2 Unix Shell 2

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

Archiving and Compression

In file management, the terms archiving and compressing are commonly used interchangeably. However, these are quite different. Archiving is combining a certain number of files into one file. The resulting file will be the same size as the group of files that were archived. Compressing takes a file or group of files and shrinks the file size as much as possible. The resulting compressed file will need to be extracted before being used.

The ZIP file format is common for archiving and compressing files. If the zip Unix command is not installed on your system, you can download it by running

```
>>> sudo apt-get install zip
```

Note that you will need to have administrative rights to download this package. To unzip a file, use `unzip`.

> **Note**
>
> To begin this lab, unzip the `Shell2.zip` file into your `UnixShell2/` directory using a terminal command.

```
# Unzip a zipped file using the unzip command.
$ unzip Shell2.zip
Archive Shell2.zip
   creating: Shell2/
   creating: Shell2/Test/
  inflating: Shell2/.DS_Store
   creating: Shell2/Scripts/
```

```
extracting: Shell2/Scripts/fiteen_secs
extracting: Shell2/Scripts/script3
extracting: Shell2/Scripts/hello.sh...
```

While the `zip` file format is more popular on the Windows platform, the `tar` utility is more common in the Unix environment.

> **Note**
>
> When submitting this lab, you will need to archive and compress your entire `Shell2/` directory into a file called `Shell2.tar.gz` and push `Shell2.tar.gz` as well as `shell2.py` to your online repository.
>
> If you are doing multiple submissions, make sure to delete your previous `Shell2.tar.gz` file before creating a new one from your modified `Shell2/` directory. Refer to `Unix1` for more information on deleting files.
>
> As a final note, please do not push the entire directory to your online repository. *Only* push `ShellFinal.tar.gz` and `shell2.py`.

The example below demonstrates how to archive and compress our `Shell2/` directory. The `-z` flag calls for the `gzip` compression tool, the `-v` flag calls for a verbose output, the `-p` flag tells the tool to preserve file permission, and the `-f` flag indicates the next parameter will be the name of the archive file. Note that the `-f` flag must always come last.

```
# Remove your archive tar.gz file if you already have one.
$ rm -v Shell2.tar.gz
removed 'Shell2.tar.gz'

# Create a new one from your update Shell2 directory content.
# Remember that the * is the wildcard that represents all strings.
$ tar -zcpf Shell2.tar.gz Shell2/*
```

Working with Files

Displaying File Contents

The unix file system presents many opportunities for the manipulation, viewing, and editing of files. Before moving on to more complex commands, we will look at some of the commands available to view the content of a file.

The `cat` command, followed by the filename, will display all the contents of a file on the terminal screen. This can be problematic if you are dealing with a large file. There are a few available commands to control the output of `cat` in the terminal. See Table 2.1.

As an example, use `less <filename>` to restrict the number of lines that are shown. With this command, use the arrow keys to navigate up and down and press `q` to exit.

Command	Description
`cat`	Print all of the file contents
`more`	Print the file contents one page at a time, navigating forwards
`less`	Like more, but you navigate forward and backwards
`head`	Print the first 10 lines of a file
`head -n`k	Print the first k lines of a file
`tail`	Print the last 10 lines of a file
`tail -n`k	Print the last k lines of a file

Table 2.1: Commands for printing file contents

Pipes and redirects

To combine terminal commands, we can use *pipes*. When we combine or *pipe* commands, the output of one command is passed to the other. We pipe commands together using the `|` (*bar*) operator. In the example directly below, the `cat` command output is piped to `wc -l` (`wc` stands for word count, and the `-l` flag tells the `wc` command to count lines).

In the second part of the example, `ls -s` is piped to `sort -nr`. Refer to the *Unix 1* lab for explanations of `ls` and `sort`. Recall that the `man` command followed by an additional command will output details on the additional command's possible flags and what they mean (for example `man sort`).

```
$ cd Shell2/Files/Feb
# Output the number of lines in assignments.txt.
$ cat assignments.txt | wc -l
9
# Sort the files by file size and output file names and their size.
$ls -s | sort -nr
4 project3.py
4 project2.py
4 assignments.txt
4 pics
total 16
```

In addition to *piping* commands together, when working with files specifically, we can use `redirects`. A *redirect*, represented as `<` in the terminal, passes the file to a terminal command.

To save a command's output to a file, we can use `>` or `>>`. The `>` operator will overwrite anything that may exist in the output file whereas `>>` will append the output to the end of the output file. Examples of *redirects* and writing to a file are given below.

```
# Gets the same result as the first command in the above example.
$ wc -l < assignments.txt
9
# Writes the number of lines in the assignments.txt file to word_count.txt.
$ wc -l < assignments.txt >> word_count.txt
```

Problem 1. The `words.txt` file in the `Documents/` directory contains a list of words that are not in alphabetical order. Write an alphabetically sorted list of words in `words.txt` to a new file in your `Documents/` called `sortedwords.txt` using pipes and redirects. After you write the alphabetized words to the designated file, also write the number of words in `words.txt` to the end of `sortedwords.txt`. Save this file in the `Documents/` directory. Try to accomplish this with a total of two commands or fewer.

Resource Management

To be able to optimize performance, it is valuable to be aware of the resources, specifically hard drive space and computer memory, being used.

