2 Unix Shell 2

Lab Objective: *Introduce system management, calling Unix Shell commands within Python, and other advanced topics. As in the last lab, the majority of learning will not be had in finishing the problems, but in following the examples.*

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* 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.

1

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

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.

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

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`.

Problem 1. Create a file called `hello.sh` that contains the previous text and save it in the `Scripts/` directory. The file extension `.sh` is technically unnecessary, but it is good practice to always include an extension.

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

```
$ cd Scripts
$ ./hello.sh
bash: ./hello.sh: Permission denied

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

Problem 2. Add executable permissions to `hello.sh`. Run the script to verify that it worked.

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`.

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 3. The Python/ directory contains a file called count_files.py, 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

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.

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.

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

The `five_secs.sh` and `ten_secs.sh` scripts in the `Scripts/` directory take 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.sh &
$ ./five_secs.sh &
$ jobs
[1]+  Running        ./ten_secs.sh &
[2]-  Running        ./five_secs.sh &
$ kill %2
[2]-  Terminated     ./five_secs.sh &
$ jobs
[1]+  Running        ./ten_secs.sh &
```

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

Using Python for File Management

Bash itself has control flow tools like if-else blocks and loops, but most of the syntax is highly unintuitive. Python, on the other hand, has extremely intuitive syntax for these control flow tools, so using Python to do shell-like tasks can result in some powerful but specific file management programs. Table 2.6 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 2.7. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`.

Shell Command	Python Function
ls	`os.listdir()`
cd	`os.chdir()`
pwd	`os.getcwd()`
mkdir	`os.mkdir()`, `os.mkdirs()`
cp	`shutil.copy()`
mv	`os.rename()`, `os.replace()`
rm	`os.remove()`, `shutil.rmtree()`
du	`os.path.getsize()`
chmod	`os.chmod()`

Table 2.6: Shell-Python compatibility

Function	Description
`os.walk()`	Iterate through the subfolders of a given directory.
`os.path.isdir()`	Return True if the input is a directory.
`os.path.isfile()`	Return True if the input is a file.
`os.path.join()`	Join several folder names or file names into one path.
`glob.glob()`	Return a list of file names that match a pattern.
`subprocess.call()`	Execute a shell command.
`subprocess.check_output()`	Execute a shell command and return its output as a string.

Table 2.7: Other useful Python functions for shell operations.

```python
>>> import os
>>> from glob import glob

# Get the names of all Python files in the Python/ directory.
>>> glob("Python/*.py")
['Python/calc.py',
 'Python/count_files.py',
 'Python/mult.py',
 'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
>> glob("**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/img_1796.jpg',
 'Photos/img_1842.jpg',
 'Photos/img_1879.jpg',
 'Photos/img_1987.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2464.jpg',
```

```
 'Photos/IMG_2679.jpg',
 'Photos/IMG_2746.jpg']

# Walk through the directory, looking for .sh files.
>>> for directory, subdirectories, files in os.walk('.'):
...     for filename in files:
...         if filename.endswith(".sh"):
...             print(os.path.join(directory, filename))
...
./Scripts/five_secs.sh
./Scripts/script1.sh
./Scripts/script2.sh
./Scripts/script3.sh
./Scripts/ten_secs.sh
```

Problem 5. Write a Python function `grep()` that accepts the name of target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern, then determine which ones contain the target string. For example, `grep("*.py", "range(")` should search Python files for the command `range(`.

Validate your function by comparing it to `grep -lR` in the shell.

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Use `subprocess.call()` to run a Unix command, or `subprocess.check_output()` to run a Unix command and record its output.

```
$ cd Shell2/Scripts
$ python
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3.sh
-rw-r--r--  1 username  groupname  21 Aug 26  2016 ten_secs.sh
0                                 # decode() translates the result to a string.
>>> file_info = subprocess.check_output(["ls", "-l"]).decode()
>>> file_info.split('\n')
['total 40',
 '-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3.sh',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 ten_secs.sh',
 '']
```

> **ACHTUNG!**
>
> Be extremely careful when creating a shell process from Python. If the commands depend on user input, the program is vulnerable to a *shell injection attack*. For example, consider the following function.
>
> ```
> >>> def inspect_file(filename):
> ...     """Return information about the specified file from the shell."""
> ...     return subprocess.check_output(["ls", "-l", filename]).decode()
> ```
>
> If `inspect_file()` is given the input `".; rm -rf /"`, then `ls -l .` is executed innocently, and then `rm -rf /` destroys the computer.[a] Be careful not to execute a shell command from within Python in a way that a malicious user could potentially take advantage of.
>
> ---
> [a]See https://en.wikipedia.org/wiki/Code_injection#Shell_injection for more example attacks.

> **Problem 6.** Write a Python function that accepts an integer n. Search the current directory and all subdirectories for the n largest files. Return a list of filenames, in order from largest to smallest.
> (Hint: the shell commands `ls -s` and `du` show the file size.)

Downloading Files

The Unix shell has tools for downloading files from the internet. The most popular are `wget` and `curl`. At its most basic, `curl` is the more robust of the two while `wget` can download recursively.

When we want to download a single file, we just need the URL for the file we want to download. This works for PDF files, HTML files, and other content simply by providing the right URL.

```
$ wget https://github.com/Foundations-of-Applied-Mathematics/Data/blob/master/↩
    Volume1/dream.png
```

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 7.** The file `urls.txt` in the `Documents/` directory contains a list of URLs. Download the files in this list using `wget` and move them to the `Photos/` directory.
>
> When you finish this problem, archive and compress your entire `Shell2/` directory and save it as `ShellFinal.tar.gz`. Include the `-p` flag on `tar` to preserve the file permissions. (Hint: see the previous lab for a refresher on `tar`. See also `https://xkcd.com/1168/`.)

One Final Note

Though there are multiple Unix shells, one of the most popular is the *bash* shell. The bash shell is highly customizable. In your home directory, you will find a hidden file named `.bashrc`. All customization changes are saved in this file. If you are interested in customizing your shell, you can customize the prompt using the `PS1` environment variable. As you become more and more familiar with the Unix shell, you will come to find there are commands you run over and over again. You can save commands you use frequently using `alias`. If you would like more information on these and other ways to customize the shell, you can find many quality reference guides and tutorials on the internet.

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.

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`.

System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in Table 2.8 are used to learn more about the computer system.

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 2.8: Commands for system administration.

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
```

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.