GUMPy

Introduction

This is mostly a quick introduction to the Unix operating system for the MCB185 bioinformatics programming class at UC Davis. Most people reading this will be using Ubuntu or some other flavor of Linux. Is there a difference between Unix and Linux? Practically, no; philosophically, sort-of; politically, yes. No time to discuss. Onward.

In addition to Unix, the other core technologies we need are Python, GitHub, and Markdown. Rearranged, that spells GUMPy.

Terminal, Command Line, and Shell

Your interface to Unix will be through a **shell** program using the **command line interface** within a **terminal** application. There are several types of shells and terminals, but the details of these are unnecessary for us. The command line is where you type instructions for what you want the computer to do. Some of these statements are "do this now" while others are "let's get ready to do this." It may seem silly in this high-tech age to type your commands when you could simply point and click or maybe even use voice activation or gesturing. It is easier to automate typing though. So when it comes time to work with thousands of files, it will be much easier through a command line interface.

Terminal Basics

Launch the terminal application (the name of this will differ from one operating system to another and even within a particular OS you will have several options). Run the `date` command by typing in the terminal and ending with the **return** key.

```
date
```

Congratulations, you just made your first Unix statement. You type words on the command line (in this case just one) and then hit return to execute the command. `date` is but one of thousands of Unix commands you have at your fingertips (literally). Like most Unix commands, `date` does more than simply output the current date in the format you just witnessed. You can choose any number of formats and even set the internal clock to a specific time. Let's explore this a tiny bit. Commands can take **arguments**. Let's tell the `date` program that we want the

date to be formatted as year-month-day and with hours-minutes-seconds also. The syntax below will seem arcane, but the various abbreviations should be obvious. Type the command and observe the output.

```
date "+%Y-%m-%d %H:%M:%S"
```

Manual Pages

If you want to learn more information about what `date` can do, you can either look online or use the Unix built-in manual pages. For a quick refresher on a command, the manual pages are often easiest. But if you have no idea what the command does, than you might want to look for assistance online. To read the manual pages for `date` you use the `man` command.

```
man date
```

You can page through this with the space bar and exit with `q`. In the above statement, `man` was the command and `date` was the argument. The program that let you page through the documentation was another Unix command that you didn't intentionally invoke, it just happens automatically when you execute the `man` command.

Injury Prevention

Typing is bad for your health. Seriously, if you type all day, you will end up with a repetitive stress injury. Don't type for hours at a time. Make sure you schedule breaks. Unix has several ways to save your fingers. Let's go back and run the `date` program again. Instead of typing it in, use the **up arrow** on your keyboard to go backwards through your command history. If you scroll back too far, you can use the **down arrow** to move forward through your history (but not into the future, Unix isn't that smart).

```
date "+%Y-%m-%d %H:%M:%S"
```

You can use the **left arrow** and **right arrow** to position the cursor so you can edit the text on the command line.

To see your entire history of commands in this session, use the `history` command.

```
history
```

Probably the most important finger saver in Unix is **tab completion**. When you hit the tab key, the shell completes the rest of the word for you if it can guess what you want next. For example,

instead of typing out `history` you can instead type `his` followed by the tab key, the rest of the word will be completed. If you use something less specific, you can hit the tab key a second time and Unix will show you the various legal words. Try typing `h` and then the tab key twice. Those are all the commands that begin with the letter h. We will use tab completion constantly. Not only does it save you key presses and time, it also ensures that your spelling is correct. Try misspelling the `history` command to observe the error it reports.

```
historie
```

Variables

The shell defines various variables which are called **shell variables** or **environment variables**. For example, the `USER` variable contains your user name. You can examine the contents of a variable with the `printenv` command.

```
printenv USER
```

We won't use variables much, but it's important to know they exist because some programs use them for configuration. If you want to see all your environment variables, you can use the `printenv` command without any arguments. Don't worry if you find the topic of environment variables is very confusing at this time. We will revisit the topic later when it matters more.

```
printenv
```

Files & Directories

Unix is mostly a text environment. You will be reading and writing a lot of text files. To do this you will use a text editor. There are many, many text editors, and some people are insanely passionate about one or the other. We don't have time for that. When you want to edit a file in the terminal, we will use **nano**. More often, we will be programming using a text editor. There are many good ones available for free. On Windows, try Notepad++ or Atom. On Mac try BBEdit or Atom. On Linux try gedit or Atom.

