computer to know where to change directories to. Because of this, the command cd takes what is called an ***argument***. An argument is a part of the command necessary for it to function, but that is variable- the user can specify different values for the argument within a certain set of possibilities. In this example, the "cd" command takes as an argument a path location, for example in the command,

```
» cd /root/physics/sally
```

/root/physics/sally serves as the argument to "cd", it tells cd where to actually change directories to. The majority of commands in UNIX have arguments, although there are a few exceptions.

So the question is, how can we efficiently use the cd command to navigate between directories in UNIX? Luckily, the cd command has several built-in shortcuts that make navigating easier. However, we would like to point out that these shortcuts only work in certain situations, which will be described below. On the other hand, there is one surefire argument for cd which will always work; unfortunately it is the most cumbersome. Below, we describe the various ways to phrase arguments for cd, the first being the surefire method.

1. **The full path** : From any directory in a UNIX system, typing cd followed by a full path, starting with the root directory, will take you immediately to the specified location. This is possible because a full path is unique, and thus the computer knows exactly where you mean to go.
2. **A nested directory**: For example, if you are in the directory "/root/physics/sally/" and want to cd into the homework directory, you can simply type

   ```
   »cd homework
   ```

   This may seem confusing at first, because there is no "/" before homework. Essentially, the computer is interpreting your lack of a "/" to mean that the directory you are looking to cd into is within the one you are currently in. (It will complain if you give a directory as an argument that is not in the current one, if you use this syntax).
3. **A secondary nested directory**: If you want to cd from a current location to two directories deeper in the nesting system, you can start with the syntax from (2), but continue it into a longer path, for example, if you were in "root/physics/sally/" and wanted to get into not just "homework," but all the way into "python" you would

   ```
   »cd homework/python
   ```

   and further nested directories can be strung onto the end, if desired.
4. **cd (space)** : defaults you to your username's home directory. This will be different on different systems, but it is typically where your personal documents, downloads, pictures, etc., folders are stored.
5. **cd (space) .** : cd followed by a space and a period takes you to the current directory. In fact, a single period is always a shortcut for "current directory," in lots of different commands. Using cd with just a period (as above) is pretty useless (it takes you nowhere). On the other hand:

   ```
   »cd ..
   ```

   is useful. This command will take you out one directory; i.e., if you were in the homework directory of the sample tree, "cd .." would take you to the "documents" directory. This command can be strung together as well:

   ```
   »cd ../..
   ```

   brings you out two directories, and so forth. The single dot is most useful in other functions, for example, when copying or moving files from remote directories to the directory you are in.

1.4.2 Viewing Files and Directories

You may notice that changing into a directory may change your prompt to reflect it, and/or typing "pwd" will "print working directory" and show you where you are. (As a side note, we type "print" though we aren't printing things to a printer, because we are in essence printing the values from the memory to the console screen). But none of this actually tells you what is in the directory in question, unlike a GUI through which you actually see the files in a directory when you view it. No one's memory is perfect, so there is of course a command for viewing the contents of a directory. Typing:

```
»ls
```

into the terminal will print a list of the files and folders in the current directory in which you are operating. 'ls' has many useful flags for various situations, which are listed in the appendix under the UNIX guide. Additionally, typing

```
»man(ls)
```

will bring up the manual for it (or any command you choose), right in the terminal. Note that 'ls' is one of those few commands that does not take an explicit argument, though it does have optional ones.

1.4.3 Making Directories

Now that we know how to view the contents in our directories, it becomes important for us to know how to create and delete files and directories as well. In order to create a directory, use the command,

```
»mkdir desired_name
```

which you may notice is a shortened version of the phrase "make directory." For instance, if one wants to create a directory in which to store their specific information regarding this course, one would type into the terminal

```
»mkdir python_decal
```

If we use 'ls' we will see that this new directory is included in the contents listed. It is imperative to note that you should avoid using spaces when using the 'mkdir' command as placing a space between two or more words will just end up creating directories named after each individual word. If you really must have a space in your directory, you can type a backslash before the desired space as this tells UNIX to ignore the space. On shared-network file systems, having an organized system of directories that make logical sense is very helpful, both for keeping yourself organized and for allowing you to direct others to specific files and folders more easily.

1.4.4 Deleting Files and Directories

Now that we know how to create directories, move between them, and look at the files inside, the next step is to learn how to delete things.

R UNIX is not like a Windows or Mac where files are sent to a trash bin. When you hit delete, things are gone forever.

Removing files is a relatively easy task in UNIX. If you are in the directory where the file to be deleted is stored, simply type

```
»rm filename
```

to delete it. Removing directories requires the use of a ***flag***, or ***option***. A flag/option is a modifier added after a command, before the argument, that changes exactly how the command is carried out. For example, typing ’ls -a’ will list “all” files, whereas ’ls’ alone usually ignores hidden files and directories. In this case, we need to make use of the “recursive” option:

```
»rm -r directory_name
```

which will go into a directory, delete the files within, and then delete the directory itself. The example here illustrates the syntax for using flags/options in general (with a dash preceding the flag).
There is a secondary way of deleting directories, which may be easier to remember:

```
»rmdir directory_name
```

which will also delete the directory in question, but not if it contains files.

1.4.5 Moving/Copying Files and Directories

The last major skill needed for operating in UNIX file systems is moving and copying files and directories from one place to another. Moving is done using the “move” command (who would’ve thought?):

```
»mv filename new_location
```

(This assumes you are in the directory with the file to be moved. Depending on where you are moving the file to, the new_location could be as simple as “..” or as complex as a full pathname to another directory tree).
The command ’mv’ also gives you the option of changing the name of a file as you move it, for example:

```
»mv file_name new_location/new_name
```

would move “file_name” to “new_location,” changing its name to “new_name” along the way. Interestingly, because of this functionality, ’mv’ serves as the “rename” command as well. To rename a file, “move” it to a new name without specifying a new location to send it. If you want to copy a file instead of moving it, use:

```
»cp filename new_location
```

which will create a copy and put it in “new_location”. The command ’cp’ also has the ability to rename files in transit, by the same syntax as ’mv.’

1.4.6 The Wildcard

One extremely useful thing to know about UNIX is the ability to use wild cards. Denoted by a “*” symbol, wildcards can stand for any character, or any number of characters. The strategic use of wildcards can save you a lot of time when working with large numbers of files. A few examples should make clear how wildcards are used:

1. **Deleting many files**: Say for example you wanted to delete all files in a certain directory that were of the type .doc (or docx for all you millenials). If you entered

```
»rm *.doc
```

The wildcard would feed 'rm' every file with any combination of characters that ended in .doc for deletion. In a similar vein, if you have a group of research files that all started with "simulation_run1" (where an example filename might be simulation_run10004.dat, simulation_run10005.dat, etc)

```
» rm simulation_run*
```

would delete all of those files, as 'rm' doesn't care what comes after the "n" in run anymore.

2. **Copying files**: This is somewhat of a trivial expansion, but it is useful to note that more often than not you are going to be copying and moving large numbers of files rather than deleting them (archiving data for later is safer than losing it). It becomes clear now why many research processes that output many files have a very regular system for naming: it allows for the easy extraction of subsets or all files within UNIX systems. Wildcards also work within names, for example:

```
»cp simulation*.dat newlocation
```

would copy all files starting with "simulation" and ending with ".dat" to a new location. This can be handy if your software also outputs files with the same prefix but different file endings, and you only want the .dat files.

1.5 SSH and Servers

An extremely important aspect of working with the command line is ssh-ing into servers to work. A ***server*** is a computer or system of computers that store files and contain programs that are accessed and run remotely. Almost any computer can be converted into a server, though generally speaking servers are set up on computers with a lot of memory and free space. Astronomers use servers frequently because they allow for the storage of large (we are talking multiple terabytes) datasets. Additionally, it allows us to log in and work on our research from any computer with an internet connection, without needing all the data and programs installed on our personal machines. Finally, with multiple users on the same server, it becomes easy to share data, code, and any other file with collaborators, instead of having to email or otherwise transfer things to their computers.

1.5.1 Logging Into a Server

Usually, the only way to interact with a server is to log in via ***SSH***. SSH is a terminal command standing for "secure shell host". When you run a command like

```
$ssh username@servername.address
```

in in the terminal, your computer reaches out to the server and establishes a connection (assuming you have an account on the server). To give a concrete example, say you have an account under the name "sjohnson" on a server called "vega" on UC Berkeley's astro network (it is typical for user accounts on servers to be first initial-last name, but it is up to the admin of the server how this works). You would type

```
$ssh sjohnson@vega.astro.berkeley.edu
```

to log in to the server. The first time you try to SSH to a new server, you will be asked whether to trust the RSA key and add it to your trusted list, (just hit "y" and enter). The server will then ask you for a password. The admin for the server will have made one for you when they created your account; once you log in you can generally chance this to something of your choosing using something like the "passwd" command (but this varies by system). Note that when you are typing in your passwords, nothing will appear on the screen- that's normal, just type the password and hit enter.

Now that you are in the server, everything works just like you are in a terminal on your own computer. You can ls, cd, and otherwise work with the files and programs installed on the computer you are ssh'd into. One extra step that's worth mentioning is that if you want to open programs with display windows (for example, ds9 which we will cover later), you will need to use the "-X" flag; that is,

```
$ssh -X sjohnson@vega.astro.berkeley.edu
```

This will allow the windows to open on your computer (other common flags include -L and -Y). For this to work, you will need something called X11 forwarding. On a Mac, this involves installing something called "XQuartz" (easily googled), and on a PC it involves installing something called "Xming" and "Putty" which have X11 options. There is a guide to getting SSH working included with this bundle which covers this.

1.5.2 Copying files to a server using SCP

Often we have the need to move files between the server we are working on and our own personal computers (or between two servers). The default command for this is "scp," which stands for "secure copy." To move a file called "test.txt" from a certain computer to, for example, a user directory on a remote server, the syntax is

```
scp test.txt username@server:/home/user/
```

assuming, of course, you are currently in the directory with the file. To give a concrete example using the same name as above,

```
scp file.txt sjohnson@vega.astro.berkeley.edu:/home/users/sjohnson/
```

would move the file to that location on the server after prompting for sjohnson's password.

1.5.3 Pulling a file from a server

Pulling a file from a remote server uses the same structure as the section above, but switches the two arguments. For example, to pull the file above back to our own computer, we would use

```
scp sjohnson@vega.astro.berkeley.edu:/home/users/sjohnson/file.txt
/Users/samjohnson/files/
```

where we specify any directory we want on the current computer.

These are the primary ways to copy files (remember, to copy multiple files we could just tar them into one file and move that). If you are ON a server, and trying to transfer from there to a specific computer, it can be slightly trickier and involves looking up the hostname and IP address of the computer in question, making the process less efficient. There are other ways, including ftp, that try to resolve this issue.

1.6 Setting up Aliases and Tab Complete

A lot of the commands we've covered so far are short and succinct- the point of the command line is to increase efficiency. Things like 'ls' take very little time to type and can easily be used to navigate a file system. On the other hand, certain commands (particularly ones with long arguments, like the SSH commands above) are a pain to write out every time. There are two main ways of decreasing the amount of time you spend typing unnecessary information: ***tab complete*** and ***aliases***.

1.6.1 Tab Complete

Tab complete is a feature of the terminal that allows you to quickly finish commands or filenames as you type them assuming they are unique. For example, let's say I have three files in a directory, "testrun1234876545635624.dat", "testrun49232450238472034.dat", and "testrun95432859234502598.dat". It would be extremely annoying to type these all out in something like a copy command. Notice, however, that after the word testrun, all three have a different character, which allows them to be differentiated by only the word testrun and the first number (i.e., if I asked you to give me "testrun9" there is only one you could choose). To quickly copy testrun9... we would simply

```
» cp testrun9<tab>
```

and when we pressed tab, it would automatically complete the rest of the filename, letting us move on to typing in the new location. In fact, tab complete works on typing in locations as well- if you are typing in a long path name, you can tab complete each directory name as you type it, as soon as it's the only one with those letters/numbers in its name. Tab complete can also be useful when you haven't yet reached the unique part of a name- use it at any point while typing to see what options you have (everything starting with what you have already typed will pop up on screen, to remind you).

1.6.2 Aliasing

Certain commands cannot be tab completed. For example, if I start typing "ssh sjohnson..." I can't tab complete because Unix has no clue where I'm going with this command. But let's say there's a server we log into all the time- it would be very annoying to have to type the full ssh command every time. This is where aliases come in. An alias is a command you make up, stored in a special file on your computer (read by your terminal) that allows you to make your own shortcuts. The file that stores your aliases depends on whether your terminal is a bash, csh, tcsh, etc., but we will use bash here for reference (the format for the other terminal types is slightly different but easy to look up). Mac computers default to bash.

In your home directory (the one you are taken to by using 'cd' with no arguments), there is a file called .bash_profile (it won't show up with 'ls' unless you use 'ls -a'). (This is the default for Macs and linux, but cshell is another version- if you are using that, use .bashrc which has slightly different syntax but a similar setup). If you open it using your preferred text editor (see the included "vim guide"), you can add aliases (among other things) to be read by your terminal.

To add an alias, simply leave a line

```
alias 'mycommand'='realcommand'
```

For example, If we wanted to alias our long ssh call, we could do something like

```
alias 'pepper'='ssh -X sjohnson@vega.astro.berkeley.edu'
```

Once the file is saved, open a new terminal, or source your bash profile by typing

```
source .bash_profile
```

in the terminal and you will find that typing in "vega" runs the ssh command and asks you for your password to the server. This is also very useful for directory shortcuts; for example if you have a research directory buried in your filesystem somewhere, you could set an alias 'research'='cd researchfullpath' to make it easier to get to your research directory.

