Lab 2

More on the Unix Shell

Lab Objective: *Introduce system management, calling Unix Shell commands within Python, and other advanced topics.*

In this lab, we will build upon the foundation of the previous lab. As in the last lab, the majority of learning will not be had in finsihing the problems, but in following the examples. By the end of this lab, you will have a solid foundation in Unix. You will be able to understand enough to learn any additional topics you want.

File Security

To begin, run the following command while inside the `Shell2/Python/` directory (`Shell2/` is the end product of `Shell1/` from the previous lab). Notice your output will differ from that printed below; this is for learning purposes.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
-rw-rw-r-- 1 username groupname 373 Aug  5 21:16 count_files.py
-rwxr-xr-x 1 username groupname  27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

Notice the first column of the output. The first character denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The remaining nine characters denote the permissions associated with that file. Specifically, these permissions deal with reading, wrtiting, and executing files. There are three categories of people associated with permissions. These are the user (the owner), group, and others. For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx` tell us the owner can read, write, and execute the file. The next three characters `r-x` tell us members of the same group can read and execute the file. The final three characters `--x` tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are two different ways to specify permissions, *symbolic permissions* notation and *octal permissions*

Command	Description
`chmod u+x file1`	Add executing (`x`) permissions to user (`u`)
`chmod g-w file1`	Remove writing (`w`) permissions from group (`g`)
`chmod o-r file1`	Remove reading (`r`) permissions from other other users (`o`)
`chmod a+w file1`	Add writing permissions to everyone (`a`)

Table 2.1: Symbolic permissions notation

Command	Description
`chmod 760 file1`	Sets `rwx` to user, `rw-` to group, and `---` to others
`chmod 640 file1`	Sets `rw-` to user, `r--` to group, and `---` to others
`chmod 775 file1`	Sets `rwx` to user, `rwx` to group, and `r-x` to others
`chmod 500 file1`	Sets `r-x` to user, `---` to group, and `---` to others

Table 2.2: Octal permissions notation

notation. Symbolic permissions notation is easier to use when we want to make small modifications to a file's permissions. See Table 2.1.

Octal permissions notation is easier to use when we want to set all the permissions as once. The number 4 corresponds to reading, 2 corresponds to writing, and 1 corresponds to executing. See Table 2.2.

The commands in Table 2.3 are also helpful when working with permissions.

Scripts

A shell script is a series of shell commands saved in a file. Scripts are useful when we have a process that we do over and over again. The following is a very simple script:

```
#!/bin/bash
echo "Hello World"
```

Problem 1. Using vim, create a file called `hello` that contains the previous text and save it. Note that no file type is necessary.

The first line starts with `"#!"`. This is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the `bash` interpreter. If we were unsure where the `bash` interpreter is saved, we run `which bash`.

To execute a script, type the script name preceded by `./`

```
$ ./hello
bash: ./hello: Permission denied

# Notice you do not have permission to execute this file. This is by default.
$ ls -l hello
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello
```

Command	Description
`chown`	change owner
`chgrp`	change group
`getfacl`	view all permissions of a file in a readable format.

Table 2.3: Other commands when working with permissions

Command	Description
`df dir1`	Display available disk space in file system containing `dir1`
`du dir1`	Display disk usage within `dir1` [`-a`, `-h`]
`free`	Display amount of free and used memory in the system
`ps`	Display a snapshot of current processes
`top`	Display interactive list of current processes

Table 2.4: Commands for resource management

Problem 2. Add executable permission to your `hello` script. Run the script again.

You can do this same thing with Python scripts. All you have to do is change the path following the shebang. To see where the Python interpreter is stored, run `which python`.

Problem 3. In the `Python/` directory you will find `count_files.py`. `count_files.py` is a python script that counts all the files within the `Shell2/` directory. Modify this file so it can be run as a script and change the permissions of this script so the user and group can execute the script.

Note: In the `subprocess.check_output` command, if `Shell2/` is not contained in your home directory (∼), you will need to change ∼ to the correct path to navigate there.

