LA-UR-18-24496
Approved for public release; distribution is unlimited.

Title: An Introduction to UNIX, Emacs, Latex, and Python

Author(s): Mockler, Jack Henry

Intended for: Report

Issued: 2018-05-23

An Introduction to UNIX, Emacs, Latex, and Python

Jack H Mockler
XTD-NTA
May 21, 2018

Introduction

This document is designed to give a brief introduction to the UNIX Operating System, Emacs, Latex, and Python. Section 1 focuses on UNIX, and presents the basics of the file system, a variety of common commands, and a handful of slightly more in depth topics. Section 2 introduces Emacs and its somewhat unusual keyboard controls. In Section 3, the basics of creating a document in Latex are demonstrated. Finally, Section 4 provides the foundation needed to begin programming in Python.

1 UNIX

The UNIX Operating System controls and allocates the resources on a computer. Commands in UNIX are entered through a terminal window, which can generally be opened by right clicking on the desktop or through the Applications menu. The terminal window provides a command line, where commands can be entered and then executed by pressing the return key. The terminal window also outputs results from many commands.

1.1 File System

UNIX provides a file system based on files and directories. A file stores information. A directory can contain both files and other directories. Four important directory references are:

`~` The home directory, and the initial present working directory when a terminal is opened.

`.` The present working directory, or the directory the computer is currently looking at.

`..` The directory that contains the present working directory. Often referred to as the directory one level above the current one.

`/` The root directory, the top of the directory hierarchy.

This file system can be visualized as a directory tree:

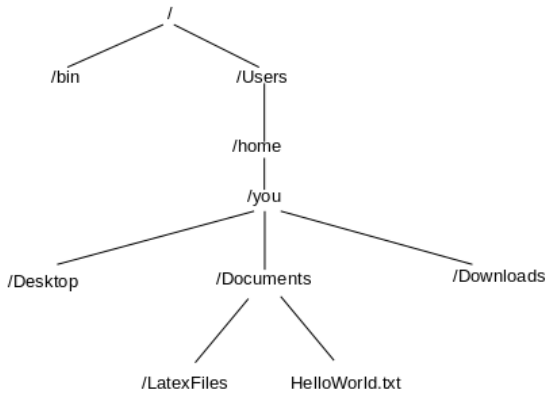

Figure 1: A visual of part of a UNIX directory tree structure.

Files are often referred to by their filename, e.g. HelloWorld.txt. However, a file's complete name is called a path, and includes all the directories from the root to the given file. For example, from Figure 1 the path to HelloWorld.txt would be:

/Users/home/you/Documents/HelloWorld.txt

For practical use, files can be referred to by just their filename if they are in the present working directory. Otherwise, either the full path to the file, or the path starting at the present working directory is required for the computer to be capable of finding it.

Some common commands for general navigation and file management include:

date - Returns the current time and date.

```
$ date
Tue Apr 17 12:50:34 MDT 2018
$
```

echo *string* - Echoes the arguments provided.

```
$ echo hello world
hello world
$
```

mkdir *name* - Creates a directory named *name*.

```
$ mkdir ~/Documents/ExampleFolder
$ mkdir ~/Documents/ExampleFolder/ExampleSubfolder
$
```

cd *path* - Changes the present working directory to the directory indicated by *path*.

```
$ cd ~/Documents/ExampleFolder
$
```

pwd - Returns the path to the present working directory.

```
$ pwd
/home/you/Documents/ExampleFolder
$
```

touch *name* - Creates an empty file named *name*.

```
$ touch exampleFile
$
```

ls - Lists the contents of the present working directory. The path to a different directory can be provided to list that directory instead.

```
$ ls
exampleFile  ExampleSubfolder
$
```

The **ls** command is a good example to introduce options. Options are usually indicated by a '-' followed by a letter, and modify the function of a command in some fashion. A common option for the **ls** command is **ls -l**, which lists the contents of a directory like **ls**, but with additional information about each item.

```
$ ls -l
total 3
-rw-------. 1 you 0 Apr 27 10:02 exampleFile
drwx--x---. 2 you 2 Apr 27 10:03 ExampleSubfolder
$
```

The information provided, from left to right, is user permissions, the number of links, the owner, the number of characters contained, the date last modified, and the name of the item. There will be more detail on permissions later in the section.

cp *file1 file2* - Copies *file1* into *file2*. Creates *file2*.

```
$ cp exampleFile copiedExample
$ ls
copiedExample  exampleFile  ExampleSubfolder
$
```

ln -s *original-file new-file* - Creates a new file linked to the original file. The -s option creates a "soft link", which links to the filename. A soft link will share the contents of the original file, as long as the name and location of the original file are maintained.

```
$ ln -s exampleFile linkedExample
$ ls -l
total 4
-rw-------. 1 jmockler jmockler  0 Apr 27 10:05 copiedExample
-rw-------. 1 jmockler jmockler  0 Apr 27 10:02 exampleFile
drwx--x---. 2 jmockler jmockler  2 Apr 27 10:03 ExampleSubfolder
lrwxrwxrwx. 1 jmockler jmockler 11 Apr 27 10:04 linkedExample -> exampleFile
$
```

mv *file dir* - Moves the given file to the given directory.

```
$ ls
copiedExample exampleFile ExampleSubfolder linkedExample
$ ls ./ExampleSubfolder
$ mv copiedExample ExampleSubfolder
$ ls
exampleFile  ExampleSubfolder linkedExample
$ ls ./ExampleSubfolder
copiedExample
$
```

wc *file* - Reports the number of lines, words, and characters in a file. **wc -l** reports only the number of lines.

```
$ wc exampleFile
```

```
 0 0 0 exampleFile
$
```

du - Reports the disk usage of the directory. Sizes are in kilobytes. If sizes are not in kilobytes, the **-k** option will force them to be. The **-s** option reports only the total size of the directory, ignoring individual components.

```
$ du
3 ./ExampleSubfolder
7 .
$ du -sk
7 .
$
```

rm *file* - Deletes the given file.

```
$ rm ./ExampleSubfolder/copiedExample
$ ls ./ExampleSubfolder/
$
```

file *file* - Returns the type of data in the given file.

```
$ file exampleFile
exampleFile: ASCII text, with no line terminators
$
```

rmdir *name* - Deletes the directory named *name*. This command only works if the given directory is empty.

```
$ rmdir ExampleSubfolder/
$ ls
exampleFile linkedExample
$
```

Going into detail about most specific options for most commands is out of the scope of this document. However, it is worth knowing about them and where more information can be found. The command:

man *command-name*

can be used to open an information page about a particular command in the terminal, which includes available options. Additionally, a Google search for a UNIX command will always yield lots of information, and can be especially useful if in-depth details regarding a particular option or scenario are required.