2. Basic Python

Introduction

Programming is the type of thing whose uses and applicabilities seem extremely straightforward once you know how to do it, and extremely nebulous and intimidating before that point. To delve into the intricacies of what programming *is* and *can be* is the job of a computer science professor (read, above my pay grade). But as far as astronomers and astrophysicists are concerned, the use of programming languages (and everything that comes with them) basically amounts to glorified calculator use. At the end of the day, we have some numbers, and we want to do things to those numbers — just like how your parents probably used a pocket calculator to sum up their taxes line by line every year.

Definition 2.0.1 **Python** is a programming language (and yes, it's named for the sketch troupe Monty Python). It is an interpreted high-level programming language for general-purpose programming, and one of the most common languages used in astronomy.

Different programming languages you may have heard of (like C, C++, Java, and Fortran) operate much like human languages in the real world — they are alternative methods of constructing statements with a certain meaning. Just like normal languages, phrasing certain things is easier in some languages than others (e.g., German has many compound words for feelings that require a whole sentence to describe in English). This textbook is concerned with Python in particular. This is because Python has become the language of choice for astronomers and astrophysicists working with data analysis and visualization. Theorists who run large scale simulations of the universe require other languages to be efficient (usually C++ or Fortran), which we will discuss a bit later, but even they now use Python as the primary means of analyzing and visualizing the results of their simulations.

Why Python?

Python has taken on the position it has for a few reasons. Right off the bat, it's easy to learn, and easy to use. In programming-speak, that means it's "high-level" — closer to human speech than computer-bit language. In fact, of all the high-level programming languages, Python is one of the easiest to pick up; this is due in part to how (relatively) new it is. Much of the archaic annoyances of older languages have been removed. Python is also open-source, which means it is free, and there is a large community of users helping to update and help each other use it, as opposed to proprietary languages (like IDL) which require you to buy a license.

A Glorified Calculator

As I mentioned above, a program, in its purest form, is something that neatly packages up a series of calculations (that at their core are simple addition, subtraction, multiplication, exponentiation, etc.) that can then be used to quickly evaluate those calculations on tens to hundreds to millions of values (data). It is this *scalability* — the ability to run a ton of data quickly and effectively through math formulas that would take years to do by hand — that makes programming so powerful. The easiest way to approach programming, mentally, is to remember that you have some data, and just want the computer to do the heavy lifting on the repetitive math you don't want to perform for each data point.

2.1 Hello, World!

It's a cute, and not altogether unhelpful tradition to begin any instructional text with a guide to showing the canonical phrase "Hello World" on the screen. In Python, this is dead easy. From the terminal (instructions for installing Python can be found in the Appendix), simply type

```
ipython
```

to begin an ipython session, and you should see your prompt change to

```
[IN]:
```

with a line number. You are now in ipython.

R When you install a distribution of Python, you have access to two different terminals, the Python terminal and the iPython terminal (interactive Python). The Python functionality is the same, but iPython has some advantages, such as “magic commands” that let you run Unix commands inside iPython, and other conveniences that make it worth using. We will exclusively be using the iPython terminal in this text, when one is used.

Now, all you have to do is type

```
print('hello world!')
```

and press <Enter> — the terminal will respond by showing your phrase in the output of the line below. Let's briefly unpack what just happened though, because understanding print statements is going to be important later on.

Definition 2.1.1 A **print statement** is a line of code which tells the interpreter (the thing that turns your "English" commands into computer bits and executes them within the bowels of your computer) to output something to the screen.

Quite literally, you can think of it as "printing" the value to the screen itself. In the case we used here, it didn't seem very useful... since I told it to print 'hello world' in the first place. But as we will find out in a moment, the beauty of coding is that you can save numeric (and other) values into little containers called variables, and no longer have to keep track of their intermediate values as they get pushed through lines of calculation. **But**, there are plenty of times when either your code is going wonky, or you want to double check that those intermediate values make sense. Those are perfect places to stick a print statement, which will output those values to the screen so you can manually evaluate them.

The other thing going on here was the phrase hello world being inside quotation marks, and being surrounded by parenthesis. Let's start with the parenthesis. Here, 'print' is acting as a *function*, something that takes an argument and returns a result. Think to math class, where you might write sin(x). The x would be the argument of the function sin, and is connoted by the parenthesis. The quotes, on the other hand, are described in the next section — they are the 'string' data type, and can be either single or double quoted. What are data types? Glad you asked.

2.2 Data types

Python, like most programming languages, divides up all the possible "things" you can play around with into what are called ***data types***.

Definition 2.2.1 **Data types** are the fundamental building blocks of a code, a property that every object/element/variable in a written code will have, and which will determine the rules by which Python operates on them.

Some of these divisions seem obvious: clearly a word like "cat" is a fundamentally different data type than an list of numbers [1,2,3,4,5]. Other divisions seem more arbitrary at first glance: For example Python makes the distinction between integers (the counting numbers), and floats (numbers with decimals). It does so because of the way computer processors store information in bits, but it leads to the interesting (and important) characteristic that "42" and "42." are different in python, and take up different amounts of computer memory. Some basic data types are listed and defined below, and you will learn more about them as we use them:

1. **Integers**: The counting numbers. Ex: -1,0,1,2,3,4,5, ...
2. **Floats**: Decimal numbers. Ex: 1., 2.345, 6.3, 999.99999, ...
3. **Strings**: An iterable data type most commonly used to hold words/phrases or path locations. Denoted by single or double quotes. Ex: "cat" , "/home/ipasha", "1530", ...
4. **Lists**: Stored lists of any combination of data types, denoted with brackets. Ex: [1,2,'star','fish'] or [1, 2, [3, 4, 5], 'star'] (notice that you can have lists within lists)
5. **Numpy Arrays**: Like lists, but can only contain one data type at a time, and have different operations. Defined in numpy, not native python, but so ubiquitous we include them here.
6. **Tuple**: Also like a list, but immutable (un-changable). Somewhat like a read-only list. These are defined with parentheses. Ex: tuple1 = ('hi', 1, 4, 'bye')
7. **Dictionaries**: A collection of pairs, where one is a "key" and the other is a "value." One can access the "value" attached to a key by indexing the dictionary by key:

   ```
   »dictionary_name['key']
   ```

   (more on this later).
8. **Boolean**: A data type with only two possible values: True, or False. They are used in conditional statements.

2.3 Basic Math

Within the python interpreter (or indeed in any written code) you can perform simple to very complex mathematical operations. Let's see how adding and subtracting works in ipython.

```
[IN]: 3 + 5
[OUT]: 8
[IN]: 9 - 3
[OUT]: 6
```

We can also test out multiplication and division (denoted in python with * and /):

```
[IN]: 4 * 3
[OUT]: 12.0
[IN]: 1 / 2
[OUT]: 0 OR 0.5
```

Whether you got 0 or 0.5 in that last step depends on if you are running python version 2.x or python 3.x. Python 2.x versions will tell you it's 0, while python 3.x versions will tell you it's 0.5, which seems pretty stupid, since clearly 1/2 is never 0 (unless we've jumped into some strange parallel reality). Also it seems a little dumb that different python versions would tell you different answers to the same operation (imagine the headache of converting a large body of code from one to the other)!

The reason we are getting 0 in python 2.x here is that python 2.x is performing integer division, meaning the answer has to be an integer. In this sort of situation, python simply rounds down to the nearest integer. The solution to this is to cast either the "1" or "2" (or both) as floats rather than integers. Only one is required to be a float because if one number in an operation (like addition, subtraction, multiplication, division, exponentiation, etc) is a float, it will convert all to floats and express the answer as a float. Now, 90% of the time you will need to be doing float division anyway, so the creators of python 3.x decided to make that the default division method. For your general knowledge, there is a function for converting integers to floats, and it looks like this:

```
[IN]: float(2)
```

However, there is a much faster way to create floats when you are entering a number manually, which is simply to add a decimal (period) to any number. Try it yourself: demonstrate that 1./2 and 1/2. both output the proper answer. The place when the float() command comes in handy is when you have a variable (say, called "x") in your code, and you don't necessarily know what its value is, perhaps it is the sum of many calculations, but is just an intermediary holding value. If before the next stage of calculations you require it to be a certain data type, you can use this hard casting, like

```
[IN]: x = float(x)
```

or

```
[IN]: x = int(x)
```

(Which will convert it to an integer if it is not already). The change from python 2.x to 3.x has been painful for many reasons, but one of them has been the fact that any old code that actually made use of integer divisions as a default now have to be changed.

The other basic math operation in python is exponentiation. In python this is denoted with a double asterisk ('**'). For example:

```
[IN]: 2**3
[OUT]: 8
```

To perform more complicated math like sin, cos, sqrt, etc., requires the use of some additional packages, which is the primary focus of Chapter 3.

2.4 Variables

While using Python as a calculator can be fun, the real power of programming comes in being able to store things (numbers, lists, etc) as ***variables*** and access them later.

Definition 2.4.1 A **variable** is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Declaring variables in Python is easy; you simply type a desired variable name, an equal sign, and what you want it to be. For example:

```
[IN]: x = 5.0
[IN]: y = 'cat'
[IN]: Berkeley = 'no life ' + 'bad grades ' +'no sleep'
```

would set the variable x to the floating point number 5, set y to the string "cat", and set Berkeley to the concatenated string "no life bad grades no sleep" (more on string concatenation in a bit).

R Throughout the rest of this book, I will, for the purpose of providing examples, be setting variables and modifying them, etc. I would just like to note that my choice of *name* for these variables is irrelevant, and chosen, when applicable, to be representative of what is contained in that variable. If you are following along in a terminal, you can choose whatever variable names you want, so long as you remain consistent with them.

Notice that Python doesn't output anything when you declare a variable as it did when you entered a math operation. But rest assured, those values are stored in the computer. If you type:

```
[IN]: print x
[OUT]: 5.0
```

it will output the value attached to your variable. The print command is almost always how we check in to see what a variable's value is at a given point in a code, and it's an extremely useful way to begin debugging your code if something isn't working the way you think it should be. Note that in Python 3.x and onwards, the change was implemented that for consistency, "print" should operate as a function, and thus, in 3.x onwards you would do the following:

```
[IN]: print(x)
[OUT]: 5.0
```

It's perhaps useful to note that in Python 2.x, print(x) used as a function works perfectly fine, and if you get into the habit of using it that way all the time, translating code between 2.x and 3.x will be much easier. As a shortcut, in any iPython terminal, simply entering a variable and hitting <Enter> will print the value:

```
[IN]: y
[OUT]: `cat'
```

Variables in Python are mutable — that is, you can change them, within certain bounds. Most simply, if you consecutively typed:

```
[IN]: x = 5
[IN]: x = 3
```

then printed "x" you would find it is equal to 3. You can also use variables to change themselves:

```
[IN]: x = 5
[IN]: x = 2 * x + 3
```

In this case, the new value for x at the end of the line would be 2 times the value of x going in, plus 3. (in this case, 13). You can also add, subtract, and multiply variables, if they are of the right data type:

```
[IN] : x = 5.
[IN] : y = 6.
[IN] : z = x + y
[IN] : x = 2 * z
[IN] : y = x / z
```

That is probably a bit confusing to follow, and illustrates why typically we avoid such oft redefining of variables, and instead come up with new variable names to store the various sums and products.

R We can see that when dealing with floats as our data type, the math operations we are used to have the typical "mathematical" results. When dealing with other data types, the behavior of these operations is unique to that data type. For example, adding two strings ‘a’ and ‘b’ produces the single string ‘ab’— and something like 4 * ‘a’ will return ‘aaaa’. But the power raising operation ‘a’**2 is meaningless, and returns an error. We will be spending time learning which operations can be used to modify each data type, and what their various effects are, over the course of this text.

There is definitely subtly involved in determining which data types can be operated together, and in which situation casting is valid (for example, the int() function we discussed can never convert "cat" to an integer, and will throw an error). We hope to cover much of these intricacies in time, but much of it is common sense and experimentation.

2.5 Storing and Manipulating Data in Python

So far, we have been concerning ourselves primarily with the data types that are responsible for storing a single piece of information — like a single float, or a string. But remember that our primary goal for using Python as astronomers is as a tool with which to explore and manipulate data. That data could be in a whole multitude of different forms, including observational images from a telescope, catalogs of measurements taken with a scientific instrument, the output files of a supercomputer simulation, etc.

But what all of these share in common is that they represent the plural inherent in the word "data" — we use python because while we could sometimes easily perform a calculation on a star’s flux to obtain its luminosity, we might have a collection of 10,000, or even over a million stars. This is where Python comes in — and primarily, where the array, list, and dictionary data types become not only useful but essential.

2.5.1 Arrays vs Lists

Lists and numpy arrays are the natural data type with which to store data, because they allow us to dump all of our individual measurements, etc., into a single variable. One of the first questions that usually emerges from students at this stage is "So what is the difference between lists and arrays? When do I use each?"

It's a good question, and one worth taking a section (or sub section, as it were), to explore. We'll begin with lists, as these are the native data type within Python. Let's define a list to play with:

```
[IN]: my_list = [1,5,2,7,3,7,8]
```

This list of numbers could be, say, the distance in parsecs (evidently rounded) of several nearby stars. What happens if I want to multiply all the distances by 2, because my flux formula has a factor of 2d in it? Let's try:

```
[IN]: my_list*2
[OUT]: [1,5,2,7,3,7,8,1,5,2,7,3,7,8]
```

Ah, crap. It seems like the default "list" behavior associated with multiplication is to create a new list with the original list repeated N times. If you think about it, this actually makes sense — recall from our original definition of lists that they can contain any data type, and indeed any *combination* of data types. If our list contained a mix including strings or any other non-numerical data type, this operation would fail if defined this way.