In Unix, pretty much everything is a file. You need to be able to create, edit, and delete files and directories to do anything. You can do some of these things from the graphical desktop interface, but resist this. The more comfortable you are using the command line for everything, the better off you will be.

The **file system** is where your files are stored. This could be a hard disk or other medium, and there could be one or more. To see how much free space you have on your file system, use the

`df` (disk free) command with the `-h` option to make it more human-readable (try it both ways).

```
df
df -h
```

Another useful command is `du -h` (disk usage) which shows how much space each of your files and directories uses.

Creating and Viewing Files

(For the this exercise and others that follow, it may be useful to have your graphical desktop displaying the contents of your home directory. That way you can see that typing and clicking are related.)

There are a number of ways to create a file. The `touch` command will create a file or change its modification time (Unix records when the last time a file was edited) if it already exists.

```
touch foo
```

The file `foo` doesn't contain anything at all. It is completely empty. Now lets create a file with some content in it.

```
cat > bar
```

After you hit return, you will notice that you do not return to a command line prompt. Keep typing. Write a bunch of nonsense lines. Keep going. Write poetry if you like. Everything you're entering at the keyboard is now going into the file called `bar`. To end the file, you need to send the end-of-file character, which is control-D. This is an invisible character that is at the end of every text file. Hit the control key and then the letter d. Another way of saying this is hit the ^D character. Note that you don't type the ^ or the capital D. It's just the way we communicate in writing that one should type the control key and then the d key when the combination is invisible.

To see the contents of the file, you can also use the `cat` command. This dumps the entirety of the file into your terminal.

```
cat bar
```

This isn't very useful for viewing unless you're a speed reader. You can inspect the first 10 lines

of a file with the `head` command.

```
head bar
```

Similarly, you can inspect the last 10 lines with `tail`.

```
tail bar
```

Both of these programs have command line options so that you can see a different number of lines. For example, to see just the first line you would do the following:

```
head -1 bar
```

A more useful way to look at files is with a **pager**. The `more` and `less` commands let you see a file one terminal page at a time. This is what you used before when viewing the manual page for the `date` command. Use the spacebar to advance the page and q to quit. `more` and `less` do more or less the same thing, but oddly enough `less` does more than `more`. There are a lot of subtle jokes in the Unix culture.

```
less bar
```

Editing Files

Let's try editing a file with `nano`, which is a terminal-based editor.

```
nano bar
```

This changes the entire look of your terminal. Now you can change the random text you just wrote. Use the arrow keys to move the cursor around. Add some text by typing. Remove some text with the delete key. At the bottom, you can see a menu that uses control keys. To save the file you hit the ^O key (control and then the letter o). You will then be prompted for the file name, at which point you can overwrite the current file (bar) or make a new file with a different name. To exit nano, hit ^X. Note, you don't need to give nano a file name when you start it up.

Unix file names often have the following properties:

- all lowercase letters
- no spaces in the name (use underscores or dashes instead)
- an extension such as .txt

Navigating Directories

Whenever you are using a terminal, your focus is a particular directory. The files you just created above were in a specific directory. To determine what directory you are currently in, use the `pwd` command (print working directory).

```
pwd
```

This will tell you what your location is, which starts in your home directory. In MacOS, it will be `/Users/your_name` and in Ubuntu it will be `/home/your_name`. In addition to this **absolute path**, your location also has a **relative path**, which is simply the dot `.` character. You have your own given name, and you also have a pronoun you call yourself: **I** or **me**. Sometimes it's more convenient to use a pronoun than a whole name.