1.2 Useful Commands for File Manipulation

The commands and capabilities provided by UNIX are far too extensive to cover here in any comprehensive fashion. This section will, however, present a handful of tools that may be of particular interest.

To begin, it would be useful to have a text file for demonstration purposes. There is an easy way to create one, using a built-in editor called ed:

```
$ ed
a
This is a text file.
It has been created to demonstrate UNIX commands.
Because of this, it is somewhat oddly formatted.
Someone more creative would probably have written a poem or something.
This is the fifth line of the text file.
This is probably long enough, goodbye!
.
w sample
271
q
$
```

Above, *a* tells the editor to begin adding text, *.* tells it to stop, and *w* writes the text to a file, in this case *sample*. *q* quits the editor and brings back the command line. To edit a file with ed, opening it will only show the character count. To print the file, use the command *,n*. To print without line numbers use *,p*. The *s/old/new/p* command substitutes an old expression for a new one and prints the result. To focus on a particular line, just type the number of that line. In the example below, some edits are made to the last line of *sample*.

```
$ ed sample
271
,n
1 This is a text file.
2 It has been created to demonstrate UNIX commands.
3 Because of this, it is somewhat oddly formatted.
4 Someone more creative would probably have written a poem or something.
5 This is the fifth line of the text file.
6 This is probably long enough, goodbye!
6
This is probably long enough, goodbye!
s/probably/too/p
This is too long enough, goodbye!
```

```
s/ enough//p
This is too long, goodbye!
w sampleEdited
q
```

Now that a file exists, it might be useful to get a quick look at its contents. A simple way to do this is the **cat *file*** command.

```
$ cat sample
This is a text file.
It has been created to demonstrate UNIX commands.
Because of this, it is somewhat oddly formatted.
Someone more creative would probably have written a poem or something.
This is the fifth line of the text file.
This is probably long enough, goodbye!
$
```

cat prints the contents of a file into the terminal. For this sample file, and other short files, it works great. However, for a much longer file it becomes significantly less useful. In such a case, it can be more convenient to print just a small part of the file's contents. This can be done with the commands **head *file*** and **tail *file***. These print the first 10 and the last 10 lines of a file, respectively. They can also be modified to print a specific number of lines, as shown in the example below.

```
$ head -2 sample
This is a text file.
It has been created to demonstrate UNIX commands.

$ tail -2 sample
This is the fifth line of the text file.
This is probably long enough, goodbye!
$
```

There is also a **less *file*** command that opens the file in the terminal with forward and backward scrolling capability. Pressing q exits the document after opening it, bringing back the command line.

```
$ less sample
This is a text file.
It has been created to demonstrate UNIX commands.
Because of this, it is somewhat oddly formatted.
```

```
Someone more creative would probably have written a poem or something.
This is the fifth line of the text file.
This is probably long enough, goodbye!
(END)
```

It is also possible to print a file character by character, showing characters that are normally invisible such as tab and new-line. This functionality can be particularly useful when invisible characters need to be removed or found. The command for this is **od -c *file***. **od** without the **-c** option will print a file in octals.

```
$ od -c sample
0000000   T   h   i   s       i   s       a       t   e   x   t       f
0000020   i   l   e   .  \n   I   t       h   a   s       b   e   e   n
0000040       c   r   e   a   t   e   d       t   o       d   e   m   o
0000060   n   s   t   r   a   t   e       U   N   I   X       c   o   m
0000100   m   a   n   d   s   .  \n   B   e   c   a   u   s   e       o
0000120   f       t   h   i   s   ,       i   t       i   s       s   o
0000140   m   e   w   h   a   t       o   d   d   l   y       f   o   r
0000160   m   a   t   t   e   d   .  \n   S   o   m   e   o   n   e
0000200   m   o   r   e       c   r   e   a   t   i   v   e       w   o
0000220   u   l   d       p   r   o   b   a   b   l   y       h   a   v
0000240   e       w   r   i   t   t   e   n       a       p   o   e   m
0000260       o   r       s   o   m   e   t   h   i   n   g   .  \n   T
0000300   h   i   s       i   s       t   h   e       f   i   f   t   h
0000320       l   i   n   e       o   f       t   h   e       t   e   x
0000340   t       f   i   l   e   .  \n   T   h   i   s       i   s
0000360   p   r   o   b   a   b   l   y       l   o   n   g       e   n
0000400   o   u   g   h   ,       g   o   o   d   b   y   e   !  \n
0000417
$
```

The \n character represents a new line. Similarly, if there were any tab characters, they would be shown as \t.

In certain cases it might be useful to examine a file's contents organized in a particular fashion. The **sort *file*** command organizes (alphabetizes by default) the lines of a file before printing it.

```
$ sort sample
Because of this, it is somewhat oddly formatted.
It has been created to demonstrate UNIX commands.
Someone more creative would probably have written a poem or something.
This is a text file.
```

```
This is probably long enough, goodbye!
This is the fifth line of the text file.
$
```

Now suppose it is necessary to find all the lines of a file that contain a certain expression. Straightforward for a short file, but for longer files, the **grep** ***expression file*** command solves the problem.

```
$ grep This sample
This is a text file.
This is the fifth line of the text file.
This is probably long enough, goodbye!
$
```

As an alternative to the example above, **grep -v** ***expression file*** prints all lines that do not contain the provided expression.