Does that mean we can't multiply every number in a list by 2? No, but it does mean that we will have do utilize what's known as "iterating" to go through the list one by one and replace each value with 2 times itself. We'll get to this later, but for fun, here's a concise way to do it:

```
[IN]: [i*2 for i in my_list]
[OUT]: [2,10,4,14,6,14,16]
```

As it turns out, there's a shorter (and computationally faster) way to apply mathematical operations to every element of a collection. Numpy arrays (defined in the importable package numpy, which we will talk about soon) obey the following:

```
[IN]: my_array = np.array([1,5,2,7,3,7,8])
[IN]: my_array*2
[OUT]: array([2,10,4,14,6,14,16])
```

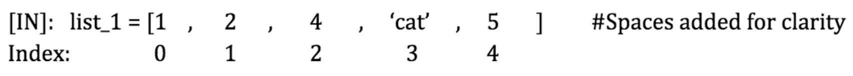
Great, so now we've shown that we can perform mathematical operations on entire arrays all at once. In a slightly more subtle point, this is also faster than the previous method shown with strings, because arrays are being treated computationally as *matrices*, which computers are *very* good at solving and operating on. So numpy's libraries are actually doing linear algebra to apply your mathematical operation to every element of the array, rather than going through and multiplying each element one by one manually.

What's the long and short of all this? Practically speaking, almost all of your data you work with will be in arrays, rather than lists. There are times, when working in your code, that it is more convenient to throw some values into a list. But particularly when dealing with large datasets, you almost certainly will be working primarily with arrays.

So now that we have all these values stored in the array container ... how do we get them out?

2.6 Array, String, and List Indexing

The procedure by which we extract subsets of a larger array/list is known as slicing, or ***indexing***, and the method is as follows: given a list, array, or string (all 1-dimensional right now for simplicity), each entry is assigned an index. By convention (that you may reasonably find annoying), this index starts with 0, rather than one (this is true for most programming languages). Below we have a sample list, with the indices for each entry listed below: i.e., '1' is the 0th entry (or element) in the list, and



```
[IN]:  list_1 = [1   ,   2   ,   4   ,   'cat'   ,   5   ]      #Spaces added for clarity
Index:           0       1       2        3         4
```

Figure 2.1: List elements and corresponding indices.

5 is the 4th. Let's say then that you wanted to extract the 0th entry from the list, to use for some other coding purpose. The way to slice a variable (of the proper data type) is by typing the variable name, attached on the right with closed brackets and an index number. For example, to extract the 0th element and set a variable for it:

```
[IN]: list_1 = [1, 2, 4, 'cat', 5 ]
[IN]: x = list_1[0]
[IN]: print x
[OUT]: 1
```

Notice that for a list, each entry is the "thing" between the commas, so typing

```
[IN]: print list1[3]
```

would print

```
[OUT]: 'cat'
```

as the string 'cat' is the third entry (if you start counting at 0).

Arrays can be sliced in precisely the same way as lists. Interestingly, strings can also be sliced. So if we had set

```
[IN]: var = list_1[3]
[IN]: print var[1]
```

then we would get an output of

```
[OUT]: 'a'
```

Unfortunately, if you have a long integer like x = 1234456653453, you can't slice through x the way you can with lists, arrays, and strings. What you can do is turn (or cast) x as a string:

```
[IN]: x = 123456789
[IN]: x = str(x)
```

Now that x is a string, you can happily index it:

```
[IN]: print x[0]
[OUT]: '1'
```

Normally if you try to convert a string like 'cat' to a float or int, python will hate you. But if you attempt to convert a string that only contains numbers, python can successfully make the conversion. So we can get the integer number of the 0th element of 123456789 like so:

```
[IN]: x = 123456789
```

```
[IN]: x = str(x)
[IN]: zeroth = int(x[0]) # or zeroth = float(x[0]) for the float
```

Sometimes we want more than a single value from a list/array/string. There is also a way to slice through multiple indices at once. The format is as follows. Take the previous example of the string '123456789'. Say we want the 0th, 1st, 2nd, and 3rd elements to be pulled, turned back into an integer, and set as the value of the variable H:

```
[IN]: H = int(x[0:4])
```

So basically, now instead of a single index in the brackets, we have a start index, a colon, and an end index. Also note, Python will go up to, but not include the end index given. As a shortcut, if you are starting from the beginning, or slicing from some midpoint to the end, you can omit the 0 before the colon, or the final index after, i.e.,

```
[IN]: print x[0:4]
```

is equivalent to

```
[IN]: print x[:4]
```

and if you don't know how long an array is but want to index it from its nth element to the end, simply

```
[IN]: print x[n:]
```

You can also slice through an array backwards using what are known as **negative indices**, that is, and index of "-1" refers to the last element in an array/list/string, and "-2" the second to last, etc. An example of indexing from the last to 5th from last element might be

```
[IN]: print x[-6:-1]
```

2.6.1 Two Dimensional Slicing

Strings and lists now primarily excluded, often astronomical data (like images from telescopes) are stored in 2d arrays — essentially a large grid or matrix of numbers described by 2 indices, a row and a column. (If it helps, you can think of the arrays above as matrices with row length 5 and column height 1, so you only needed to index which column you were interested in.)

Lets cut to the chase with an example. Let's say "A" is a 2d array that looks like this:

```
print A
[[1 , 3, 4, 5, 6]
[ 4, 5, 9, 3, 7]
[ 9, 4, 6, 7, 1 ]]
```

Notice the way python is handling the list structure here; there are three one dimensional lists stacked within an extra set of brackets (like a list of lists). We slice it with two indices, row, then column.

R Be careful, row then column translates into (y,x), which is the opposite of how we are usually taught to determine ordered pairs of coordinates.

To pull the 3 in the second row, we type:

```
»print A[1][3]
```

Alternatively, you can use the comma syntax **A[1,3]** to equal effect. To pull the 6 in the first row:

```
»print A[0][4]
```

Try it out: what would be the way of slicing to pull the 4 in the last row? Using the same colon notation from above, how would you pull a whole row?

Given a 2D array, you may want to take a chunk of it, either end to end, or somewhere in the middle. The syntax for doing so is a combination of commas and colons. Remember that colons either separate a start and end index, or refer to a whole column if no start/end are specified. Lets say you have an image with 1000x1000 pixels, which you are viewing as a 2d array of 1000x1000 values. The following is a list of example slices, from which you can infer how to slice any section you'd like.

Exercise 2.1 Slicing Images

1. »array[350:370,:]
   - takes the full rows 350-370 in the image (fig. 2.2)
2. »array[:,350:360]
   - takes the full columns 350-360 in the image (fig. 2.2)
3. » array[350:370, 350:360]
   - takes the box in fig. 2.2. (the region between/including rows 350-370 and cols 350-360)

■

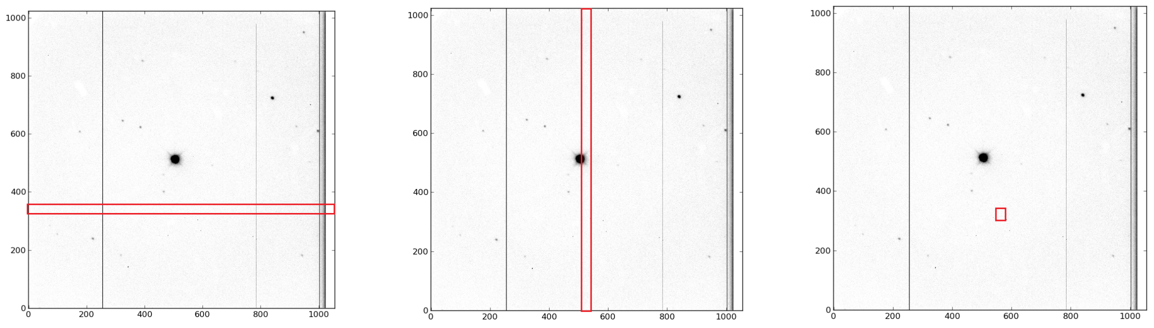


Figure 2.2: Left: Rows 350 to 370 pulled. Center: Columns 350 to 360 pulled. Right: Box of rows 350-370, cols 350-360.

2.7 Modifying Lists and Arrays

While we have shown how you can create a list of elements and how to extract and see specific values within them, we haven't talked about adding and removing, or changing, elements of lists and arrays. Say we have a list of integers as follows: [1, 2, 3, 4, 5, 6, 7].

2.7.1 Altering Elements

The most simple way to change a value within the list is to set a new value equal to the slice of that list. For example:

```
[IN]: list1 = [1,2,3,4,5,6,7]
[IN]: list1[2] = 'hi'
```

When we print the list one now, we will see that the third element of the list (formerly the integer 3) will have been replaced:

```
[IN]: print list1
[OUT]: [1,2,'hi',4,5,6,7]
```

Of course, now that there is a string in our list, we can't do things like sum(list1) and expect to get a proper value. Now let us see how to delete values out of a list. This will involve use of the 'del' command.

2.7.2 Deleting Elements

If we continue using our list1 from above:

```
[IN]: del list1[-1]
```

will delete the last entry in the list. (Note, a -1 index means the last element in a list or array, and -2 references the second to last, etc). We could also have used forward indexing just fine.

R Be careful with this command. Remember that once you delete an entry, the indexes corresponding to all the remaining values get shifted. So you could run del list[0] 3 times and it would keep deleting the "new" 0th entry in the list.

Now while the principles of what we've used apply equally well to arrays, the syntax of how everything is done will be somewhat different, due to the way numpy.array was created. We will discuss working with numpy arrays later on, after having formally introduced numpy and other scientific packages.

2.7.3 Appending to Lists

If we want to add a new value to the end of a list (an extremely common task), we simply have to type:

```
[IN]: list_name.append(new_value)
```

This is possible because 'append' is a method (built-in function) of the list object in Python. We'll see in the next chapter that appending to a numpy array, for example, has a bit more syntax to it.

2.7.4 Flipping Arrays

Before moving on we'd like to list 2 very basic image (2d array) manipulation commands that might come in handy. We will go through more in much more depth later.

Let's go back to our 1000x1000 entry 2D array. There are simple commands for if you want to flip the image vertically and horizontally. For a vertical flip (about the horizontal centerline):

```
[IN]: flip_vert_array = array[::-1] # see fig. 2.2
```

(this is shorthand for array[::-1,:] - it does the same thing but seeing it the second way makes the next command make sense). For a horizontal flip (about the vertical centerline):

```
[IN]: flip_hor_array = array[:,::-1] # see fig. 2.2
```

This technique works for 1D arrays as well — simply use

```
[IN]: some_list = original[::-1]
```

to reverse the elements in the list.

2.7.5 Concatenation

Concatenation is the process of joining two things together end-to-end. We've already seen how to do this for lists, and the same method works for strings, e.g.

```
[IN]: string1 = ‘hello’
[IN]: string2 = ‘ world’
[IN]: finalstring = string1 + string2
```

Note, we could've accounted for the space at the end of string1 instead, or made it a separate string. Concatenating arrays takes a little more work, so I'm going to punt that topic until we've covered what Numpy actually is in full!

2.8 Dictionaries

The final primary data container data type in Python are ***dictionaries.***

Definition 2.8.1 A **dictionary** is a Python container, like a list or array, but which uses "keys" instead of indices to specify elements within the container. That is, the *order* of elements (values) in a dictionary is irrelevant, and values are retrieved by indexing for the appropriate key (which can be almost anything)

Dictionaries in Python are created using curly brackets, inside which go key-value pairs (colon separated), which themselves are separated by commas, e.g.

```
simple_dict = {‘key1’:value1,‘key2’,val2}
```

where, to pull value1 from the dictionary, I would index it as

```
pulled_value = simple_dict[‘key1’]
```

We can also easily change values in a dictionary, or add new key-value pairs, using this index notation; for example, if we wanted to change val2, we would use

```
simple_dict[‘key2’] = new_val
```

and to add a new key-value pair, I would simply type

```
simple_dict[‘new_key’] = new_value
```

Note, here I have chosen my keys to be words within strings. This is not required by the dictionary data type — I could have chosen keys that were numeric (i.e. 1, 2, 3), and that would have worked fine. But the strength of dictionaries is *generally* that the keys hold meaning, and are easy to remember because they relate to what I'm placing in the values, which can be *any* data type. For example, say I wanted to tally how many of different kinds of fruits I have. I might set up a dictionary:

```
fruits = {‘bananas’:5,‘apples’:3,‘pears’:17}
```

Now we can see how I can easily query for how many pears I have, as opposed to creating a list [5,3,17] and having to keep track of the fact that pears were the 3rd entry. If I went and bought some mangoes, I could easily add them in via

```
fruits['mangoes'] = 42
```

(I like mangoes.) Dictionaries, of course, can be nested inside dictionaries, as can lists, hinting at the rich data structures one can create to house complicated sets of data (of course, the more you nest lists/arrays/dictionaries within each other, the more complicated and irregular the indexing process becomes). Later on, when we get to Object Oriented Programming, we will learn a more developed method for dealing with this kind of multifaceted data using something called classes.