If you want to know what files are in your current working directory, use the following command:

```
ls .
```

`ls` will list your current directory if you don't give it an argument, so an equivalent statement is simply:

```
ls
```

Notice that `ls` reports the files and directories in your current working directory. It's a little difficult to figure out which ones are directories right now. So let's add a command line option to the `ls` command.

```
ls -F
```

This command adds a character to the end of the file names to indicate what kind of files they are. A forward slash character indicates that the file is a directory. These directories inside your working directory are called *sub-directories*. They are **below** your current location. There are also directories **above** you. There is at most one directory immediately above you. We call this your parent directory, which in Unix is called `..` (that wasn't a typo, it's two dots). You can list your parent directory as follows:

```
ls ..
```

The `ls` command has a lot of options. Try reading the `man` pages and trying some of them out. Now is a good time to experiment with a few command line options. Note that you can

specify them in any order and collapse them if they don't take arguments (some options have arguments).

```
man ls
ls -a
ls -l
ls -l -a
ls -a -l
ls -la
ls -al
```

There are two ways to specify a directory: **relative** path and **absolute** path. The command `ls ..` listed the directory above the current directory. The command `nano bar` edited the file `bar` in the current directory. You could also have written `nano ./bar`. What if you want to list some directory somewhere else or edit a file somewhere else? To specify the absolute path, you precede the path with a forward slash. For example, to list the absolute root of the Unix file system, you would type the following:

```
ls /
```

To list the contents of /usr/bin you would do the following

```
ls /usr/bin
```

This works exactly the same from whatever your current working directory is. But this is not true of the relative path.

```
ls ..
```

To change your working directory, you use the `cd` command. Try changing to the root directory

```
cd /
pwd
ls
```

Now return to your home directory by executing `cd` without any arguments.

```
cd
pwd
ls
```

Now go back to the root directory and create a file in your home directory. There are a couple short-cut ways to do this. The `HOME` variable contains the location of your home directory. But ~ (tilde) is another shortcut for the same thing and requires less typing. Note that to get the contents of the `HOME` variable, we have to precede it with a `$`.

```
cd /
touch $HOME/file1
touch ~/file2
```

Now create a couple more files but using absolute and relative paths rather than short-cuts.

```
touch /home/bios180student/file3
cd ~/Documents
pwd
ls
touch ../file4
```

Moving and Renaming Files

Your home directory is starting to fill up with a bunch of crap. Let's organize that stuff. First off, let's create a new directory for 'Stuff' using the `mkdir` command.

```
cd
mkdir Stuff
```

Notice that the directory starts with a capital letter. This isn't required, but it's a good practice. Now let's move some files into that new directory with the `mv` command.

```
mv foo Stuff
```

If there are 2 arguments and the 2nd argument is not a directory, the `mv` command will rename the file. Yes, it's weird that `mv` both moves and renames files. That's Unix.

```
mv bar barf
mv barf Stuff
ls
```

You can move more than one file at a time. The last argument is where you want to move it to.

```
mv file1 file2 file3 file4 Stuff
ls
ls Stuff
```

Copying and Aliasing

The `cp` command copies files. Let's say you wanted a copy of barf in your home directory.

```
cp Stuff/barf .
ls
ls Stuff
```

Now you have 2 copies of barf. If you edit one file, it will be different from the other. Try that with nano.

```
nano barf
# do some editing and save it
cat barf
cat Stuff/barf
```

When you do the copying, you don't have to keep the same name.

```
cp Stuff/barf ./bark
ls
```

Things are getting a little messy, so let's put these files back into the Stuff directory. But let's do it using the * wildcard, which is somewhat magical because it fills in missing letters of arguments. Every file that starts with `ba` will now be moved to Stuff.

```
mv ba* Stuff
ls
```

That was just two files, but I think you can imagine the power of moving 100 files with a single command.

An alias is another name for a file. Like a nickname for a person. These are often useful to organize your files and directories. Let's try it using the `ln -s` command (we always use the `-s` option with `ln`).

```
ln -s Stuff/foo ./foof
nano foof
# do some editing
cat foof
cat Stuff/foo
```

Deleting Files

You delete files with the `rm` command. Be careful, because once gone, they are gone forever.

```
ls Stuff
rm Stuff/file1
ls Stuff
rm foof
```

You didn't type those file names completely, right? You used tab completion, right?