It is also useful to be able to quickly compare two files. This can be accomplished using the **diff** ***file1 file2*** command. The example below first creates a new, similar text file and then compares it to the old sample.

```
$ ed
a
This is a text file.
It has been created to demonstrate a UNIX command.
Because of this, it is somewhat oddly formatted.
Someone more creative would probably have written a poem or something.
This is the fifth line of the text file.
This is probably long enough, goodbye!
.
w newSample
272
q

$ diff sample newSample
2c2
< It has been created to demonstrate UNIX commands.
---
> It has been created to demonstrate a UNIX command.
$
```

The output from **diff** above states that line 2 of the first file has changed to line 2 of the second file, and prints both lines. There is another file comparison

command, **cmp *file***. **cmp** is less useful for text files because it only reports the byte and line where a difference occurs, but it has the advantage of working on binary files.

```
$ cmp sample newSample
sample newSample differ: byte 57, line 2
$
```

There are special characters in UNIX called globs that increase flexibility when selecting or searching for things. The most common example of a glob is the asterisk (*) which represents any string. One practical use for this is finding files of one particular type, for example all of the text files in a directory:

```
$ ls *.txt
```

The above command lists any file in the present working directory that ends in .txt. Another common glob is the question mark (?), which represents any single character. This allows searches for strings of a particular length, for example listing all files with five character names in a directory:

```
$ ls ?????.*
```

One last glob is the square brackets ([ ]), which represent any of the characters contained within them. For example, to search for any file that begins with a number:

```
$ ls [0-9]*
```

Finally, it is possible to code and run for loops in the terminal. The general format is:

```
for var in list-of-words
do
    commands
done
```

The primary use of this type of for loop is to iterate over a list of files. For example, to print the contents of the present working directory using a for loop:

```
$for i in *
>do
>echo $i
>done
$
```

1.3 Processes

A process is simply a running program. Every time a command is issued from the terminal, it starts a process. The **ps** command provides a list of currently running processes.

```
$ ps
  PID TTY          TIME CMD
21029 pts/0    00:00:00 bash
24360 pts/0    00:00:00 ps
$
```

In the process list, PID is the process id, TTY is the terminal associated with the process, TIME is the processor time used for the process, and CMD is the command running.

Processes can be broken into foreground and background processes. A foreground process must be completed before additional commands can be entered. A background process runs while still allowing other work to be done. Processes run in the foreground unless told otherwise.

An **&** at the end of a command tells the process to run in the background. Alternatively, a foreground process that has already started can be suspended with **ctrl-Z**, and then moved to the background with the **bg** command.

```
$ sleep 1000 &
[1] 4690
$
```

The **jobs** command gives a list of current background jobs, their status, and an identifier. The identifier can be used with the **fg** ***%job-number*** command to move a background process into the foreground. Without arguments **fg** will move the most recent background process.

```
$ jobs
[1]+  Running                 sleep 1000 &
$
```

The **kill** ***%job-number*** command can be used to end a background process. **kill** can alternatively take a process id from **ps** as an argument.

```
$ kill %1
$ jobs
[1]+  Terminated              sleep 1000
$
```

kill can be called with the *-9* option (**kill** ***-9 process-id***), which forces the process to immediately die without any opportunity to ignore the command.

1.4 Pipes

This section contains examples using a directory and files that were set up as follows:

```
$ mkdir ~/Documents/PipeExamples
$ cd ~/Documents/PipeExamples
$ touch fileA fileD fileG
$ ls
fileA  fileD  fileG
$
```

It is useful to be able to chain commands using the output of previous commands as input. This allows for one program's output to continue being processed by another program. Commands can save their output to a file rather than printing it in the terminal, which allows for this functionality. For example to sort the contents of a directory:

```
$ ls ~/Documents/PipeExamples >filelist
$ sort -r filelist
fileG
fileD
fileA
$
```

The example above returns a reverse-alphabetically sorted list of the contents of the PipeExamples directory. The >tells the ls command to save its output as a file with the indicated name. However, creating temporary files to run a command is disorganized. Using a pipe is a way to avoid the creation of temporary files. A pipe connects the output of one command to the input of

another without using a temporary file, and is created using the | character. The previous example can be accomplished with a pipe as shown below.

```
$ ls ~/Documents/PipeExamples | sort -r
fileG
fileD
fileA
$
```

A second example of pipes is using **head** and **tail** to print lines from the middle of a file. Using the old *sample* file, a pipe can be used to print the 4th and 5th lines:

```
$ head -5 sample | tail -2
Someone more creative would probably have written a poem or something.
This is the fifth line of the text file.
$
```

Though these examples were somewhat trivial, they still demonstrate the efficiency of using pipes. The commands were less complex, and no unnecessary files were created. The advantages of using pipes become even more noticeable when strings of programs get longer.

1.5 Filters

A filter is a UNIX program that takes an input, transforms it in some simple way, and then returns some output. Several previously mentioned commands are filters - head, tail, grep, and sort, for example. More detail on two particular filters will be presented in this section.

tr *old-character replacement*

This filter replaces all instances of one character in an input with another. Using an example file that looks like:

```
$cat exampleText
there are a lot  of  invisible

characters
in this file
$od -c exampleText
0000000   t   h   e   r   e       a   r   e       a  \t  \t   l   o   t
```

```
0000020          o   f      \t   i   n   v   i   s   i   b   l   e  \n
0000040  \n  \t   c   h   a   r   a   c   t   e   r   s  \n   i   n
0000060   t   h   i   s       f   i   l   e  \t  \n  \t  \n
0000075
$
```

tr can be used to replace all tab characters with something different.

```
$ cat exampleText | tr '\t' '&'
there are a&&lot  of &invisible

&characters
in this file&
&
$ cat exampleText | tr '\t' '&' | od -c
0000000   t   h   e   r   e       a   r   e       a   &   &   l   o   t
0000020          o   f       &   i   n   v   i   s   i   b   l   e  \n
0000040  \n   &   c   h   a   r   a   c   t   e   r   s  \n   i   n
0000060   t   h   i   s       f   i   l   e   &  \n   &  \n
0000075
$
```

sed *'commands' files*

The **sed** filter applies **ed** commands to input from files. The most common command is *s* for substitution. For example:

```
$ sed 's@a@4@g' exampleText
there 4re 4 lot  of  invisible

ch4r4cters
in this file

$
```

In the example above the @ characters are simply separators, and can be replaced with any character that does not appear in the substitution strings. The final section of the command *@g* makes **sed** work globally, or on more than one occurence of the substitution per line. The command in this example would translate to something like "substitute the expression a with the expression 4 globally in exampleText."