2.9 Problems for Chapter 2

Problem 2.1 Create a variable num_list, and set it equal to a list containing 42,46,54,65, and 90 as elements. Then set a variable called sliced to num_list[3]. Print both to see what it outputs. Now define a variable called "total' and use the method shown above to set it equal to the sum of the first two elements in the list (do this programmatically, not by typing them in manually). What is the total you get?

Problem 2.2 Create a string called fname and set it equal to a string containing your first name. Also create a string called lname and set it equal to your last name. Concatenate the two strings into a single string that prints your full name, with a space between the two words.

Problem 2.3 Identify all the syntax errors in the following block of code:

```
list_1 = [1,2,'3',)
string = 'this is a string"
new_list = append.old_list[new_value]
second_val = some_list[2]
```

3. Libraries and Basic Script Writing

Introduction

We saw earlier that one can use the iPython interpreter to do basic math, and that there were various data types that come "preinstalled" within Python (like lists, strings, integers, etc). However, once a code requires more sophisticated analytical tools (especially for astronomical processes), it becomes apparent that the vanilla iPython functions are not sufficient. Luckily, there are hundreds of functions that have been written to accomplish these tasks, most of which are organized into what are called **libraries**.

Definition 3.0.1 A ***library*** is a maintained collection of functions which can be installed and imported into a Python code to be used. Numpy and Scipy are examples of libraries.

Most Python distributions come with a lot of these libraries included, and installing new libraries is generally straightforward.

There are 4 key libraries that we will be discussing in detail in this text: numpy, matplotlib, and (sometimes) scipy and astropy. Numpy is an extremely versatile library of functions to do the things Python can't. For example, while you can create a polynomial yourself (x**2 + 3*x + 1), Python provides no way to make sine and cosine functions. That's where numpy comes in. Matplotlib, meanwhile, is a library with functions dedicated to plotting data and making graphs. Astropy is a library with functions specifically for astronomical applications: we will be using it to import fits images (images taken by telescopes), among other things. Scipy is a library that contains special use functions that are often used in science. Since there are thousands of these functions, instead of memorizing them all, the best way to learn is to Google or query Stack Exchange for the type of function you are looking for, and you'll find the scipy or numpy function you need. The ones you use most often will then become second nature.

3.1 Installing Libraries

As I mentioned above, most scientific distributions of Python (like Anaconda) come with important packages like numpy preinstalled. However, for most smaller packages, like astropy, or pyfits, or those for programs you are using written by other scientists, you will likely have to install them yourself. The easiest way, when available, is to use pip. If a library has been added to pip, (a package installer already on most windows, mac, and linux computers), then installing a new one is as simple as typing (in the *terminal*, **not** the *python/ipython terminal*):

```
» pip install packagename
```

The easiest way to see if a package is in pip is to just try to pip install it, if it works then you're done, if it says "not found" then you might have to do some google hunting to see how to install it- usually by downloading a folder and running a python script within called setup.py by the following:

```
» python setup.py install
```

If, for example, you find you don't have pip already, you can download it from their website and install it via the above method. Then you can pip install other things. When it comes to installing small, scientist maintained packages, you will have to get familiar with Git and Github, which are version-controlled code repositories that let you update your software as the author does. That's outside the scope of this text, but there are plenty of guides online for how to clone git repositories to your computer, and then install the code using the setup instruction I listed above.

3.2 Importing Libraries

Because these libraries are not automatically loaded up when Python runs, we have to ***import*** them — you can do this in the python interpreter, as shown here, or as the first few lines of a script. The syntax variations for doing so are shown below:

```
[IN]: import numpy
[IN]: import matplotlib.pyplot
[IN]: import astropy.io.fits
```

Notice that there is a dot notation within some of the imports. This is associated with classes. These libraries are huge, and loading all of the functions in them is unnecessary if you know what you want. Since pretty much everything you need to plot is within the "pyplot" sub-library of matplotlib, we can just import that sub-library. Now that the functions are loaded, you can use them in your code. However, the syntax for using them is slightly different than that of normal python functions. Because Python needs to know where the function you are calling is coming from, you have to first write the library, then the function, using the same dot notation as above. For example, a sin function might be:

```
[IN]: import numpy
[IN]: x = numpy.arange(100)
[IN]: y = numpy.sin(x)
```

Clearly, writing out numpy all over your code would take forever. Luckily, python allows us to import the libraries and name them whatever we want for the purposes of our code. Two standard choices are:

```
[IN]: import numpy as np
```

and

```
[IN]: import matplotlib.pyplot as plt
```

We are already discovering that as the tasks we are trying to handle become more complicated and involve importing libraries, performing said tasks within the iPython terminal environment is unwieldy and inefficient. Hence, we shall write a program instead.

3.3 Writing A Program

Thus far, we have been working entirely in the iPython interpreter. While this is a quick and easy way to practice with Python, it is unsuitable for the majority of things that you might want to accomplish. Thus, most of the times we write what are known as scripts, or programs.

Definition 3.3.1 A **program** is a self-contained list of commands that are stored in a file that can be read by Python. Essentially, it is a text file, with each line being the exact syntax you would have typed into the terminal. Python then opens up your program and runs it through the interpreter, line by line.

For example, if this is what you did in interpreter before:

```
[IN]: import numpy as np
[IN]: import matplotlib.pyplot as plt
[IN]: x = np.arange(100)
[IN]: y = x**2 + np.sin(3*x)
[IN]: plt.plot(x,y)
[IN]: plt.show()
```

then you could write a program in a text file that looked like this:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(100)
y = x**2 + np.sin(3*x)
plt.plot(x,y)
plt.show()
```

You would type this up in any plaintext text editor (popular examples include vim, emacs, sublime text 2, pycharm, atom, etc), and save it as something like ‘simple_program.py’. Then to run it, simply open up the interpreter (in the same directory as the file) and type:

```
[IN]:run simple_program.py
```

and your plot will be output. Note that if your code doesn’t involve any interactive elements, you can also run it from the regular terminal via

```
» python simple_program.py
```

There are innumerable advantages to writing scripts rather than working directly in the interpretor, most of which are hopefully self evident. Your code will then be transportable (between computers, people, etc.). You can adjust a single element (or fix a mistake) and rerun your script without having to retype every line (which would be required in the terminal). So without further ado, lets jump into some exercises that deal with the basics of script writing.

Exercise 3.1 Loading 1-dimensional data from a file & plotting
The first step needed to work with any astronomical data is, of course, to load it into your code. For that, we utilize a numpy function called **loadtxt**. Let's say we have a file on our computer called "spectrum.txt" that contains two columns, wavelength and flux.

We can load these two columns of data into python in several ways. The first is to specify a general variable that just stores everything in the file in a single container:

```
data = np.loadtxt(`spectrum.txt')
```

What has python done when the interpreter ran this command? We can simply print **data** to find out. If you try, you'll see that data is a single array that contains multiple (x,y) pairs, one for every *row* in the original data file (this would be an (x,y,z,...) pair for files with more columns). As it turns out, our plotting module matplotlib.pyplot requires us to feed it an array "x" with all our x values and an array "y" with all our y values. To get our array of tuples to look like this, we are going to *transpose* it using a numpy command:

```
data = np.transpose(data)
```

where I have simply overwritten the previous data variable with the now-two-dimensional array containing the first column as its first index and the second column as its second. I can now index data to pull out the wavelengths and fluxes:

```
fluxes, wavelength = data[0], data[1]
```

where I have used a shortcut to quickly define both variables in a single line — one could split it into two lines if one wished. Now, if I want to (as simply as possible) plot the two, I would run the following:

```
plt.plot(x,y)
plt.show()
```

which would create and show the plot in a new window. There are a million ways I could adjust how exactly the plot looks, but we will get there in Chapter 5.

As it turns out, there's a slightly shorter way to do the above. We can use the "unpack" parameter of loadtxt to immediately split it along the columns into separate variables. I'll jump straight to the example, and then explain the steps:

```
wavelengths, fluxes = np.loadtxt(`spectrum.txt',unpack=True)
```

What's going on here is that the optional argument "unpack=True" within the loadtxt function tells it that I want to load it column-wise, and that I am willing to define as many variables on the left hand side as there are columns in the file. In a more general case, there may be more columns in a file than you want to load, but you can easily use the same formulation:

```
wls,fls=np.loadtxt(`spectrum.txt',usecols=(0,1),unpack=True)
```

where the usecols option tells it which columns to choose (same number as the number of variables you define on the left, in order to not throw an error). Notice I've also begun the traditional, lazy coding practice of using as few characters to define variables while preserving meaning. You'll soon do the same. I can now plot the data in the same way as I did before. ■

We've seen how we can use loadtxt to load multicolumn data into python and generate a basic plot. For now, we are ignoring the thorny issues of if your data isn't "regular," that is, each column is the same length and filled with a proper value. Loadtxt is ill-equipped to handle such such cases. But numpy also provides a more versatile function **np.genfromtxt** to handle such cases, and the documentation for it is extensive.

On that note, how do we know how a function actually works — what its inputs and outputs are? For now, you've been taking our words on it. But no need! Besides googling a package function to find its usage online, we can do this straight within the interpreter. Simply type

```
[IN]: help(np.genfromtxt)
```

(plug in your function of choice) and Python will give you a helpful rundown of how the function works. To advance through the documentation, keep hitting <Enter>, or hit "q" to exit out of it.

3.4 Working with Arrays

Earlier we discussed how you can initialize a list, add to it, replace values in it, etc. We will now repeat the discussion with the syntax for numpy arrays, given that we now know how to import numpy into our code.

3.4.1 Creating a Numpy Array

Here's a bunch of ways to initialize a basic numpy array:

Exercise 3.2 **Methods of initializing numpy arrays:**
empty = np.array([])
zeros = np.zeros(len_desired) # creates an array of zeros
ones = np.ones(len_desired) # creates an array of 1's
twos = np.ones(len_desired)*2 # creates an array of 2's
count = np.arange(start,stop,step) # creates an array of integers from start to stop in jumps of step
resolution = np.linspace(start,stop,num) # creates an array of floats from start to stop with num equally spread values
logresolution = np.logspace(start,stop,num) # creates an array of floats from 10**start to 10**stop with num logarithmically spread values ■

3.4.2 Basic Array Manipulation

Remembering back to lists, the syntax for appending was listname.append(newvalue). For arrays, we call the specific function

```
arrayname = np.append(arrayname, new value)
```

If you need to change a value in an array, the syntax is identical to before, simply set

```
arrayname[index] = new value
```

to change it.

To delete values from an array, you can use

```
arrayname = np.delete(arrayname, indices)
```

where indices can be a single index or a range.

To insert values into an array, call

```
arrayname = np.insert(arrayname, index, value)
```

and your value will be inserted before the index specified.

If you want to append one array onto the end of another (i.e., concatenate them), you can't use the '+' syntax used for strings and lists, because you'll end up making a new array, the same size as the originals, with each new value being the sum of the two values in corresponding positions in the original arrays. Instead, we need to call

```
np.concatenate((arr1, arr2, ...))
```

to join them together.

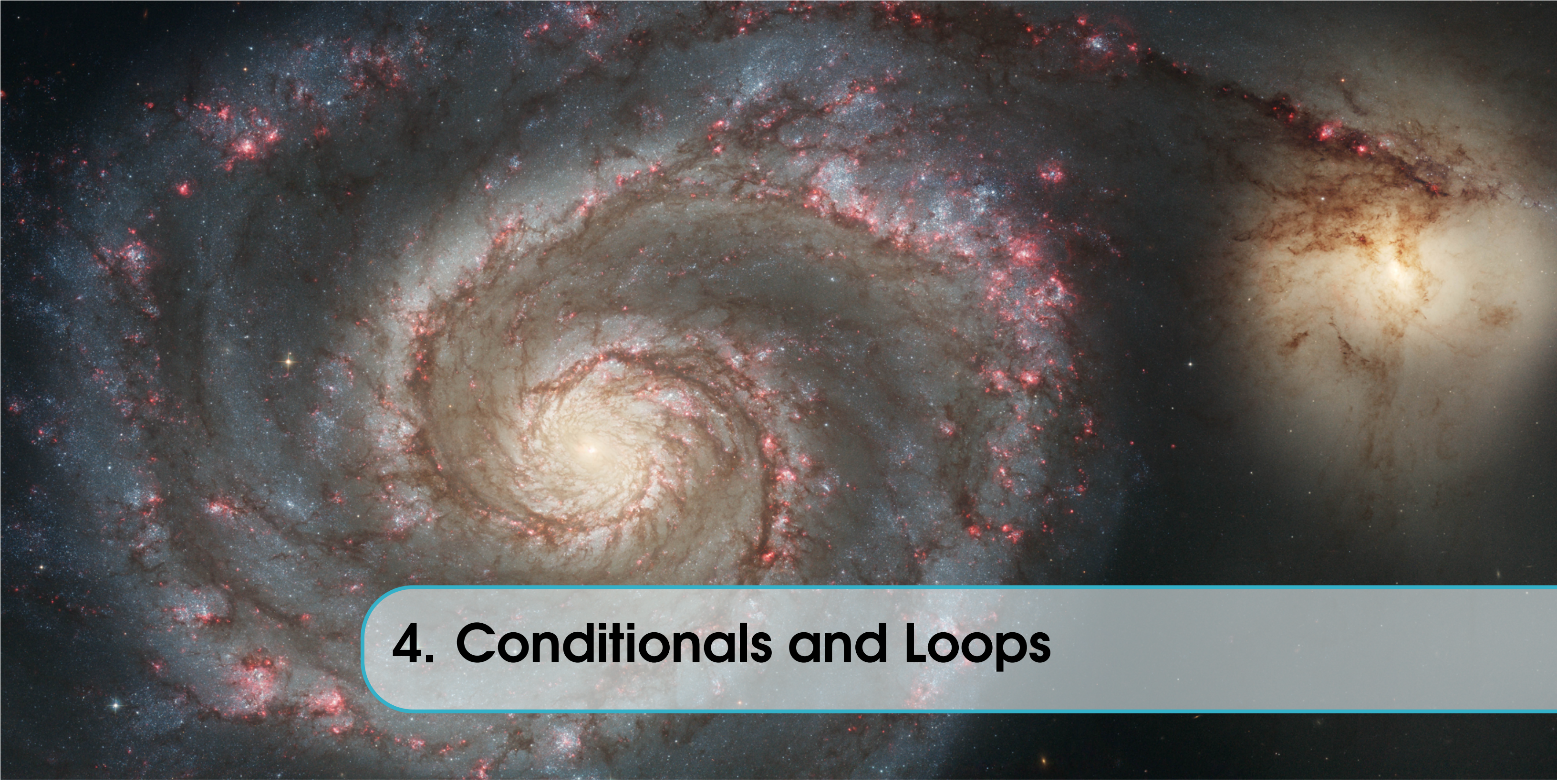
Alternatively, if you have an array you need to split up, you can use