If you would like to learn how to run scripts on a set schedule, consider researching *cron jobs*.

Resource Management

To be able to optimize performance, it is valuable to always be aware of the resources we are using. Hard drive space and computer memory are two resources we must constantly keep in mind. The commands found in table 2.4 are essential to managing resources.

Command	Description
`COMMAND &`	Adding an ampersand to the end of a command runs the command in the background
`bg %N`	Restarts the Nth interrupted job in the background
`fg %N`	Brings the Nth job into the foreground
`jobs`	Lists all the jobs currently running
`kill %N`	Terminates the Nth job
`ps`	Lists all the current processes
`Ctrl-C`	Terminates current job
`Ctrl-Z`	Interrupts current job
`nohup`	Run a command that will not be killed if the user logs out

Table 2.5: Job control commands

Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In table 2.5, we have listed some of the most common commands used in job control. We strongly encourage you to experiment with these commands. In the `Scripts/` directory, you will find a `five_secs` and a `ten_secs` script that takes five seconds and ten seconds to execute respectively. These will be particularly useful as you are experimenting with these commands.

```
# Don't forget to change permissions if needed
$ ./ten_secs &
$ ./five_secs &
$ jobs
[1]+  Running       ./ten_secs &
[2]-  Running       ./five_secs &
$ kill %2
[2]-  Terminated    ./five_secs &
$ jobs
[1]+  Running       ./ten_secs &
```

Problem 4. In addition to the `five_secs` and `ten_secs` scripts, the `Scripts/` folder contains three scripts that will each take about a forty-five seconds to execute. Execute each of these commands in the background so all three are running at the same time. To verify all scripts are running at the same time, write the output of `jobs` to a new file `log.txt` saved in the `Scripts/` directory.

Python Integration

To this point, we have barely scratched the surface of all the functionality that Unix has to offer. However, the tools and commands we have addressed so far provide us with a foundation of the basics. Using the `subprocess` module in Python, we can call Unix commands. By combining Python and the Unix commands, our toolset is automatically broadened.

There are two functions in particular within the `subprocess` module we will use. When wanting to run a Unix command, use `subprocess.call()`. When wanting to run a Unix command and be able to store and manipulate the output, use `subprocess.check_output()`. These functions have a keyword argument `shell` that defaults to `False`. We want to set this argument to `True` to run the command in the Unix shell.

```
$ cd Shell-Lab/Documents
$ python
>>> import subprocess
>>> subprocess.call("ls -l", shell=True)
-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt
-rw-rw-r-- 1 username groupname  427 Aug  5 20:21 doc1.txt
-rw-rw-r-- 1 username groupname  326 Aug  5 20:21 doc2.txt
-rw-rw-r-- 1 username groupname  612 Aug  5 20:21 doc3.txt
-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 doc4.txt
-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt
-rw-rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt
>>> files = subprocess.check_output("ls -l", shell=True)
>>> files
'-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt\n-rw-rw-r-- 1 ↩
    username groupname  427 Aug  5 20:21 doc1.txt\n-rw-rw-r-- 1 username ↩
    groupname  326 Aug  5 20:21 doc2.txt\n-rw-rw-r-- 1 username groupname  612 ↩
    Aug  5 20:21 doc3.txt\n-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 ↩
    doc4.txt\n-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt\n-rw-↩
    rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt\n'
>>> files.split('\n')
['-rw-rw-r-- 1 username groupname  142 Aug  5 20:20 assignments.txt',
 '-rw-rw-r-- 1 username groupname  427 Aug  5 20:21 doc1.txt',
 '-rw-rw-r-- 1 username groupname  326 Aug  5 20:21 doc2.txt',
 '-rw-rw-r-- 1 username groupname  612 Aug  5 20:21 doc3.txt',
 '-rw-rw-r-- 1 username groupname  298 Aug  5 20:21 doc4.txt',
 '-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt',
 '-rw-rw-r-- 1 username groupname  920 Aug  5 23:50 words.txt',
 '']
>>> files = files.split('\n')
# To get rid of the last empty string in the list
>>> files.pop()
''

# Now that we have a list object, we can manipulate and analyze this data in ↩
    Python.
We can make it even more accessible by splitting the lines again
>>> files = [line.split() for line in files]
```

Problem 5. Create a `Shell` class in Python. Write a `find_file()` method that will search for a filename using the `find` command in the given directory.