1.6 Regular Expressions

Regular expressions are a powerful tool to increase the functionality of filters. They are patterns of characters that define searches. In this section some regular expressions will be demonstrated using the **grep** filter, but they are also useful for several others, such as **sed**.

Some commonly used regular expressions are:

^	Anchors search expression to the beginning of a line.
$	Anchors search expression to the end of a line.
[**character list**]	Matches any characters contained in the list.
.	Matches any single character.
*****	Matches as many of the previous expression as are found.
[^**character list**]	Matches any characters not contained in the list.
+	Matches one or more of the previous expression.

The following text file will be used for some examples:

```
$ed
a
A text file for regular expression examples.
John is at the beginning of this line.
On this line, John is in the middle instead.
The filename textFile.txt is hidden on this line.
Files that end in .txt are text files.
.
w grepSample
q
```

To find every line that begins with the name John:

```
$ grep ^John grepSample
John is at the beginning of this line.
$
```

To find every line that ends with line:

```
$ grep line.$ grepSample
John is at the beginning of this line.
The filename textFile.txt is hidden on this line.
$
```

To find every line that does not begin with a vowel:

```
$ grep ^[^aeiouAEIOU] grepSample
John is at the beginning of this line.
The filename textFile.txt is hidden on this line.
Files that end in .txt are text files.
$
```

To find any line that contains a .txt file with any string of letters as its name:

```
$ grep [a-zA-Z][a-zA-Z]*.txt grepSample
The filename textFile.txt is hidden on this line.
$
```

1.7 Permissions

Files and directories all have permissions associated with them. Recall that the **ls -l** command prints these permissions (among other things) for the contents of a directory. The permissions are represented by a string of 10 characters. For example:

```
drwx--x---
```

The first character is a '-' for a file, or a 'd' for a directory. The next three characters are permissions for the owner with 'r', 'w', and 'x' representing read, write , and execute permissions. The next three characters represent permissions for the group, and the final three permissions for all other users. A '-' indicates not having a particular permission.

Read permission grants the ability to view the contents of a file or directory.

Write permission allows modification of the contents of a file or directory.

Execute permission grants the ability to run a file as a program.

Permissions can be changed with the **chmod** command. There are two ways to use **chmod**. These will be demonstrated using the following file:

```
$ ls -l
total 2
-rw-------. 1 you 61 Apr 16 14:43 exampleText
```

The first method is the symbolic method, using the +, -, and = operators. u, g, and o represent the owner, group, and all user categories, respectively.

```
$ chmod o+r,g=rw,u+x exampleText
$ ls -l
total 2
-rwxrw-r--. 1 you 61 Apr 16 14:43 exampleText
$
```

The second method is using numbers to represent permissions. In this method, 4 is read, 2 is write, and 1 is execute. Each group's permissions are represented by the sum of the numbers for the permissions they should have.

```
$ chmod 640 exampleText
$ ls -l
total 2
-rw-r-----. 1 you 61 Apr 16 14:43 exampleText
$
```

1.8 Further Reading

For more information on UNIX, the book *The UNIX Programming Environment* by Kernighan and Pike is quite useful and easy to understand.

2 Emacs

Emacs is a text editor. It is free, and is programmable using Lisp. For the purposes of this document Emacs is treated as a tool for creating text documents for Latex and Python, but it certainly has broader uses that could be interesting to investigate further.

To create a text document using Emacs, enter **emacs *name.tex*** or **emacs *name.py*** into the command line. This will create a new file in the present working directory and open it in Emacs. The .tex extension indicates that it is intended to be a Latex document, which causes Emacs to color code the document and later allows it to be properly converted to a formatted pdf. The .py extension allows a file to be read as a Python script.

2.1 Emacs Keyboard Input

Emacs uses unique keyboard inputs that differ from a lot of commonly used software. A list and description of some of the more common and useful Emacs commands is presented here.

C- refers to holding down the ctrl key while pressing the next key in a command.

M- refers to either tapping the escape key before or holding down the alt key while pressing the next key in a command.

C-x C-s saves the file.

C-g cancels the current command.

C-x C-c quits Emacs.

C-_ undoes the previous action.

C-spc sets a mark at the current cursor location for selection purposes.

C-w cuts selected text.

M-w copies selected text.

C-y pastes (or yanks) text that was copied or cut.

C-s opens a search function to search for an expression.

C-r same as **C-s** but searches in reverse.

C-f moves the cursor forward one space.

C-b moves the cursor backward one space.

C-n moves the cursor one line down, to the next line.

C-p moves the cursor one line up, to the previous line.

C-e moves the cursor to the end of the current line.

C-a moves the cursor to the beginning of the current line.

M-> moves the cursor to the end of the file.

M-< moves the cursor to the beginning of the file.

C-h opens the Emacs help page.

There is a built-in tutorial that can be accessed from the bottom section of a new Emacs window or from the help menu. It is designed to familiarize new users with basic commands, and is a good resource to start becoming comfortable with Emacs. For additional keyboard commands there is a GNU Emacs reference card at https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf.

3 Latex

Latex is a markup language which uses commands mixed with text to control the eventual format of a document.

Once a Latex document is created in Emacs, it can be converted to a formatted pdf with the **pdflatex *file*** command in the terminal.

3.1 General Document Structure

A comment in Latex is indicated with a %, and a command with a \

The beginning of a Latex document should declare the document class with:

```
\documentclass{article}
```

A document class sets default formatting and layout options. There are many document classes other than article, such as book, report, or proc. Any packages that will be used when constructing a document should also be declared at the start with:

```
\usepackage{graphicx}
```

Commonly used packages include:

graphicx - Provides commands for adding graphics to a document.

amsmath - Provides additional math environment functionality.

float - Provides additional options for figure placement.