You can delete multiple files at once too and even whole directories. Watch out though, that can get dangerous.m

Working with Text Files

There are 2 kinds of files: text and binary. A text file looks like the file you're reading. There are familiar letters, words, and punctuation (include spaces between words). Binary files don't look like this. Instead, they look like a bunch of random letters, some of which don't even display properly on your screen. Let's look at one.

```
cat /bin/ls
```

Yuck. Most of the programs you run on a computer are binary files. Generally speaking, binary files are for machine consumption and text files are for human consumption (or readability). Text files come in 3 common flavors. Unix text files have a newline character at the end of each line. Mac text files generally also use newline characters, but carriage returns were used in the past and are still sometimes used. Windows files end in carriage returns plus newlines (they have 2 characters to denote end of line). These differences in line endings can cause problems when interchanging files among computers. For this reason, avoid interchanging files between you Linux, Mac, and Windows computers. **One of the stupidest things you can do is to put sequence data into Microsoft Word or Excel (or similar software) and then attempt to use it in Unix.**

> Never create new files in Mac or Windows, only Unix/Linux.

Hello World

It's time to write your first python program. We're going to do this with nano. Start nano by typing the command name followed by the file you want to create.

```
nano hello_world.py
```

Now type the following lines:

```
#!/usr/bin/env python3

print('hello world')
```

Hit ^O (that's the command key plus the o key) to save the file and the ^W to exit nano. Now let's talk about the 3 lines you wrote.

Line 1 is what's known as an **interpreter directive**. It's also sometimes called a **hashbang**. It tells the shell which interpreter to use when the program executed (more on this in just a bit).

Line 2 is a blank line. Use blank lines to separate *thoughts* from each other. The hashbang is one such thought and the other is a print statement.

Line 3 prints 'hello world to the terminal. Now let's try running the command from the shell.

```
python3 hello_world.py
```

If everything worked okay, you should have seen 'hello world' in your terminal. If not, don't go on. Ask for help fixing your computing environment.

File Permissions

You might be wondering what line 1 did in your hello_world.py program. You're about to find out.

A file can have 3 kinds of permissions: read, write, and execute. These are abbreviated as `rwx`. Read and write are obvious, but execute is a little weird. Programs and directories need executable permission to access them.

Generally, you want read and write access to the files you create. But if you have some incredibly important data file, you might want to protect it from being edited, so you may want to remove write permission to make it read-only.

In addition to having 3 types of permissions, every file also has 3 types of people that can

access it: the owner (you), the group you belong to (e.g. a laboratory), or the public. For the purposes of learning how to program, we can treat these all the same. However, you can imagine that some files should not be readable by others (for example, your private poetry efforts).

So let's examine the file permissions on the hello_world.py program.

```
ls -lF hello_world.py
```

This will produce something like the following.

```
-rw-r--r--  1 iankorf  staff     45 Mar 29 13:56 hello_world.py
```

After the leading dash, there are 3 triplets of letters. The first triplet shows user permissions. I have read and write permission but not execute. The next triplets are for group and public. Both have read permission, but not write or execute. Let's first turn on all permissions for everyone using the `chmod` command and then list again.

```
chmod 777 hello_world.py
ls -lF hello_world.py
```

Notice that you can now see `rwx` for owner, group, and public.

```
-rwxrwxrwx  1 iankorf  staff     45 Mar 29 13:56 hello_world.py*
```

There is also an asterisk after the program name. The `-F` option in `ls` shows you what kind of file something is with a trailing character. If the file is a directory, there will be a trailing `/`.

You won't need to get complicated with permissions. The following 3 are all you need right now.