```
np.split(arr, indices)
```

If you specify a single number, like 3, it will attempt to divide your array into 3 equal length arrays. If you provide a range of indices in order, it will know to split your array at those spots.

There is a ton more fiddly things you can do with arrays, particularly once you start working with 2 and 3 dimensional arrays. We will touch on that in Chapter 5, but primarily the scipy documentation and the web are good resources for learning about numpy array functions.

R Even through operations like concatenating two arrays, or adding a row or column to an array seem natural, numpy is actually a little bit annoying in the way it handles them, which is by making copies of all the arrays involved and working with the copies. Normally this is not a big deal, but when you have very large arrays it can start to significantly using your computer's memory. The best way to avoid this, if you have the fore-knowledge to, is to initialize your array at the beginning to be the size of the largest array you'll need to work with (and just fill it with zeros), and then adding a column becomes as simple as setting the index of that column to new values. But critically, this involves no copying or creating of new arrays. This is almost never a concern when working with smaller (like, less then a few thousand entry) arrays.



4. Conditionals and Loops

We saw in Chapter 3 how to create programs and run them in Python. That powerful structure allows us to save text files containing coherent sets of Python commands which Python can run for us all at once. As of now, understanding how Python interprets our simple programs is easy: it takes each line and enters it into the terminal. The real power of programming, however, lies in our ability to write programs that don't just contain a list of sequential commands. We can write code that repeats itself automatically, jumps around to different sections of the document, runs different functions depending on various inputs, and more.

4.1 Conditional Statements

As you might have guessed from the chapter title, we create programs like this by implementing various ***conditional statements*** and ***loops***.

Definition 4.1.1 A **conditional statement** begins a defined, separated block of code which only executes (runs) if the conditional statement is evaluated by the interpreter to be "true". Essentially, you are telling the computer "only run this block of code IF some condition is true." The condition itself is determined by the programmer.

Let us start with some examples of conditional statements. The primary conditional you will use is "IF". The syntax for creating an if-statement is as follows:

■ **Example 4.1 A Simple Conditional**

```
x = 5
y = 7
if 2*x**2 >y**2:
  print(''Wow, thats cool!'')
```

■

We start the line with the word "if", which is a special word in Python (and your text editor will probably color it differently) that tells the interpreter to evaluate the truthiness of the rest of the line, up to the colon (again, the colon is important, don't forget it). In the case above, the if-statement would indeed print "Wow, that's cool!", because $2*(5^2) = 50 > 49$. In this case of course, because x and y were simply defined to be numbers, the condition would always be true, and the print statement would always occur. But most of the time in your code, you have variables which are arrays, or parts of arrays, and the values have been changed in various steps of the code that you can't keep track of. Also note that, like for functions, all lines to be considered part of the conditional must be indented one tab.

To create a conditional with an "equals" condition, you have to use the strange syntax of the "==" double-equals, in the spot where you otherwise had > or <. The reason for the "double-equals" notation is that in python, a single '=' sign is reserved for setting the values of variables. As we will mention later, the "+=" notation means "set x = x+1" . Some other conditional combinations are "not equal," given by "!=", greater than or equal to, ">=", and less than or equal to "<=".

Conditional	**Symbol**	**Conditional**	**Symbol**	**Conditional**	**Symbol**
Equals	==	Greater than	$>$	Less than	$<$
Not equals	!=	Greater than/equal to	>=	Less than/equal to	<=

Table 4.1: Symbols for various conditional statements

4.1.1 Combining Conditionals

We are not limited to one conditional per statement; we can combine as many as we need (within reason).

Exercise 4.1 Multiple Conditionals

```
x = input(''Enter a number:'')
x = float(x)
y = 15
z = 20
if (x >y) and (x !=z):
  print ''Nice!''
if (z > x) or (x != y):
  z = x + y + z
```

So here we have 2 if-statements, with the two possible combinations of conditionals, 'or' and 'and'. These statements can be combined indefinitely (for example, if ((a and b and c) and (d and f)) or (g +1>y) demonstrates how you can combine 'and' and 'or's' to suit your needs). ■

From now on, we will begin dropping new python commands and code into our examples, and will explain them either in comments in the code, or after the example. In this example, the command raw_input('text') prints 'text' to the screen and waits for the user to enter something. Whatever is entered is stored as a string in the variable x. (So above, if you said "enter a number" and a user entered a letter, the code wouldn't work).

So using the if-statement we have been able to set off blocks of code to be run only if some combination of conditionals is true. What happens otherwise? Typically we include an "else" statement following the if block, to determine all other cases.

■ **Example 4.2 An Else Statement**

```
x = raw_input('Enter a number: ')
if int(x) == 5:
  print('Wow, this was an unlikely coincidence.')
else:
  print('Well, that's interesting.')
```

■

If your 'else' statement contains an if statement as well, you can use the "elif" command, which stands for else if. This saves you the trouble of an extra indent.

Exercise 4.2 Using Elif

```
if x < 0:
    print 'Negative"
else:
    if x==0:
        print 'Zero'
    else:
        print 'Positive
```

Can be condensed to:

```
if x < 0 :
    print 'Negative'
elif x ==0:
    print 'Zero'
else:
    print 'Positive'
```

■

So now we know how to set up a "fork" in our code, to allow it to go in different directions based on various conditions. There is another type of block which instead continues to run the block over and over as long as some condition is met (to be clear, we refer to block as the indented section of code within various loops, conditions, functions, etc). This is known as a ***while-loop.***

4.2 Loops

The two primary loops in Python are the while and for loops:

Definition 4.2.1 A **while-loop** is a set off block of code that will continue to run sequentially, over and over, so long as a certain condition is met.

Definition 4.2.2 A **for-loop** is a set off block of code that contains a temporary variable known as an iterator, and runs the block of code over and over for different specified values of that iterator.

4.2.1 While-Loops

Lets begin with a simple example of a while-loop.

■ **Example 4.3 A while-loop**

```
x = 100
while x > 5:
    print x
    x = x -1
```

■

What's going on here? We initialize x to be some value. The next line of code read by the interpreter (remember it goes line by line) tells it that as long as x is greater than five, keep running the indented code over and over. The indented code in question prints x, then sets x = x-1. Eventually, after 95 times through the loop (and 95 prints), x would become 6-1 = 5, which would no longer satisfy the while statement. The interpreter would then move on to the next line of code in the document. This brings up a very important point: you can see that if we had not included the "x = x-1" part of the code, x would *never* end up being 5 or less. Thus, your code would hang in this loop for all eternity. Luckily, if you find yourself in this situation, there is hope besides frantically shutting off the computer. Python interpreters have built in keyboard shortcuts to interrupt and stop your code from running. (In the lab computers this is ctrl+c). When using while loops, be sure you have included something within the loop that will eventually cause it to end. As a precaution, most programs that are more involved have special if statements within the while loop that will automatically break out of the while loop if, say, a certain threshold of time has passed. The rules for the conditionals themselves (the x>5 above) are the same as for if.

4.2.2 For-Loops

For-loops are one of the most powerful tools in Python. What they allow us to do is write a block of code that's like a template- it has the code we want to run, but without defining exactly "on what" the code acts. We then initialize a for-loop, picking a range of values, variables, etc., to plug into those designated spots in our block of code.

Exercise 4.3 A simple for-loop

```
arr = [1,2,3,4,5,6,7,8,9,10]
for i in arr:
    if i %2 ==0:
        print i
```

would print 2,4,6,8,10 (the even numbers). The % sign means "modulo," and the conditional would read "if i divided by two has a remainder of 0:". The letter i in this loop is a generalized iterator- when you type "for i in arr" you are telling the computer to run the block of code, replacing i in the block with the first second, third, etc. element in the array. (you could use any character/combination of characters for i, but i is standard practice (followed by j, and k if necessary). ■

The point of for-loops is that they are as generalizable as possible. In the above example, the array "arr" could be replaced with any variable that is an iterable data type. You could say, "for i in range(15)" to have it plug the numbers 0 through 14 into your block of code, wherever a variable 'i' appeared. you could even iterate over a string, and it would plug in the elements of the string (as single character strings) into your block of code.

One common iteration practice is to iterate over an ascending list of numbers equal to the length of a certain array. In this situation you could use "for i in range(len(array)-1)", where "array" is your array and len( ) is the command for returning the number of elements in an array, list, or string. The minus one is needed because the nth element of an array, list, or string is has an index of n-1.

4.2.3 Nested For-Loops

Just briefly, we'd like to mention that you can in fact nest multiple for-loops together, if you need to iterate over more than one value in your code. This often happens when dealing with two-dimensional arrays.

■ **Example 4.4 Iterating a 2D Array**

```
for i in range(len(x)-1):
    for j in range(len(y)-1):
        if arr[i,j]<1500.:
            arr[i,j]=0
```

■

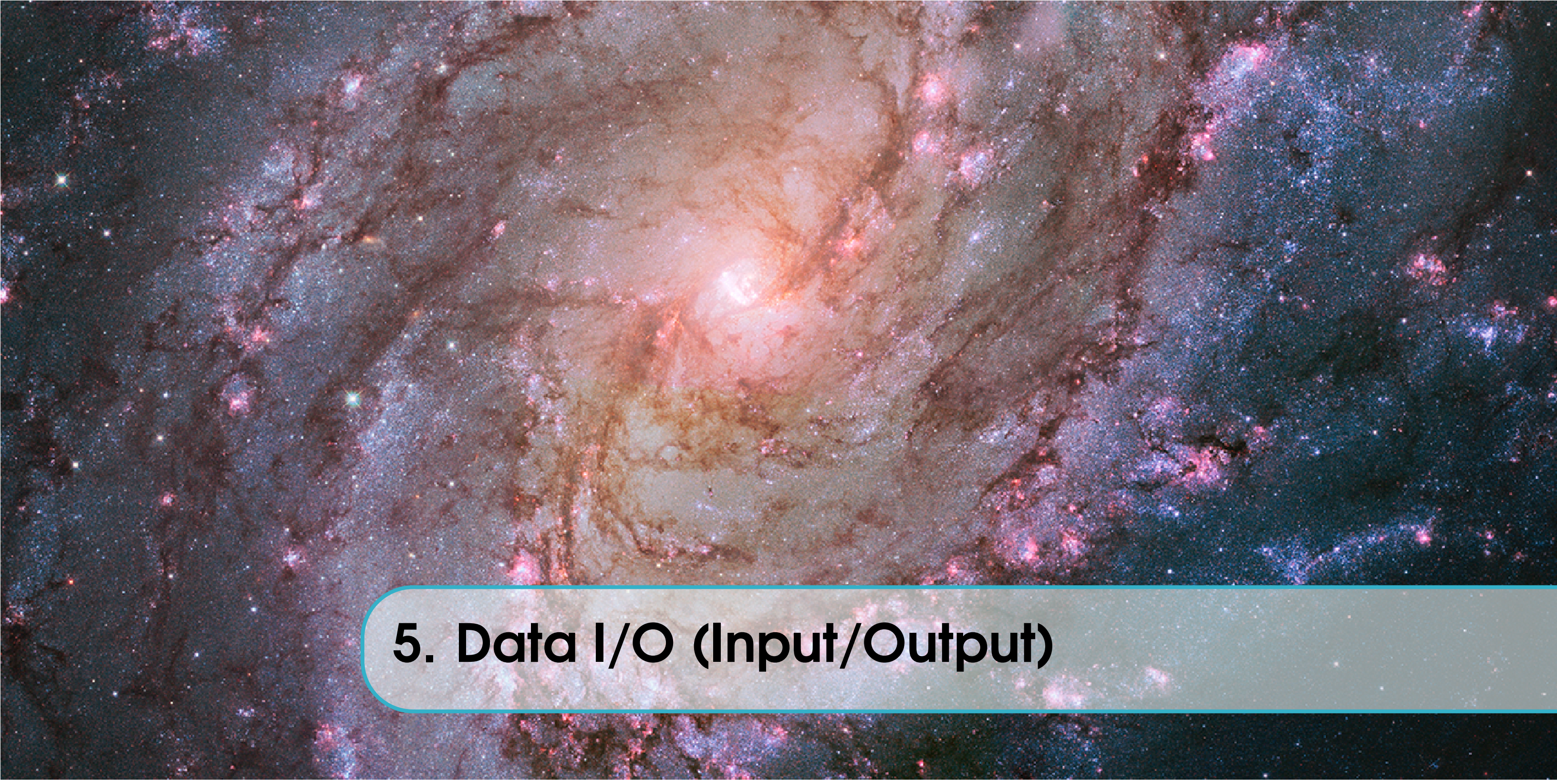
In the above example, x would be a variable representing the x coordinates in the array, with a similar deal for y. This particular block of code would run through every combination of i, j to hit every spot on the 2D array, and if the value at any given point was below the 1500 threshold, it would just set that element to be 0.

This might be a good spot to point out that the above code isn't the most *efficient* way to accomplish its task — faster, for example, is