The body of a document must be enclosed in a document environment:

```
\begin{document}
```

```
Main body of the document
\end{document}
```

Sections with headers can be created using:

```
\section{heading of the section}
\subsection{heading of the subsection}
\subsubsection{heading of the subsubsection}
```

Bulleted lists can be added to a document with the itemize environment:

```
\begin{itemize}
\item first thing
\item second thing
\end{itemize}
```

- first thing
- second thing

Numbered lists are added similarly with the enumerate environment:

```
\begin{enumerate}
\item first thing
\item second thing
\end{enumerate}
```

1. first thing
2. second thing

To create a descriptive list, the description environment is used:

```
\begin{description}
\item[label 1] Text describing label 1
\item[label 2] Text describing label 2
\end{description}
```

label 1 Text describing label 1

label 2 Text describing label 2

Finally, to show plain text of commands as in this section, use the verbatim environment.

3.2 Paragraphs

By default Latex indents paragraphs, and does not put spaces between them. There are several ways to manipulate this.

A new paragraph can be defined by either a blank line or the *\par* command. These new paragraphs will take the default formatting for paragraphs in the document, as described above. This formatting, as well things like the depth of indentation, can vary depending on the document class.

To change the default indentation or spacing between paragraphs for a document, use the following commands, generally just before the beginning of the document environment:

```
\setlength{\parindent}{4em}
\setlength{\parskip}{1em}
```

The first command sets the depth of indentation for the document, and the second the amount of space between paragraphs. The unit 'em' is the length of one 'm'. There are at least a few other options to choose from if needed. To remove indentation entirely the indent length can be set to *0pt*.

It is also possible to control the indentation and spacing of individual paragraphs. Commands and options for this can be found in the Sharelatex documentation.

3.3 Figures

Many documents require figures, and adding a figure using Latex is fairly straightforward. The basic structure for adding a figure to a document is as follows:

```
1 \begin{figure}[H]
2 \centering
3 \includegraphics[height=5cm, width=7cm]{TutorialFigure}
4 \caption{This is a graph of $y=x^2$}
5 \label{ExFigure}
6 \end{figure}
```

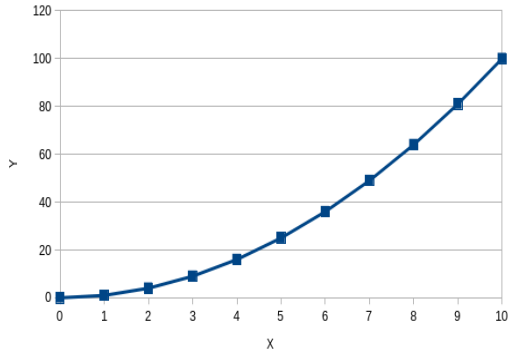

Figure 2: This is a graph of $y = x^2$

This centers a 5cm tall, 7cm wide figure from the specified file at the current location in the document. Line 1 opens the environment and specifies the location of the figure with *H*. *H* tells Latex to put the figure at the location of the code, and to override Latex's normal attempts to find a better place for it. It is only available if the float package is being used. Line 2 centers the figure. Line 3 tells Latex what file the figure is in, and specifies the width and height. The dimensions can also be input in other formats, such as *em*. *includegraphics* is only available with the graphicx package. Lines 4 and 5 caption the figure and assign it a label. The caption describes the figure, and will automatically number it. The label is just a reference that can be called later in the document to recall the appropriate figure number. For example, to call the figure above like this: Figure 2, use

```
Figure \ref{ExFigure}
```

3.4 Tables

Latex can also be used to format tables. The basic structure for creating a table is:

```
1  \begin{center}
2  \begin{tabular}{|l|l|}\hline
3  first  & second \\
4  column & column \\ \hline
5  item11 & item12 \\
6  item21 & item22 \\
7  item31 & item32 \\ \hline
```

```
\end{tabular}
\end{center}
```

first column	second column
item11	item12
item21	item22
item31	item32

This creates a centered table with two left-aligned columns separated by vertical lines. Line 1 places the table in a centered environment. Line 2 opens the tabular environment and defines the table as having two left aligned columns separated by vertical lines. The vertical lines can be removed individually by removing the | characters from the brackets. \hline creates a horizontal line. In lines 3 through 7, the & separates the columns of the table. The text on either side is placed into the respective cells. \\ indicates the start of a new line.

3.5 Special Characters

There are ten special characters in Latex that cannot be typed normally as text. This is because they have special meanings used elsewhere in the language.

These characters are &, %, $, #, _, {, }, ~, ^, \.

In order to create these characters as text the first seven must be preceded by a \. The final three have special macros.

```
\&
\%
\$
\#
\_
\{
\}
\textasciitilde
\textasciicircum
\textbackslash
```

Additionally, if the >and <characters appear as upside down punctuation marks, they must also be generated with macros.

```
\textgreater  = >
\textless  = <
```

3.6 Math

One of the main advantages of Latex is its ability to display math coherently.

To display an equation in text as $A_t = A_0e^{-kt}$ is shown here, simply bracket the equation with $ on either side:

```
beginning of sentence $A_t = A_0e^{-kt}$ end of sentence.
```

To display math in a standalone fashion the equation environment is used. Adding an * will suppress the automatic numbering functionality. Note that the * will not work unless the amsmath package is being used. Also note that there are several other ways to display in this fashion, this is just the easiest to remember.

```
\begin{equation}
A_t = A_0e^{-kt}
\end{equation}
```

$$A_t = A_0e^{-kt} \tag{1}$$

```
\begin{equation*}
A_t = A_0e^{-kt}
\end{equation*}
```

$$A_t = A_0e^{-kt}$$

There are other environments to display multiple equations, and to align them in particular ways. They can be easily found along with descriptions and examples in the Sharelatex documentation.

Equations can be referenced with labels much like figures. Reference labels can be added to an equation like so:

```
\begin{equation}
A_t = A_0e^{-kt}\label{RateLaw}
\end{equation}
```

The equation can then be referenced in text as shown here: (1) using its label as follows:

```
\eqref{RateLaw}
```

3.7 Hyperlinks

All the references in a document can be hyperlinked by importing the *hyperref* package. It must be the last package imported. It can also be used to add web links. The style of links can be changed as shown below:

```
\usepackage{hyperref}
\hypersetup{
colorlinks=true,
linkcolor=blue,
urlcolor=cyan,
}
```

colorlinks=true makes the links colored, linkcolor sets the color of the internal links in the document (equations, figures, etc), and urlcolor sets the color of web links.