Command	Description
`passwd`	Change user password
`uname`	View operating system name
`uname -a`	Print all system information
`uname -m`	Print machine hardware
`w`	Show who is logged in and what they are doing
`whoami`	Print userID of current user

Table 2.6: Commands for system administration.

Write a `find_word()` method that finds a given word within the contents of the directory using the `grep` command. Both functions should accept a directory keyword as input which defaults to `None`. If no directory location is provided, then set it to be the current directory within the function. For both these functions, return a list of filepaths.

Problem 6. Write a method for the `Shell` class that recursively finds the n largest files within a directory. Have a keyword argument for the directory that defaults to the current directory. Be sure that your function only returns files. Hint: To view the size of a file `file1`, you can use `ls -s file1` or `du file1`

System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in table 2.6 are used to learn more about the computer system.

Secure Shell

Let's say you are working for a company with a file server. Hundreds of people need to be able to access the content of this machine, but how is that possible? Or say you have a script to run that requires some serious computing power. How are you going to be able to access your company's super computer to run your script? We do this through *Secure Shell* (SSH).

SSH is a network protocol encrypted using public-key cryptography. The system we are connecting *to* is commonly referred to as the *host* and the system we are connecting *from* is commonly referred to as the *client*. Once this connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type `exit`.

As a warning, you cannot normally SSH into a Windows machine. If you want to do this, search on the web for available options.

```
$ whoami    # use this to see what your current login is
```

```
client_username
$ ssh my_host_username@my_hostname

# You will then be prompted to enter the password for my_host_username

$ whoami    # use this to verify that you are logged into the host
my_host_username

$ hostname
my_hostname

$ exit
logout
Connection to my_host_name closed.
```

Now that you are logged in on the host computer, all the commands you execute are as though you were executing them on the host computer.

Secure Copy

When we want to copy files between the client and the host, we use the *secure copy* command, `scp`. The following commands are run when logged into the client computer.

```
# copy filename to the host's system at filepath
$ scp filename host_username@hostname:filepath

#copy a file found at filepath to the client's system as filename
$ scp host_username@hostname:filepath filename

# you will be prompted to enter host_username's password in both these instances
```

Problem 7. You will either need a partner for this problem or have access to a username on another computer. Experiment with SSH. Verify that you can connect from a client to a host. Copy a few files between the host and the client.

Generating SSH Keys (Optional)

If there is a host that we access on a regular basis, typing in our password over and over again can get tedious. By setting up SSH keys, the host can identify if a client is a trusted user without needing to type in a password. If you are interested in experimenting with this setup, a Google search of "How to set up SSH keys" will lead you to many quality tutorials on how to do so.

Web Related

Sometimes you will need to download files from the internet. `wget` and `curl` are both used to download content from the web, and in many applications they both

perfform the same tasks. Most of the differences between `wget` and `curl` are beyond the scope of this book. At its most basic, `curl` is the more robust tools of the two while `wget` can download recursively. The provided examples will use `wget`.

Downloading files using Wget

When we want to download a single file, we just need the URL for the file we want to download. Running the command below will downlad a JPEG image of a person writing on a chalkboard. Similarly, you can download PDF files, HTML files, and other content simply by providing a different URL.

```
$ wget http://acme.byu.edu/wp-content/uploads/2013/07/0906-13-00903.jpg
```

The following are also useful commands using `wget`.

```
# Download files from URLs listed in urls.txt
$ wget -i list_of_urls.txt

# Download in the background
$ wget -b URL

# Download something recursively
$ wget -r --no-parent URL
```

Problem 8. In the `Documents/` directory, you will find a file named `urls.txt` with a list of URLs. Download the files in this list using `wget`. Move the pictures that will be downloaded to the `Photos/` directory.