```
array_name[np.where(array_name < 1500)[0]] = 0
```

would be more efficient. We will get into why soon, but it boils down to the fact that loops always involve performing a task over and over many times, while some numpy functions leverage linear algebra to act upon entire arrays simultaneously. A lot of the time spent after learning the basic building blocks of programming is focused on determining the most efficient ways of completing a programming task, minimizing either run time, memory usage, or both.

Believe it or not, that's all there is to basic programming. By cleverly combining for loops, while loops, and conditional statements, we can do a lot of powerful analysis. While there is a lot more to python (for example, you can introduce classes and object-orientation (chapter 6), this is all you need to do the majority of scientific coding. What is missing in the above descriptions is the multitude of python and numpy functions you will need to use along the way. A list of useful functions is included in the appendix, and we will go over many functions in class.



5. Data I/O (Input/Output)

5.1 Loading and Writing Files

At numerous points in this text we have alluded to the ability for loops to aid in the process of loading multiples files of data. Now that you know how to concatenate strings and generate for-loops, we can cover the file loading/writing process.

There are several ways of opening data files in python. Python itself has a built in mechanism for openin/writing files, and numpy also has support for file handling. To open a file in python's interface, we type:

```
»file1 = open('filename.txt','w')
```

where 'w' indicates we plan to write to the file. (We could instead use 'r' for read only, or 'a' for appending to a file that already contains data.

■ **Example 5.1 Writing to a File**

```
»file1 = open('file.txt','w')
»file1.write( 'this is a file')
»file1.write('this is not a drill')
»file1.close()
```

■

The close statement above tells python to close and save the file to the hard disk.

Exercise 5.1 Writing data to a file
You may have to analyze data in python but then export it to be analyzed more extensively by other programs. For example, you might have an array of planet distances and a second array with corresponding planet velocities that you wish to do some statistical analysis on with some other software. Likely you will want to save your data in a format that is easily usable in other programs. Thus, we can write it as such using a for loop. Assume we have already opened the file in write mode and have predefined arrays of the same length.

```
for i in range(len(planet_dist)-1):
    file.write(planet_dist[i] + ' ' +planet_vel[i] + '\n')
```

where the \n is necessary for us to create a new line when writing the file so the data will be properly divided into their respective row and column. ■

Numpy also has a file input output framework that is often useful to use. The two we will discuss here are np.loadtxt and np.genfromtxt. These are useful tools because they have many specifiable options, and load your data straight into numpy arrays that you just love to work with!

■ **Example 5.2 Loading files using loadtxt**

```
data = np.loadtxt('filename.txt')
```

Lets say the file we loaded had three columns:times, positions, and velocities. These would all be stored in data, and could be singled out as such:

```
data = np.transpose(data)
times = data[0]
positions = data[1]
velocities = data[2]
```

■

R Note: Because of the way columns/rows work in python, data in multiple columns are read in as individual data pairs. On the other hand, simply running an np.transpose on them sorts them to be 3 long separate arrays with all the times, all the positions, and all the velocities respectively.

Oftentimes data files have headers and footers- text that tells you what data is stored in the file. Of course, we don't want to try to read these into python as our data. For example, to skip a 5 line header and 3 line footer text, use

```
[IN]: data = np.genfromtxt('file.txt', skip_header=5, skip_footer=3)
```

This function is pretty versatile, and also has options for skipping columns, specifying data types, etc.

5.2 Loading Astronomical Fits Files

Being able to manipulate data stored in images is one of the most important things you should try to take away from this course. In many physical science fields and especially in astronomy, images are taken using either microscopic or telescopic techniques and each pixel in the images corresponds to a specific intensity value. The meaning of these values is dependent on the actual instruments and the physical system you're studying.

In astronomy, in particular, it is useful to work with FITS (flexible image transport system) files. FITS files are widely prevalent because of a feature they contain called a header. Headers often contain information about the image itself. For instance, they will often contain things like the declination, ascension, exposure time, as well as a description about the image itself.

FITS files are, like most things we will work with, not actually native to python itself. To use them, we will have to import a library called pyfits (or on the lab computers, astropy.io.fits); then we can start working with our images. Typically we import either of these as "pf". The syntax for opening a fits file is:

»hdu = pf.open(path)

where path is a string with the path location of the fits file, or, if your python file and fits file are in the same folder, then just a string with the filename is sufficient.

The reason we often refer to fits "files" rather than fits "images" is because a fits file actually contains more information in it than just an image. The two most important "sections" stored within a fits file are a header, and the image itself.

5.2.1 The Header

The header is a dictionary containing a lot of useful information about the images stored in the fits file: when they were taken, what exposure time was used, what type of filter was on the telescope, what the RA and DEC of the object viewed were, etc. Assuming we continue with hdu being the raw imported fits file, we can single out the header with dot notation:

»head = hdu[0].header

At this point, you could print "head" and see the entire header file. Alternatively, to see or pull individual pieces of information from the header, you query it the way you would a dictionary, using a key.

■ **Example 5.3 Pulling from a header**

ra = head['RA']
dec = head['DEC']
time = head['EXPTIME']

Note: the strings used in the dictionary call are not case sensitive. ■

5.2.2 The Image

To access the image itself, we call

»img = hdu[0].data

as the data attribute of the fits file contains the 2D array of the image, and the hdu[0] is due to the fact that sometimes fits files have multiple images stored within them.

Now, if you simply print img, you'll see a 2D array of data (likely with the whole center chopped out to save spave). To see what that array looks like as an image, you can use plt.imshow(), as we discussed earlier in the chapter.

6. Functional Programming

Thus far, we have focused primarily on writing scripts — code that is read line by line and in one fell swoop executes all the individual manipulations needed to accomplish our programming goals. However, as you begin to write longer, more complex code, the ability to organize it into individual sub-components becomes increasingly useful. The primary way we do this in Python is via functions and classes.

Functions shouldn't be scary — in fact, we've been using them this whole time! When we import a library like numpy and then call np.sin(), we are using a function someone else wrote and included within the numpy library of functions. The only difference is that now, we will be writing these functions ourselves. At first, it is easy to define functions within the scripts you are writing, but eventually, we'll talk about how to package up your own libraries of home-made functions into importable modules much like numpy.

There are two main reasons to use functions: first, they take each major "step" we are attempting to accomplish with our codes and separates them into individually testable, easily-debuggable chunks. Second, if you write your functions to be general enough, you can often simply copy and paste them from one code to another, using them when the need for their functionality arises at a future point.

Take, for example, a rudimentary pipeline for producing a one dimensional spectrum of a galaxy from the 2D CCD-image data you have stored in a bunch of fits files. You would want to write a function to read fits files into your code and perform the needed sky-subtractions, a function to identify the spectral orders in your images and one to extract (collapse) the 2D data into 1 dimension, etc.

Let's start with a simple example. In the previous chapter, we discussed how to use the astropy.io.fits() function to load fits image files into Python. As a reminder, generally, the fits.open() method reads in an object known as an hdulist, which generally has a PrimaryHdu that is the hdulist[0] (indexed at 0), though this is not universally the case. We learned in the previous chapter how to query this hdulist object ot get out the header and data needed for our work. Now, generally, within the body of my code, I'm attempting to take a fits file and simply extract the 2D array containing the science image itself. So while the astropy load function is helpful, I generally create my own wrapper function that looks like this:

```
def load_img(fname):
    with fits.open(fname) as hdulist:
        img = hdulist[0].data
    return img
```

This function assumes I've imported

```
import astropy.io.fits as fits
```

and also assumes that the science image is located in the 0th extension of the fits file. This isn't fully "flexible," but as long as I know that I'm working with data that satisfies this criterion, it makes the loading of fits images into my code a few lines faster later on. This is a good opportunity to discuss the structure of defined functions. In the above example, we have a "def" keyword, followed by a space and then the name of our function. Then we have a parenthesis set, inside which we add all the names of the arguments we want to be needed to use out function. These can be named whatever we like, as long as we are consistent about their use within the function. As we'll discuss in the next section, these names *only* matter within the function and are discarded after. Finally, we have a colon, and an indented block comprising the code we want to run when the function is called, followed by a return statement which specifies what the function spits out back into our global code when it's done.

We can make the above function a bit more flexible in the following way:

```
def load_img(fname,extension=0):
    with fits.open(fname) as hdulist:
        img = hdulist[extension].data
    return img
```

What I've done here is add a second argument to the function which specifies the extension of the hdulist the image is stored at. By setting it to 0 in the definition statement, we are setting 0 as the "default" value, and thus, if we ran the function

```
sci_img = load_img('20180403.fits')
```

without specifying an extension, it would assume 0, and use that when indexing the hdulist. But, if we knew that our science image was actually in the *first* extension, we could call the function as

```
sci_img = load_img('20180403.fits',extension=1)
```

This is a handy way to write functions where certain parameters have usual values most of the time, but you'd like the flexibility to change them on the fly. One rule to keep in mind: when laying out the arguments for a function you are writing, any arguments you want to set defaults for must appear *after* all the arguments that do not have set-defaults an require user entry.

6.1 Variable Scope in Functions

An important aspect of functions is that the variables defined and used within a function are what is known as "local in scope." That means that those variables are created when the function is called and destroyed once the output is returned- those values are not retrievable outside the function. On the flip-side, global variables (like those defined in your script that are not in functions), are accessible from within the function, whether you've listed them as an argument or not. BUT, it's bad programming practice to rely on this. Ideally, your function should only rely on the variables listed as inputs to it, so that you could move the function to any other code and not have it break. For example, examine the following block of code:

```
constant = 5
def load_img(fname,extension=0):
    with fits.open(fname) as hdulist:
        img = hdulist[extension].data + constant
    return img
```

What would happen if we ran the above block? As it turns out, the function would have no problem adding the value of "constant" to our image, since it's defined globally in our code (outside the function). But, as mentioned, If I copied and pasted this function into a different script, which didn't have a "constant" variable defined, I'd get an error. Even scarier, if I *did* have one defined but for a completely different reason in a different part of the code, I'd never know. Gah! So, remember to set any needed quantities in your functions as requirements in the function definitions. Meanwhile, the contrary is never true — any variable I define within a function and don't explicitly return in the return statement can't be accessed from anywhere else in the code (or from the command line in Python).

One important way to ensure we are doing all this correctly, as well as making things easier on ourselves later and on anyone else who may use our function is good ***documentation***. This is slightly different than commenting our code to describe what's going on (though we should do this). Documentation is a built-in feature of the way Python does functions. The way to set documentation for a function is to place it inside triple quotes right at the top of our function definition, as follows:

```
def load_img(fname,extension=0):
    "'A function to quickly extract the data extension of a fits file
    INPUTs:
        fname (string):  path/file name of the fits file to be loaded
        extension (int) [default:  0]:  extension to index
    RETURNS:
        img (array_like):  data attribute queried
    with fits.open(fname) as hdulist:
        img = hdulist[extension].data
    return img
```

When you have set documentation this way, you can run the command

```
help(load_img)
```

from the Python/iPython interpreter to pull up your documentation on what arguments your function takes and what format they need to be in.

6.2 Setting optional arguments, args, and kwargs

Optional Arguments

It's possible that you may have a function that can take many potential arguments, but most uses of the function will have the same defaults for many arguments save a few critical ones. We can take care of this by setting the variables we create in the argument of the def call equal to some default values. The rule is that these must come after the non-defaulted variables. An example:

```
def somefunction(var1,var2,var3=1,var4='cat'):
    output = str(var1+var2+var3) + var4
    return output
```

What's going on here is that anytime someone calls the function **somefunction** will have to specify values for *var1* and *var2* (in order), but technically they could stop there, as defaults for *var3* and *var4* are already set. If we want, we could specify new, non default values for them when we run the function. For example:

```
a = somefunction(2,3)
b = somefunction(2,3,6,'dog')
```

are two possible calls of **somefunction**, one of which specifies the final two arguments and one of which lets them remain their default.

Args and Kwargs

The ***args** and ****kwargs** commands allow us to feed variable numbers of arguments to a function. If you look up at how we've defined functions above, you'll see that it specifies a number of inputs. If you tried to call **somefunction** with 5 inputs, python would complain that you are giving **somefunction** 5 arguments when it only takes 4. But say you have a different part of your code which will generate several outputs, but you don't know how many, or it might vary every time the code is run on different data. You want your function to be able to handle taking 3, or 5, or 7 arguments if needed. That's where ***args** and ****kwargs** come in. Let's jump straight into an example and pick it apart afterwords.