Internal references are automatically linked when using *hyperref*. To link a url use \href{text}{url}. The example below hyperlinks the Sharelatex documentation.

```
\href{www.sharelatex.com/learn}{www.sharelatex.com/learn}
```

3.8 Further Reading

For information beyond what is included in this section, the documentation for Sharelatex and *The Not So Short Introduction to Latex* 2_e by Oetiker, Partl, Hyna, and Schlegl are both excellent resources.

For papers that require references, Latex can manage bibliographies and reference links using the Bibtex system. Information on how to use Bibtex can be found on the Bibtex website, http://www.bibtex.org/ or in the Sharelatex documentation at https://www.sharelatex.com/learn/Bibliography_management_with_bibtex.

4 Python

Python is an object oriented programming language commonly used for data analysis and scientific computing. It is designed to be easily readable, using white space for syntax and less punctuation than many other languages.

Python can be run in a terminal window by simply entering **python** as a command. This is useful for testing purposes, accessing help pages, and brief coding uses, but for any more extensive coding needs, it is better to work in a text file. A Python script written in a text file can be run with the **python *file*** command in the terminal.

4.1 General Concepts

Functions and their syntax will be discussed in more detail in a future section, but there are a few common, built-in functions presented here so they can be understood in examples:

print() - Prints its argument as output.

```
>>>print("This is a sentence.")
This is a sentence.
```

range() - Creates a list of values from 0 to its argument.

```
>>>range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

len() - Returns the length of its argument.

```
>>>len(range(10))
10
```

input() - Prompts the user for text input.

```
>>>x = input("Enter a sentence: ")
Enter a sentence: "This is a sentence."
>>>print(x)
This is a sentence.
```

help() - opens a help page for the argument provided.

To find more information on a particular function, there is extensive online Python documentation that can be found through a Google search or on docs.python.org.

Python syntax is largely based on indentation. Any block of code, such as a loop, a function, or a class is defined by indentation. For example, there is a pair of nested if statements below. If statements will be discussed in more detail later, for now just realize that an if statement contains its own block of code.

```
if x < 10:
    if x%2 == 0:
        print(x)
    else:
        print("x is not divisible by 2.")
```

Here the two print statements are part of the inner if and else blocks, and are therefore indented twice. The outer if block contains the inner if and else statements, which are indented once. It is important to keep track of indentation carefully, as improper indentation will not necessarily cause an exception. Notice also that the opening line of a block of code ends with a ':'.

Also in the example above, the '==' operator is used. This leads to an explanation of assignment vs. comparison. In Python a single '=' is the assignment operator. It is used to assign a value to a variable, such as in the statement $x = 5$. To compare two values, the '==', or comparison operator must be used. The comparison operator compares the two values on the left and right, and returns True if they are the same, or False if they are different.

```
>>> x = 5
>>> x == 5
True
```

One last important concept in Python coding is comments. Everything in a Python script is read as code unless specifically noted as a comment by the '#' character. When writing a script, it is good practice to use comments to describe the function of various blocks of code, as well as the script overall. This allows others to pick up the code more easily, and can be useful as a refresher for the original programmer.

4.2 Types

Information stored in a variable has a type, indicating what kind of information it is.

Particularly important types to be aware of include:

int - an integer.

```
2
```

float - a decimal.

```
2.0
```

string - a sequence of characters.

```
"abc def ghi"
```

list - a sequence of values.

```
[1,2,3,4,5]
```

tuple - an immutable list.

```
(1,2,3,4,5)
```

dictionary - an unordered set of key:value pairs.

```
{'Column1': [1,2,3,4,5], 'Column2': [6,7,8,9,10]}
```

It is important to understand types and keep track of them when performing operations on variables, since the type of a variable limits its interactions. For example, using the '+' operator, numbers can be added to numbers, and strings can be added to strings, but numbers cannot be added to strings.

4.3 Modules

A huge amount of additional functionality in Python requires importing modules that contain functions and classes. Modules are imported by issuing an import statement at the start of code:

```
import math
```

Importing a module allows its functions and classes to be used throughout the code. It is possible to import only specific parts of a module. For example:

```
from math import sqrt
```

An important difference between these two methods is how the functions are called. Using the **sqrt()** function from above:

In the first example the function would be called as **math.sqrt()** in the code.

In the second example, it could be called simply as **sqrt()**. In this case, the imported **sqrt()** function would conflict with any other function called **sqrt()** in the script code.

One more useful feature is the ability to import modules with a user-defined name, which would be used instead of the module name to call functions. For example, if **math.sqrt()** is too much to type regularly:

```
import math as m
```

would allow it to be typed as **m.sqrt()**. It would work similarly for any other function in the math module.

Some other specific modules will be discussed in more detail in the Specific Topics section.

4.4 Functions

A function is an executable object that can take arguments and return values. Functions are used to organize code, streamline repeated use of the same code, and increase readability. It is generally good practice to make use of functions whenever possible. An example of a simple function that returns the sum and product of two input numbers is below.

```
def SumXY(x, y):
    summ = x + y
    prod = x * y
    return summ, prod
```

def defines a function name and the arguments it will take. A function can have any name that does not conflict with a different function or a special name. The arguments a function will expect are specified in the parenthesis after the name, and the line should end with a ':'. Many, but not all, functions return some value or values upon completion. These must be assigned in the function, and are noted by the **return** statement at the end of the function.

To call a function after it has been defined, simply type its name and enter any arguments it requires. If a function returns values, they should probably be assigned to variables so they are saved. In the example below the function above is called on the integers 2 and 3.

```
>>>summ, prod = SumXY(2,3)
>>>print(summ, prod)
(5, 6)
```

4.5 Flow Control

Flow control statements are loops or conditionals that specify how or when to execute a particular block of code. This section presents basic applications of standard flow control statements.

if - The if statement sets a condition that must be met to execute certain code.

```
if x < 10:
    x = x + 1
```

In the example above, the program will check the value of the variable x, and only execute the code inside the if block if the x <10 condition is met.

for - The for statement is used to iterate over items in any iterable object, such as a list.