Additional Material

sed and awk

`sed` and `awk` are two different scripting languages in their own right. Like Unix, these languages are easy to learn but difficult to master. It is very common to combine Unix commands and `sed` and `awk` commands. We will address the basics, but if you would like more information see ¡http://www.theunixschool.com/p/awk-sed.html¿

Printing Specific Lines Using sed

We have already used the `head` and `tail` commands to print the beginning and end of a file respectively. What if we wanted to print lines 30 to 40, for example? We can accomplish this using `sed`. In the `Documents/` folder, you will find the `lines.txt` file. We will use this file for the following examples.

```
# Same output as head -n3
$ sed -n 1,3p lines.txt
line 1
line 2
line 3

# Same output as tail -n3
$ sed -n 3,5p lines.txt
line 3
line 4
line 5

# Print lines 2-4
$ sed -n 3,5p lines.txt
line 2
line 3
line 4

# Print lines 1,3,5
$ sed -n -e 1p -e 3p -e 5p lines.txt
line 1
line 3
line 5
```

Find and Replace Using sed

Using `sed`, we can also perform find and replace. We can perform this function on the output of another commmand or we can perform this function in place on other files. The basic syntax of this `sed` command is the following.

```
sed s/str1/str2/g
```

This command will replace every instance of `str1` with `str2`. More specific examples follow.

```
$ sed s/line/LINE/g lines.txt
LINE 1
LINE 2
```

```
LINE 3
LINE 4
LINE 5

# Notice the file didn't change at all
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5

# To save the changes, add the -i flag
$ sed -i s/line/LINE/g lines.txt
$ cat lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
```

Formatting output using awk

Earlier in this lab we mentioned `ls -l` and as we have seen, this outputs lots of information. Using `awk`, we can select which fields we wish to print. Suppose we only cared about the file name and the permissions. We can get this output by running the following command.

```
$ ls -l | awk ' {print $1, $9} '
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we use quotation marks. Note it is a common mistake to forget to add these quotation marks. Inside these quotation marks, commands always take the same format.

```
awk ' <options> {<actions>} '
```

In the remaining examples we will not be using any of the options, but we will address various actions. For those interested in learning what options are available see ¡http://www.theunixschool.com/p/awk-sed.html¿.

In the `Documents/` directory, you will find a `people.txt` file that we will use for the following examples. In our first example, we use the `print` action. The `$1` and `$9` mean that we are going to print the first and ninth fields. Beyond specifying which fields we wish to print, we can also choose how many characters to allocate for each field.

```
# contents of people.txt
$ cat people.txt
male,John,23
female,Mary,31
female,Sally,37
male,Ted,19
male,Jeff,41
female,Cindy,25
```

```
# Change the field separator (FS) to "," at the beginning of execution (BEGIN)
# By printing each field individually proves we have successfully separated the ↩
    fields
$ awk ' BEGIN{ FS = "," }; {print $1,$2,$3} ' < people.txt
male John 23
female Mary 31
female Sally 37
male Ted 19
male Jeff 41
female Cindy 25

# Format columns using printf so everything is in neat columns in order (gender,↩
    age,name)
$ awk ' BEGIN{ FS = ","}; {printf "%-6s %2s %s\n", $1,$3,$2} ' < people.txt
male   23 John
female 31 Mary
female 37 Sally
male   19 Ted
male   41 Jeff
female 25 Cindy
```

The statement `"%-6s %2s %s\n"` formats the columns of the output. This says to set aside six characters left justied, then two characters right justified, then print the last field to its full length.

Problem 9. Inside the `Documents/` directory, you should find a file named `files.txt`. This file contains details on approximately one hundred files. The different fields in the file are separated by tabs. Using `awk`, `sort`, pipes, and redirects, write a file named `date_modified.txt` with the following specifications:

- in the first column, print the date the file was modified
- in the second column, print the name of the file
- sort the file from newest to oldest based on the date last modified

All this can be accomplished using one command.

We have barely scratched the surface of what `awk` can do. Performing an internet search for "awk one-liners" will give you many additional examples of useful commands you can run using `awk`.