```
def test_function(farg,*args):
    print 'formal argument:', farg
    for arg in args:
        print 'new arg:', arg
```

So what's going on here? the formal argument *farg* is read in like a normal argument. We could have any number of these. But we've specified the last argument as ***args**, which tells python "Hey, you're gonna get some unknown number of inputs after this- stick em all in a list called 'args' for me.' Then, within the function, you can iterate through the list of extra inputs (using that for loop which comes next chapter), and do things with them individually. Even without the loop, you could do something like

```
print len(args)
```

to show how many extra arguments got passed to the function.

So if those are ***args**, what about ****kwargs**? Keyword-args are very similar to the ***args**; they let you pass a variable number of extra variables to the function. The difference is, when you feed those extra arguments into the function, you individually give each a new keyword by setting it equal to it in the function call. Then, instead of putting all the extra arguments into a list, they are put into a dictionary where each value is linked to the key and can be accessed via dictionary style slicing. For example:

```
def kwarg_examplefunction(farg, **kwargs):
    print 'formal argument:  ', farg
    for key in kwargs:
        print 'argument:  ', kwargs[key]
```

We havent talked too much yet about dictionaries, so don't worry if you haven't used them much yet. But the idea here is that the ****kwargs** tells python "Hey, you're about to get an unknown number of values, each accompanied by a key- stick those in a dictionary for me so I can figure out what to do with them." The way, in closing, to call this function would be

```
kwarg_examplefunction('tree',arg1='cat',blah='dog')
```

which would print ’tree’, ’cat’, ’dog’. This is a more advanced part of function writing, usually not necessary until you are writing more complex functions, so don’t worry too much if it’s initially confusing.

7. Plotting

While we introduced the matplotlib library, and occasionally used plotting in examples, we'd like to go into more detail about plotting here, as being able to produce graphs and plots is not only important for use in scientific papers, etc., but also being able to quickly visualize data properly will save you a lot of time when working with large data sets.

7.1 Basic Plotting

Let's start with the basics. Say we have an independent variable (like time), and a measured variable (like position). This type of data could easily be read in from a 2 column text file and then plotted against each other.

Exercise 7.1 Plotting x vs. y

```
import matplotlib.pyplot as plt
import numpy as np

file_name = '/home/sally/data.txt'
data = np.loadtxt(file_name)
times = data[0]
positions = data[1]
# Now let's plot the data
plt.plot(times,positions)
plt.show()
```

■

R Note that in python when plotting, the first argument is an array of x values and the second value is an array of y values, and the number of elements in the two arrays must match.

If you want to try this example, try creating an array of times using np.arange(1,11), and position values as an array you define manually: (using position = np.array([1,2,6,34,56,57,...]). Make sure that the number of positions you make is the same as the length of the times array. Try plotting as we did above. You'll notice that the default way python plots is by plotting the positions against the times and connecting them with blue lines. Now, as scientists, we know that raw data shouldn't be connected- what we were graphing was individual pairs of points. When using plt.plot, there are other optional settings you can specify. We will focus on color and linestyle. First, lets attempt to plot just discrete points, without a connecting line. Fig. () has a chart of how to specify colors and symbols within the plot command. If you choose a discrete symbol (like 'o' for circles or '+' for plusses), then python won't connect them automatically.

You can use a matplotlib shortcut to simultaniously choose a color and linestyle as follows:

```
»[IN]: plt.plot(times, positions, 'ro')
»[IN]: plt.show()
```

would plot the discrete points as red circles, while

```
»[IN]: plt.plot(times, positions, 'b+')
»[IN]: plt.show()
```

would plot the discrete points as blue plusses. You can also specify the size of the symbol by including the argument ms=10 (play around with the number till you get the size you want).

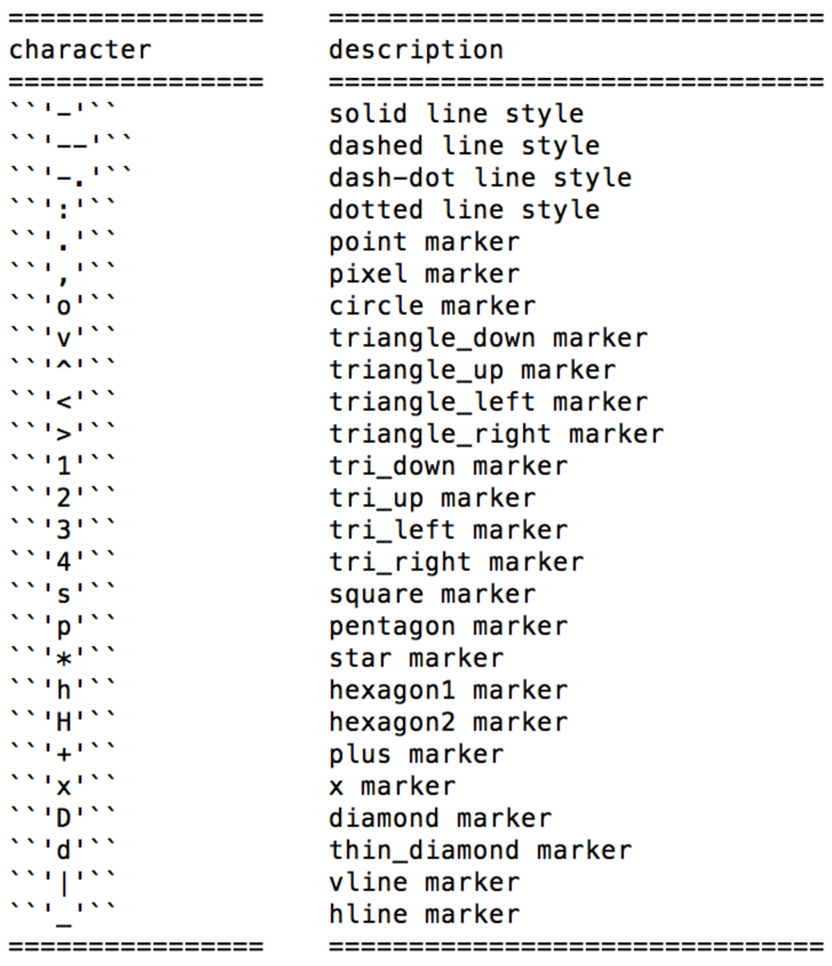
If you are plotting multiple dependent variables against one axis (say, positions of multiple objects over the same time intervals), you'll want to create a legend to show which is which. to do so, use the optional command "label" within your plt.plot as follows:

```
»[IN]: plt.plot(times, positions_obj_1, 'k+', label='car one')
»[IN]: plt.plot(times, positions_obj_2, 'bo', label='car two')
»[IN]: plt.legend()
»[IN]: plt.show()
```

The labels defined in the plot functions will now show up in a legend. You can also comment out the plt.legend line- the labels will still exist but no legend will be shown. Legend has some optional inputs as well, primarily the one you need is plt.legend(location=1), where 1 is a number 1-4 corresponding to the 4 corners of the plot. So if you find your legend covering up some of your data, try moving it to a new location.

One helpful plotting command to use is plt.ion(). This stands for interactive. It doesn't take any arguments. You may notice that generally when you use plt.show(), a plot pops up, and then your terminal stops accepting inputs until you close the plot. If you have a plt.show within the body of a large code, the rest of the code won't run until you manually close the figure. Using plt.ion() once, before all the plotting, will make it so that the plot opens, but the code continues running, and the terminal is still accessible. This allows you to make multiple plots in a row pop up as well. We recommend just putting plt.ion() right at the top of your programs next to the matplotlib import.

As you may have noticed above, if you type plt.plot(any arguments) multiple times, the graphs appear on the same plot (so you only want to do this when they share an axis). If you want to make two separate plots in a row , with different axes and such, just type plt.figure() in between the plots you are trying to make (so, for example, in between the first plot's .show() and the second plots first .plot()).



```
================    ===============================
character           description
================    ===============================
``'-'``             solid line style
``'--'``            dashed line style
``'-.'``            dash-dot line style
``':'``             dotted line style
``'.'``             point marker
``','``             pixel marker
``'o'``             circle marker
``'v'``             triangle_down marker
``'^'``             triangle_up marker
``'<'``             triangle_left marker
``'>'``             triangle_right marker
``'1'``             tri_down marker
``'2'``             tri_up marker
``'3'``             tri_left marker
``'4'``             tri_right marker
``'s'``             square marker
``'p'``             pentagon marker
``'*'``             star marker
``'h'``             hexagon1 marker
``'H'``             hexagon2 marker
``'+'``             plus marker
``'x'``             x marker
``'D'``             diamond marker
``'d'``             thin_diamond marker
``'|'``             vline marker
``'_'``             hline marker
================    ===============================
```

Figure 7.1: How to specify different symbol types in matplotlib

7.2 Plotting in Detail

The above descriptions of setting up plots utilize a shortcutting feature built into matplotlib — in short, typing plt.plot actually does several steps for you, including setting up a figure object and axis object. These are where things are actually plotted, and for us to be able to fine-tune our plots we are going to want to set these up ourselves so we can access their properties. The most straightforward way to do this is the following:

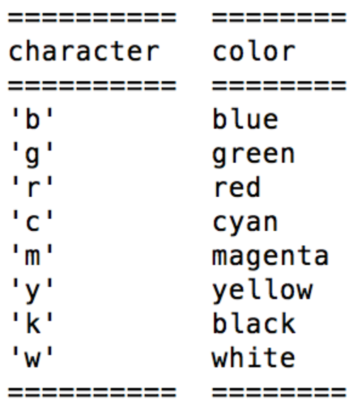
```
fig, ax = plt.subplots()
```

This command has multiple arguments you can feed it, which allows you to set up multiple axes in the same figure (i.e., multiple plots), as well as set the size of the figures. For example, we can set a size by using

```
fig, ax = plt.subplots(figsize=(5,5))
```

And we can set up two plots with a shared axis by using

```
fig, ax = plt.subplots(2,sharex=True)
```



character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

Figure 7.2: How to specify different symbol/line colors in matplotlib

For now we are going to simply focus on the basic case, and see how to manipulate the figure and axes objects we've defined.

You can think of the figure object as being the canvas onto which you are laying down plots, all of the properties of which are contained within axes. So, no matter how many plots you are putting on a canvas, the overall size of the canvas is something we would define or change by manipulating the figure object. Meanwhile, anything we want to change about our actual plots (like symbols, axis tick parameters, limits, etc) are controlled by adjusting the axis (or axes) object(s).

Once we have created a figure and axis as shown above, we can plot data in it directly by using the ax.plot command:

```
fig, ax = plt.subplots()
ax.plot(x,y,'ro')
```

We can also make all the standard adjustments that we used to do using plt commands, e.g.,

```
ax.set_xlim([1,10])
ax.set_ylim([20,100])
ax.set_xscale('log')
```

and more. So far though, we haven't seen a real reason to use this methodology compared to the previous. However, as we get further into the nitty-gritty of adjusting our plots, we're going to start adjusting parts of the axes that are either difficult or impossible to do the other way. The first example I'll use is tick params:

```
ax.tick_params(which=`both',axis=`both',direction=`in',top=True,right=True)
```

So what just happened? Here, we adjusted the ticks in our plot, specifying we wanted to adjust both axes (x and y), and both types of ticks (major and minor), to be facing inwards from the axes, and we also added ticks to the right and top of the plot, which are not turned on by default. This is just a single example, but the point is that as you start wanting to fine-tune your plots, you'll want to be using the fig, ax framework to have full control.

7.3 Subplotting

7.4 Plotting 2D Images

Earlier we discussed 2-D arrays. Pyfits/the Astropy libraries have a way of displaying these as images. The easiest way to think about a 2-D array in terms of plotting is to pretend it is a black and white image. Each "pixel" is a value within the array. Some pixels might have low numbers (not bright), others higher (very bright). Matplotlib can generate an "image" based on this data- it simply assigns a color-table to follow the varying brightnesses, and displays the strength/intensity of each 'pixel' exactly as you would expect. (This is actually not a bad way to think, since images taken by telescopes are simply 2D CCD pixel arrays "counting" how many photons hit each pixel and returning a 2-D array with the totals).
Lets say we used astropy to read in a fits image, and turn it into a 2-D array (we cover how to do this in the next section). Now we have a two dimensional numpy array, with array[0,0] giving the number of photons in the top-left pixel, and so forth. To plot it, we would type:

```
»[IN]: plt.imshow(array, cmap='gray_r')
»[IN]: plt.show()
```

R Note: for reasons that really don't matter, you don't have to call plt.figure() before using imshow, even if it is after plotting other things. It's a different kind of plot, and will show up in its own figure on its own.

In this example, we chose a cmap (color map) to be gray_r, which is essentially "reversed black and white". Most of the time, when viewing images from telescopes, we want to use this setting, even though there are many wacky and colorful color-maps to chose from. The primary reason is that the astronomical image (unless taken with a specific filter), contains information only about raw numbers of light particles. So there is only one gradient- which is easily modeled as a transition between black and white. So why the reversal? Using the 'gray' cmap alone produces images that look quite a lot like the night sky anyhow. The answer is that when trying to pick out faint objects and stars, it is easier to see contrast between dark things on light backgrounds than the reverse. Furthermore, often times these figures end up in papers which are printed, and reversing the color-scheme saves on ink.

Plotting 2-D arrays, whether real images or other values, can sometimes be tricky. You are looking to get a certain level of contrast between light and dark, which maximally displays the information in the array (you don't want it to be washed out, or not visible). We encourage you to see the documentation for plt.imshow() to see how to select different scales (linear, quadratic, logarithmic, etc). This will also be covered in the image processing tutorial. An easy way to start pulling useful ranges within an images are the vmin and vmax commands. They are used to set the upper and lower range of the linear (by default) scale between black and white. Basically, if you set vmin=50, and vmax=500, it would create a linear scale from pure white to pure black between these two values; anything less than 50 is white, anything higher than 500 is just black. What this seemingly does is take away your ability to discern by a gradient a pixel of 500 and a pixel of 600. This is true. But when viewing astronomical images, often times there are several bright objects (like stars or galaxies), and a mostly black background (the sky). Since the difference between the sky brightness and object brightness is so huge, it doesn't make much sense to attempt to see the "gradual" shift between them. Furthermore, in a simple plot like this, we only really care what is "not-sky" and what is "sky", so we want a high level of contrast. If you need to know which pixels in a given star are brighter, and which are dimmer, comparatively, you would probably want to be more quantitative and write a piece of code to determine that for you.

R A boring but important (sometimes) note: The convention amongst astronomers and scientists in general is the the "origin" of an image is in the lower left hand corner, (0,0), i.e., what we see is the first quadrant of a coordinate plane. Unfortunately, matplotlib has other ideas. When you use imshow, it displays like a matrix, the way arrays are defined, with (0,0) in the upper left corner. If you want to conform to convention and plot with (0,0) in the bottom left (which you should), you'll want to use the command origin='lower' within your plt.imshow command. Unfortunately, doing THIS will end up flipping your image vertically. Sometimes, astronomical images come in upside down anyway (to the convention of north being up and east being left). Then origin='lower' actually solves your problem. But if your image was rightside up when plotted before, it is now upside down. Luckily we know how to fix this: just set your image = image[::-1] to flip the array, before plotting (and therefore flipping it).

8. Classes and Object Oriented Programming

Object Oriented Programming is a relatively new system of organizing programs which is included in (or is the basis of) many higher level programming languages. There are many overheads associated with choosing to structure your research code in an object oriented (or OOM) scheme, but there are also payoffs that sometimes make it worth it.

Think, for a minute, of what we have (hopefully) taught you about the advantages of modular programming—that is, taking the full amount of code needed to accomplish a task and splitting it into individual, task-oriented functions that can be universally and easily documented.

While being able to trust the individual robustness of your code is useful, sometimes, you need a lot of information about the "object" quantities you are dealing with — or, indeed, the flexibility to "see" those quantities in the process of programming.

Object Oriented Programming (OOP) allows for the blending of these two modes: interrogative programming and modularity. As a key note: Every **library** we have imported into our scripts thus far has been organized in an OOM, so it's not a fully unfamiliar structure to you. For example, when I import numpy as np, then call a sin function:

```
import numpy as np
y = np.sin(range(10))
```

What I'm doing is importing an OOM-class object (numpy), then dot-notation accessing a given function stored in that library (sin). To understand exactly how that works, we need to dissect that underlying structure. As a fun side note, though: Once we understand that structure, you'll not only be able to define your own classes in your code, but be able to write entire Python packages like Numpy on your own, and import them into your code for your own use.

8.1 Defining Classes

Objects, or ***Classes***, are useful for a variety of reasons; primarily we use them to easily organize groups of functions that act on a specific "object" that we define. There are two main components to a defined class: ***attributes*** and ***methods***. Attributes are properties which are created when an ***instance*** of a class is initialized (when you create a new instance of an certain class in your code).

Methods are functions which each instance of a class carries and can utilize. Accessing attributes and methods is done via dot notation, similarly to the way we utilize functions inherited from the libraries we imported. Essentially you can think of those libraries as large, complex classes.

Let's start with the basic syntax of classes. Before we can initialize an object of a certain class, have to define one first. Say we want to create a class called Planet which has certain attributes such as planet name, revolution period, and mass.

■ Example 8.1 Defining a Planet Class

```
class Planet(object):
  def __init__(self, planet_name, rev_period, mass):
      self.planet_name =planet_name
    self.rev_period =rev_period
    self.mass = mass
```

■

This looks a little funky, but its just the basic syntax for defining a class. The first function in a class is always the init function (which has a double underscore before and after the word init). The first argument of init is always "self" (the word self isn't special, it could be any word, so long as it was consistent in the rest of the class. But self is the overwhelmingly common choice. The position of self as being the first argument is what makes it special). Variables within classes can have various levels of scope (that is, how much of the code they are accessible to). If you look below to how you initialize an instance of a class, you'll see that we want to be able to access attributes set within the class. But variables defined within the init function are inside a function- which means their scope is local (only accessible within the function). Classes get around this by creating the "self" object, and you can map variables onto that object so they can be accessed as we show below. If I were to add a line to the end of the init function that set

```
galaxy_in = "milky_way"
```

I would not be able to type

```
print earth.galaxy_in
```

because the galaxy_in variable is local to that function, and was never mapped onto "self" which, once you initialize a variable as an instance of the class, becomes the name of that variable. That said, not every variable in your class has to have a "self." in front of it. If your class is running intermediary steps and those variable values are not really of use later on, it's fine to leave them local. But the user will only be able to access the stuff with a self in front of it within the code.

R Note: The reason we have the word 'object' in Example 6.1 is that we are actually creating a subclass of the superclass object. Don't worry too much about it, (we will get to subclasses in a moment), but just know that when defining a class in your code, you will probably use object in the class line.

Once we have created an init function, we can put other functions in our class as well. These are known as methods. For example, our Planet class could be updated to easily return the semimajor axis of the planet's orbit using Kepler's third law.

■ Example 8.2 Methods

```
class Planet(object):
  def __init__(self, planet_name, rev_period, mass):
    self.system_name = '2014-B178h'
    self.planet_name = planet_name
    self.rev_period =rev_period
    self.mass = mass

  def semimajoraxis(self):
    return self.rev_period**(2./3)
```

■

As you can probably infer, this method will be inherent to any instance of the Planet class you create and you can use it on any of them, just as any other method you include in the Planet class.

In addition to instance attributes (attributes set when an object is made) there can also be class attributes which are the same regardless of the instance. You simply set them without having the init function require input for it. In example 6.2, every planet you create will have the attribute of a system_name, and it will always be '2014-B178h'. Technically speaking, a better way to do this is to include system_name as the last argument of the init, but set it equal within the argument definition to '2014-B178h'. Then you would set

```
self.system_name = system_name
```

and things would work the same. The advantage here is the user still doesn't have to set anything, but they know the variable even exists, that they can access it, and that they can change it if they so desire.

8.1.1 Subclasses

When discussing the class initialization, we mentioned that you use the word object within the parenthesis to pull methods and attributes from the object superclass. We usually don't call the classes we create subclasses though, since the object class is basically inherently necessary to working with classes. Thus, the classes in your code are the highest level ones you create. It is possible to make subclasses of your own, which ***inherit*** all the attributes and methods of their parent class, while having some specialized methods and attributes of their own. For example, we can make a subclass of Planet called Dwarf:

■ Example 8.3 Subclasses

```
class Dwarf(Planet):
    def describe(self):
        return self.name + ':' + 'Mass-' + self.mass + 'Period-' + self.rev_period
```

■

Notice how instead of using object, we call Planet within the class call. Because Dwarf inherits everything from Planet (including the init function), we only have to worry about the special methods and attributes we want to apply only to the dwarf planets we define.

Classes and subclasses have lots of applications. For example, if you were designing a chess game, you might have a "board" class and a "piece" class, and then subclasses for the different kinds of pieces and the rules they follow, all while still having the common attributes of a piece.

8.2 Initializing an Instance of a Class

Now that we have created definitions for classes, how do we create objects of them? Quite simply, we can initialize a variable as an object (say, a planet), like this:

```
earth = Planet('Earth', 24, 5.9E24)
```

where 'Earth' is the planet_name, 24 is the revolution period, and 5.9E24 is the mass, as required in the init of our Planet class above. We can access those properties of any of our class objects (or see if they've changed), using dot notation:

```
print earth.mass
```

should print the Earth's mass. This allows us to easily use that value in calculations, for example, we can multiply earth.mass by jupiter.mass (assuming we initialized an object jupiter) without having to look those values up.

Now that we have the instance of the Planet class initialized as the variable earth, we can also execute the methods that we have defined for the given class. For example, using the method we defined above, we can call

```
print earth.semimajoraxis()
```

Now, the difference between our finding the semimajor axis and the mass is that we have defined the mass to be an attribute of the class, while it will go ahead and calculate the semimajor axis every time, and we then need the parenthesis to call the function. In this case, our method wasn't super useful- it doesn't take any arguments (the code doesn't need any additional information to calculate it), so we might've just set the semimajor axis as an attribute instead.

We can also change and adjust attributes of our initialized object at will. For example, say we were running a simulation during which the revolution period of the earth was changing with time, and thus needed to be continuously updated. We could set

```
earth.rev_period = new_value
```

at any time.

9. Glossary

Argument

An argument is essentially an input to a function. The term can be seen in the mathematics application, in which the sine function takes an argument, such as (x-5), in the form sin(x-5). In UNIX, the commands like mkdir, rm, and vim serve as built-in functions, and the syntax for applying arguments is by typing the function, a space, and then the argument. Within python, functions, both built in and user defined, are referenced much like in the math example: via parenthesis attached to the function call. (Ex: np.sqrt(15), my_function(name, dob, gender)).

CLI

Command-line Interface (terminal). A means of interacting with a computer system via successive lines of text in the form of commands.

Command

A word, phrase, or instruction that can be understood and interpreted by a computer system which then executes the command in question.

Conditional Statement

A statement defining a certain condition, using operators like greater than, less than, equals to, their opposites, or some combination. These statements enclose a block of code that runs only if the conditional statement is evaluated to be true.

Data Type

Refers to the different types of objects in python to which python places certain rules. For example, integers and floats can be added, but not indexed.

Directory

Within a UNIX system, a directory is analgous to "folders" on a PC or Mac. It's where all your files are stored.

Documentation

A special string added when defining functions which specifies the arguments of the function and their data types, accessible via help(function_name). Also, the more broadly used term for instructions on using a piece of code or codes.

Element
A single discrete object within an iterable set. For example, a single character in a string, or a single entry in a list or array, or an entry in a dictionary.

FITS
Flexible Image Transport Syste. A file format typical to astronomical images.

Flag
Also known as an option, a flag is a way to modify a UNIX command to alter the way it performs a task. Flags are entered between the command and the argument, with spaces in between both the command and the flag, and the flag and the argument. The typical syntax for a flag is a dash (-) followed by a letter or short combination thereof. (Ex: ls -l). To see what flags are available to use for any UNIX command, typing man(command) will have them listed.

For-loop
A for loop is a block of code that contains some iterative variable like "i" within it, with "i" cycling through different values defined in the initiation of the loop.

Function
A function is an operator that takes some inputs (or none) and, when called, performs some operations and outputs something (or multiple things). Functions can come from libraries, within python, or be user defined.

GUI
Graphical User Interface. A means of interacting with a computer system via graphical icons and visual indicators, through the direct manipulation of graphical elements (e.g., clicking, double clicking, and dragging)

Index
Given a list, array, or other sliceable data type (these are called iterable data types), every element is assigned an index based on its position from the leftmost element, starting with i = 0, and “reverse index” which starts on the rightmost element with -1, and gets more negative as you move left.

Indexing / Slicing
The process by which a subset or subsets of an iterable variable (list, array, string, etc) is extracted by specifying which indices whose values to retrieve.

Library
A large collection of functions that can be used in Python by importing the library. Calling functions from libraries usually requires the dot notation call of the library name (dot) function name. Libraries have defined names like numpy and matplotlib, but you can import them into your program as anything you like.

Loop
A block of code that is run multiple times, either due to iteration through some predefined range of values, or indefinitely so long as some condition is met.

Negative Index
Defined as the position of an element in an iterable data type, but with respect to the final element rather than the front. E.g., the last element is -1, the second to last is -2, etc.

Operating System
Software that manages the hardware and software resources for a computer. A vast majority of applications and programs require an OS be installed on a computer in

order to function. Common OS's include Windows, MacOS, and Ubuntu.

Path

The description of your location within a file system, indicated by the names of successively nested directories, usually separated by slashes (e.g., home/documents/project/) Prompt: In essence, the computer is "prompting" you to issue a command. This string of letters could be anything, but is usually set to be some variance on the current path (meaning the prompt will change as the directory does).

Prompt

A line of text appearing on the left of the the command line interface, usually containing some sort of path information. It can also be altered to read anything. You type in terminal next to be prompt.

Python

The programming language that is the subject of this textbook. Also refers to the specific distribution of the operating system-based software needed to interpret (i.e., "run") code that has been written in the python language syntax. The name comes from "Monty Python and the Holy Grail."

Root Directory

The root directory is the directory within which all other folders/directories and files are contained. In essence, one could start from any directory in a file system and move up in directories until the root directory is reached.

SSH

Secure Shell Host. A way of logging into a server of networked computers via the terminal of any external computer.

Syntax

The specific set of words, phrases, and commands that can be successfully interpreted and understood by a computer. For example, a computer can understand that upon receiving the typed command "ls" it should display all files in the current directory. Had the user typed "list files" the computer would have thrown an error as this phrase is not in its syntax.

Terminal

Also called a shell, the terminal is where the prompt and command line interface are contained. It is the program through which one can issue commands directly to the computer's OS through text commands.

UNIX

An operating system developed by Bell Labs upon which many other systems are currently built. UNIX is also the primary operating system on scientific machines like telescopes and supercomputers.

Variable

A variable is a user defined set of characters (could be a word, or number, or combination), which is assigned a value, array, etc. Variables allow us to implement large arrays and strings etc. in shorthand throughout our code.

While-loop

A while loop is a block of code that will run over and over so long as a conditional statement is still true after each run through of the block. This of course means that there must be something within the loop that will eventually force the condition to become false.