```
for i in range(5):
    print(i)
0
1
```

```
2
3
4
```

while - The while statement continues executing a block of code until its condition is no longer true.

```
x = 100
while x > 10:
    x = x / 2
    print(x)
50
25
12
6
```

break and **continue** allow more control over exiting loops. In the examples below a list is used, which is defined as:

```
ExampleList = [0, 1, 2, 3, 4, 5]
```

break - The break statement immediately exits a loop.

```
for i in range(len(ExampleList)):
    if ExampleList[i] == 2:
        break
    print(ExampleList[i])

0
1
```

continue - The continue statement skips the remainder of the current iteration of a loop and moves on to the next iteration.

```
for i in range(len(ExampleList)):
    if ExampleList[i] == 2:
        continue
    print(ExampleList[i])

0
1
```

```
3
4
5
```

Specific exceptions can be caught using **try** and **except**. A program will first attempt to run whatever is inside the **try** block. If an exception occurs, the program will jump to the except statement and check if the exception matches the type specified. If it does, the program will execute the except block.

```
try:
    x = float(input("Enter a number: "))
except ValueError:
    print("That wasn't a number!")

Enter a number: "number"
That wasn't a number!
```

4.6 Classes

A Class is an object with local variables and functions, called attributes and methods, associated with it. Classes are defined using the **class** keyword. Variables contained within the __init__ method of a class are attributes, and are associated with individual objects of the class, assigned upon creation.

```
class Pet:
    def __init__(self):
        self.species = "Dog"

    def change_species(self, newSpecies):
        self.species = newSpecies
```

The Pet class has a species attribute, and a method that allows its species to be changed. Any newly defined Pet will have a species of Dog until it is changed. Attributes and methods of a class are called upon with the dot (.) operator. An example of using the Pet class is shown below.

```
>>>Spot = Pet()
>>>print(Spot.species)
Dog
>>>Spot.change_species("Cat")
>>>print(Spot.species)
Cat
```

Classes can be intiated with input when new class objects are created. For example, the Pet class can be rewritten to allow a newly created object to have its species and a new attribute, age, defined for it at inception.

```
class Pet:
    def __init__(self, species, age):
        self.species = species
        self.age = age

    def change_species(self, newSpecies):
        self.species = newSpecies
```

Now when an object is made as a Pet class, it takes two arguments to define the new Pet's attributes, species and age.

```
>>>Spot = Pet("Dog", 5)
>>>print(Spot.species, Spot.age)
('Dog', 5)
```

4.7 Interacting with the System

Many Python programs must be able to take command line arguments such as options or inputs when they are run. The sys module, described in a bit more detail later, provides access to a list of the command line arguments that are provided when a program is run.

A second useful module is the argparse module, which streamlines handling different types of arguments and helps create useful error messages.

There are two kinds of arguments: optional and positional. Optional arguments are indicated by a '-' or '- -'. Positional arguments are just names, such as a list of files.

The argparse module allows for the easy addition of expected arguments, and immediately classifies them. It also automatically creates a help message, which is updated by the individual help statements provided for new arguments.

Below is an example of a program making basic use of argparse.

```
1  import argparse
2
3  def mult(x):
4          prod = 1
```

```
5           for i in range(len(x)):
6                   prod = prod*x[i]
7           return prod
8
9   parser = argparse.ArgumentParser(description='Multiply
10  integers.')
11
12  parser.add_argument('integers', metavar='N', type=int,
13  nargs='+', help='an integer for the accumulator')
14
15  parser.add_argument('--mult', dest='accumulate',
16  action='store_const', const=mult, default=max,
17  help='sum the integers (default is max)')
18
19  args = parser.parse_args()
20  print(args.accumulate(args.integers))
```

As usual, the program begins by importing the relevant modules, in this case argparse. Lines 3-7 define a simple function that takes a list of integers, x, and returns their product.

In line 9, the argument parser is initialized, and is provided a brief description of the program's purpose. This must be done before any arguments are added.

Lines 12-17 add the relevant arguments. The first argument added is the list of integers to be multiplied. The arguments provided here are:

'integers'	The name of the argument.
metavar='N'	A pseudonym for the argument. It will appear in help text, for example.
type=int	The type that the argument should be parsed as.
nargs='+'	The number of arguments expected. '+' indicates any number more than 0.
help='a list of integers'	The message that should appear inside the help text for the given argument.

The second argument added is the optional argument to tell the program to run the mult function. The arguments for this addition are:

'- -mult'	The name of the argument. The '- -' indicates it is an optional argument.

dest='accumulate'	The name of the attribute added to the object returned by **parse_args()**.
action='store_const'	Defines the action to be taken with the associated command line arguments. 'store_const' stores the value specified by the const keyword. The default action is 'store', which stores the command line argument's value.
const=mult	Defines the value to be stored by the action.
default=max	Defines the value to be used if the optional argument is not entered in the command line. Here the default is the built in max function, which returns the maximum of the integers.
help	Same as in the first argument, indicates the message to be displayed when the help option is called.

Line 19 tells the program to run the argument parser on any arguments entered into the command line when the program is run.

Line 20 simply prints the results of the program.

Here are some examples of successful and unsuccessful attempts to run the multiplier.py program:

```
$ python multiplier.py 1 2 3 --mult
6

$python multiplier.py --mult
usage: multiplier.py [-h] [--mult] N [N ...]
multiplier.py: error: too few arguments

$python multiplier.py --help
usage: multiplier.py [-h] [--mult] N [N ...]

Multiply integers.

positional arguments:
  N           a list of integers

optional arguments:
  -h, --help  show this help message and exit
  --mult      multiply the integers provided. Example command: $python
              multiplier.py 1 2 3 --mult
```

```
$python multiplier.py 1 2 3
3
$
```

4.8 Specific Topics

4.8.1 String Formatting

Strings can be concatenated with other strings in Python, but adding other values (without first changing them to a string) can require some special formatting. The **string.format()** function provides flexible options for such formatting.

Arguments of **format()** can be given as a list or assigned to variables. These arguments are called in the string using {}, and either the index in the list or the variable name associated with a particular argument.

```
>>> "I can count! {0}, {1}, {2}!".format(1, 2, 3)
'I can count! 1, 2, 3!'