- `chmod 444` file is read only
- `chmod 666` file may be read and edited
- `chmod 777` file may be read, edited, and used as Unix command

The `chmod` command has two different syntaxes. The more human readable one looks like this.

```
chmod u-x hello_world.py
ls -lF hello_world.py
```

This command says: “change the user (u) to remove (-) the execute (x) permission from file hello_world.py”. You add permissions with +.

```
chmod u+x hello_world.py
ls -lF hello_world.py
```

The less readable `chmod` format is assigning all parameters in octal format. 4 is the read permission. 2 is the write permission. 1 is the execute permission. Each rwx corresponds to one octal number from 0 to 7. So `chmod 777` turns on all permissions for all types of people and `chmod 000` turns them all off.

Now that your hello_world.py program has execute permissions, you can use it like a Unix program. That is, you don’t have to type `python3` before the program name.

```
./hello_world.py
```

But what’s with the `./` before the program name. You don’t have to type that when you run the `ls` command or the `chmod` command, for example. That’s because those programs are in your **executable path** and `hello_world.py` is not. We’ll fix that in a bit.

Customizing Your Environment

There are 3 things you should do to customize your programming environment.

1. Set up your code directory
2. Create your GitHub repository
3. Customize your profile

Code Directory

Create a directory where you’re going to organize all of your programming projects. Let’s call this “Code”. Directories in your home directory often start with a capital letter, and we’ll follow this practice to reduce unintentional surprise. All of the programs and other stuff we do will end up in this directory.

```
cd
mkdir Code
```

GitHub Repository

If you're going to be a bioinformatics programmer (or just look like one), you need a GitHub account as part of your CV. If you want people to believe you're a programmer, you need a place where you're putting your code and logging your activity. Use your web browser to go to github (https://github.com) and create your account.

Choose a username that isn't stupid. Remember, this will be part of your CV. After setting your email and password, choose the free plan and then answer a few questions about your interests to create your account.

Now check your email to verify your email address. It's time to create your first repository! Name this `learning_python`. Select the radio button to make this Public, and also check the box to initialize with a README. Lastly, you should add a license in the dropdown menu to get in the habit of doing all programming things properly. I generally use the MIT License.

The next step is to add your repository to the Code directory you just created. Click on the green **Clone or download** button and copy the URL there. It should look something like "https://github.com/USERNAME/learning_python.git", where USERNAME is whatever you chose as your GitHub name.

```
cd ~/Code
git clone https://github.com/USERNAME/learning_python.git
```

This will create a new directory called `learning_python` in your `Code` directory. If you look inside, you will see that it contains two files: `LICENSE` and `README.md`. The LICENSE basically says that other people can use your code but that they have to acknowledge that you wrote it and if anything bad happens, it wasn't your fault. The README is the start of some documentation. Click on the README.md file and it will show you that it contains almost nothing. Click on the pen icon on the right, and you can edit the file. Write a sentence below the first line about what this is for and then hit the green "Commit changes" button at the bottom of the page.

Now you should see your new text. Note that the title is in a much larger font than your sentence. The `.md` suffix on the file indicates that it is written in Markdown format. This is a very simple way to create formatting from simple text. Get in the habit of using Markdown whenever you write plain text.

Now let's add another repository to your `Code` directory. This is the repository for the course. It contains a variety of files, including the document you're reading in its original Markdown format.

```
cd ~/Code
git clone https://github.com/iankorf/MCB185.git
```

Customize Profile

In order to simplify a few things, we need to customize your shell. First, we have to figure out which shell you're running. Your shell is in your SHELL environment variable. Here are two ways of seeing that.

```
printenv SHELL
echo $SHELL
```

If your shell is `/bin/bash` then check if you have a file called `.profile` or `.bash_profile` or `.bashrc` in your home directory. If you already have one of those files, edit it with nano. If not, create a `.profile` with nano.

If your shell is `/bin/zsh` then check if you have a file called `.zshrc`. If it exists, edit it with nano. If not, create it with nano.

Now enter the following 2 lines into the file you're editing.

```
alias ls="ls -F"
export PATH=$PATH:.
```

The first line makes it so that whenever you use the `ls` command, you're actually invoking `ls -F` which displays the file type by appending a `*` to executable files and a `/` to directories.