>>> "I can count! {first}, {second}, {third}!".format(first = 1, second =  2, third = 3)
'I can count! 1, 2, 3!'
```

The arguments from format can be formatted in specific ways, such as having a particular number of decimal places, using scientific notation, or being padded with zeroes.

```
>>> x = 3.4589798645679456
>>> y = 4.58976894576
>>> "x with 3 floating points is: {0:.3f}. y with 2 floating points is {1:.2f}".format(x,y)
'x with 3 floating points is: 3.459. y with 2 floating points is 4.59'

>>> z = 123456789
>>> "z in scientific notation is: {0:.2E}".format(z)
'z in scientific notation is: 1.23E+08'

>>> c = 22
>>> "c with 3 figures is: {0:03d}".format(c)
'c with 3 figures is: 022'
```

Specific attributes of arguments that have them can also be formatted. For example, using an object of the Pet class from the Classes section:

```
>>> Spot = Pet("Dog", 5)
>>> "Spot is a {0.species}, and is {0.age} years old.".format(Spot)
'Spot is a Dog, and is 5 years old.'
```

4.8.2 Array Slicing

Suppose there is an array defined as:

```
ExampleList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

It is possible to take various slices of the whole array using the ':' with indices. The basic format is *ExampleList[start:end]*. Here are some examples:

```
# The array from the 3rd index to the end
>>>ExampleList[3:]
[3, 4, 5, 6, 7, 8, 9, 10]

# The array from the 4th index to the 8th index. Note that the end index is non-inclusive.
>>>ExampleList[4:8]
[4, 5, 6, 7]

# The array from the beginning to the 7th index.
>>>ExampleList[:7]
[0, 1, 2, 3, 4, 5, 6]
```

The numpy module includes lots of functionality for working with arrays. When working with a 2-dimensional numpy array, the syntax for slicing is similar, but must account for rows and columns.

```
>>>import numpy as np

# Defines and prints a 2-d array
>>>ExampleArray = np.array([[1,2,3,4,5],[5,6,7,8,9,10]])
>>>ExampleArray
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])

# Prints the 2nd row of the array
>>>ExampleArray[1,:]
array([ 6,  7,  8,  9, 10])
```

```
# Prints the 4th column of the array
>>>ExampleArray[:,3]
array([4, 9])

# Prints the value of the array at the 2nd row, 5th column.
>>>ExampleArray[1,4]
10
```

4.8.3 re Module

The re module offers regular expression operations in Python. It recognizes the same regular expressions described in the UNIX regular expressions section, as well as a multitude of others that can be found in documentation. The module also provides several options for search functions. Here are a few:

re.search(*pattern, string*) - Looks for the first location in the string that matches the pattern and returns a match object. If there are no matches returns None.

re.match(*pattern, string*) - Checks if the pattern matches the beginning of the string, and returns a match object if it does. Otherwise returns None.

re.findall(*pattern, string*) - Returns all matches to the pattern in the string as a list.

```
>>> re.search('this', 'Does this line contain this?')
<_sre.SRE_Match object at 0x7f0c1c7027e8>

>>> re.match('this', 'Does this line contain this?')

>>> re.findall('this', 'Does this line contain this?')
['this', 'this']
```

A match object has a boolean value of True, and contains a list of subgroups of the match.

match.group() - Returns subgroups of a match. **match.group(0)** returns the entire match, while **math.group(1)**, etc, return the subgroups in order.

In the following example, \w+ matches any single word. '()' indicate the start and end of a group.

```
>>> m = re.search('(\w+) (\w+)', 'Word1 Word2 Word3')
```

```
>>> m.group(0)
'Word1 Word2'
>>> m.group(1,2)
('Word1', 'Word2')
```

There are a few operators to be aware of when inputting regular expressions using re:

? Makes the preceding expression optional for a match.

\ Reads a special character as plain text.

| Matches either the left or the right side expression.

4.8.4 glob Module

The glob module provides the **glob.glob** function which allows Python to find pathnames similarly to the UNIX command line. It accepts UNIX globs.

To demonstrate, here is a directory with some files in it:

```
$ mkdir ~/Documents/GlobExample
$ cd ~/Documents/GlobExample
$ touch 1.txt 10.png 2.txt 20.png random.txt 2.png
$ ls
10.png 1.txt  20.png 2.png 2.txt  random.txt
$
```

Now glob can be used to find various file combinations:

```
>>> import glob

>>> glob.glob("*.png")
['2.png', '10.png', '20.png']

>>> glob.glob("?.txt")
['1.txt', '2.txt']

>>> glob.glob("?.*")
['1.txt', '2.png', '2.txt']
```

4.8.5 sys Module

The sys module provides interfacing options with the interpreter. Its primary use is to provide a list of command line arguments that a program is given. This list is called **sys.argv**. The first item in the list, **sys.argv[0]** is always the name of the script that is being run. Items afterwards are the arguments provided.

The example code below shows how to use the **sys.argv** list of the sys module to read one file, reverse its lines, and print them in a new file.

```
# Imports the sys module
import sys

# Opens the first file in read mode, and copies its lines to a
# variable. Closes the file afterwards.
input = open(sys.argv[1], "r")
lines = input.readlines()
input.close()

# Reverse the list of lines.
lines.reverse()

# Opens the second file in write mode and copies the reversed
# lines into it one at a time.
output = open(sys.argv[2], "w")
for l in lines:
    # lines.strip() removes any whitespace from the start
    # and end of each line.
    print >> output, line.strip()
output.close()
```

4.9 Further Reading

Many Python modules have their own documentation that can be found online. There are a few modules not mentioned here that it would be particularly worth looking in to:

Sympy - A module for symoblic math operations.

http://www.sympy.org/en/index.html

Pandas - Contains the Dataframe object and input and output functionality for csv files.

https://pandas.pydata.org/

MatPlotLib - Provides plotting functionality.

https://matplotlib.org/

References

The UNIX Programming Environment by Kernighan and Pike

The Not So Short Introduction to Latex 2_e by Oetiker, Partl, Hyna, and Schlegl

Sharelatex documentation - www.sharelatex.com/learn

Python documentation - docs.python.org