The second line adds your current directory `.` to the executable path. Now when your terminal is in the same directory as your program, it will run. Open up a new terminal and it will automatically load the new customizations.

```
cd Code
hello_world.py
```

Git Commands

Lets move `hello_world.py` into your `learning_python` repository and run it from there.

```
mv hello_world.py learning_python
cd learning_python
hello_world.py
```

Now let's add it to your repository so that it becomes part of your GitHub repo. Type the following command and observe the output.

```
git status
```

This shows that `hello_world.py` is not in your repository. In order to upload it back to GitHub, we issue the following 3 commands.

```
git add hello_world.py
git commit -m "initial upload"
git push
```

The `add` argument tells `git` we want to start tracking changes to this file. The `commit` tells `git` we are done. The `push` tells git to upload it back to GitHub.

The general workflow with `git` is the following.

1. Create a file
2. `git add`
3. `git commit`
4. `git push`
5. Time passes…
6. `git pull`
7. Edit the file
8. Go back to step 2

Quick Unix Reference

Token	Function
.	your current directory (see pwd)
..	your parent directory
~	your home directory (also $HOME)
^C	send interrupt signal to current process
^D	send end-of-file character
tab	tab-complete names
*	wildcard - matches everything
|	pipe output from one command to another
>	redirect output to file

Command	Example	Intent
`cat`	`cat > f`	create file f and wait for keyboard (see ^D)
	`cat f`	stream contents of file f to STDOUT
	`cat a b > c`	concatenate files a and b into c
`cd`	`cd d`	change to relative directory d
	`cd ..`	go up one directory
	`cd /d`	change to absolute directory d
`chmod`	`chmod 644 f`	change file permissions
	`chmod u+x f`	change file permissions
`cp`	`cp f1 f2`	make a copy of file f1 called f2
`date`	`date`	print the current date
`df`	`df -h .`	display free space on file system
`du`	`du -h ~`	display the sizes of your files
`git`	`git add f`	start tracking file f
	`git commit -m "message"`	finished edits, ready to upload

	`git push`	put changes into repository
	`git pull`	retrieve latest documents from repository
	`git status`	check on status of repository
`grep`	`grep p f`	print lines with the letter p in file f
`gzip`	`gzip f`	compress file f
`gunzip`	`gunzip f.gz`	uncompress file f.gz
`head`	`head f`	display the first 10 lines of file f
	`head -2 f`	display the first 2 lines of file f
`history`	`history`	display the recent commands you typed
`kill`	`kill 1023`	kill process with id 1023
`less`	`less f`	page through a file
`ln`	`ln -s f1 f2`	make f2 an alias of f1
`ls`	`ls`	list current directory
	`ls -l`	list with file details
	`ls -la`	also show invisible files
	`ls -lta`	sort by time instead of name
	`ls -ltaF`	also show file type symbols
`man`	`man ls`	read the manual page on ls command
`mkdir`	`mkdir d`	make a directory named d
`more`	`more f`	page through file f (see less)
`mv`	`mv foo bar`	rename file foo as bar
	`mv foo ..`	move file foo to parent directory
`nano`	`nano`	use the nano text file editor
`passwd`	`passwd`	change your password
`pwd`	`pwd`	print working directory
`ps`	`ps`	show current processes

	`ps -u ian`	show processes user ian is running
`rm`	`rm f1 f2`	remove files f1 and f2
	`rm -r d`	remove directory d and all files beneath
	`rm -rf /`	destroy your computer
`rmdir`	`rmdir d`	remove directory d
`sort`	`sort f`	sort file f alphabetically by first column
	`sort -n f`	sort file f numerically by first column
	`sort -k 2 f`	sort file f alphabetically by column 2
`tail`	`tail f`	display the last 10 lines of file f
	`tail -f f`	as above and keep displaying if file is open
`tar`	`tar -cf ...`	create a compressed tar-ball (-z to compress)
	`tar -xf ...`	decompress a tar-ball (-z if compressed)
`time`	`time ...`	determine how much time a process takes
`top`	`top`	display processes running on your system
`touch`	`touch f`	update file f modification time (create if needed)
`wc`	`wc f`	count the lines, words, and characters in file f