Job Control

One way to monitor and optimize performance is in job control. Any time you start a program in the terminal (you could be running a script, opening ipython, etc.,) that program is called a `job`. You can run a job in the foreground and also in the background. When we run a program in the foreground, we see and interact with it. Running a script in the foreground means that we will not be able to enter any other commands in the terminal while the script is running. However, if we choose to run it in the background, we can enter other commands and continue interacting with other programs while the script runs.

Consider the scenario where we have multiple scripts that we want to run. If we know that these scripts will take awhile, we can run them all in the background while we are working on something else. Table 2.2 lists some common commands that are used in job control. We strongly encourage you to experiment with some of 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 2.2: Job control commands

The `fifteen_secs` and `five_secs` scripts in the `Scripts/` directory take fifteen seconds and five seconds to execute respectively. The python file `fifteen_secs.py` in the `Python/` directoy takes fifteen seconds to execute, this file counts to fifteen and then outputs *"Success!"*. These will be particularly useful as you are experimenting with these commands.

Remember, that when you use the ./ command in place of other commands you will probably need to change permissions. For more information on changing permissions, review *Unix 1*. Run the following command sequence from the Shell2 directory.

```
# Remember to add executing permissions to the user.
$ ./Scripts/fifteen_secs &
$ python Python/fifteen_secs.py &
$ jobs
[1]+  Running       ./Scripts/fifteen_secs &
[2]-  Running       python Python/fifteen_secs.py &
$ kill %1
[1]-  Terminated    ./Scripts/fifteen_secs &
$ jobs
[1]+  Running       python Python/fifteen_secs.py &
# After the python script finishes it outputs the results.
$ Success!
# To move on, click enter after "Success!" appears in the terminal.

# List all current processes
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   44 tty1     00:00:00 ps

$ ./Scripts/fifteen_secs &
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   59 tty1     00:00:00 fifteen_secs
   60 tty1     00:00:00 sleep
   61 tty1     00:00:00 ps

# Stop fifteen_secs
$ kill 59
$ ps
  PID TTY          TIME CMD
    6 tty1     00:00:00 bash
   60 tty1     00:00:00 sleep
   61 tty1     00:00:00 ps
[1]+  Terminated              ./fifteen_secs
```

Problem 2. In addition to the `five_secs` and `fifteen_secs` scripts, the `Scripts/` folder contains three scripts (named `script1`, `script2`, and `script3`) that each take about forty-five seconds to execute. From the `Scripts` directory, execute each of these commands in the background in the following order; `script1`, `script2`, and `script3`. Do this 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.
(Hint: In order to get the same output as the solutions file, you need to run the ./ command and not the *bash* command.)

Using Python for File Management

OS and Glob

Bash 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 2.3 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

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 2.3: Shell-Python compatibility

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 2.4. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`. Look at the example below and then try out a few things on your own to try to get a feel for them.

Function	Description
`os.walk()`	Iterate through the subfolders and subfolder files 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 2.4: Other useful Python functions for shell operations.

```
# Your output may differ from the example's output.
>>> 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/fifteen_secs.py
 'Python/mult.py',
```

```
 'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
# The recursive parameter lets '**' match more than one directory.
>> glob("**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/img_1879.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_2746.jpg',
 'Photos/IMG_2679.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/img_1796.jpg',
 'Photos/IMG_2464.jpg',
 'Photos/img_1987.jpg',
 'Photos/img_1842.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/hello.sh
./Scripts/organize_photos.sh
```

Problem 3. Write a Python function `grep()` that accepts the name of a target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern. Next, check within the contents of the matched file for the target string. For example, `grep("range", "*.py")` should search Python files for the command `range`. Return a list of the file paths that matched the file pattern *and* the target string. For example, if you're in the Shell2/ directory and your grep function matches the `'calc.py'` file then your grep should return `'Python/calc.py'`

The Subprocess module

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Some important commands for executing shell commands from the `subprocess` module are listed in Table 2.5.

```
$ cd Shell2/Scripts
$ python
```

Function	Description
`subprocess.call()`	run a Unix command
`subprocess.check_output()`	run a Unix command and record its output
`subprocess.check_output.decode()`	this tranlates Unix command output to a string
`subprocess.Popen()`	use this to pipe togethether Unix commands

Table 2.5: Python subprocess module important commands

```
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r--  1 username  groupname  20 Aug 26  2016 five_secs
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2
-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3
-rw-r--r--  1 username  groupname  21 Aug 26  2016 fiften_secs
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',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script1',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script2',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 script3',
 '-rw-r--r--  1 username  groupname  21 Aug 26  2016 fiften_secs',
 '']
```

`Popen` is a class of the subprocess module, with its own atrributes and commands. It pipes together a few commands, similar to we did at the beginning of the lab. This allows for more versatility in the shell input commands. If you wish to know more about the `Popen` class, go to the subprocess documentation on the internet.

```
$ cd Shell2
$ python
>>> import subprocess
>>> args = ["cat Files/Feb/assignments.txt | wc -l"]
# shell = True indicates to open a new shell process
# note that task is now an object of the Popen class
>>> task = subprocess.Popen(args, shell=True)
>>> 9
```

ACHTUNG!

If shell commands depend on user input, the program is vulnerable to a *shell injection attack*. This applies to Unix Shell commands as well as other situations like web browser interaction with web servers. Be extremely careful when creating a shell process from Python. There are specific functions, like shlex.quote(), that quote specific strings that are used to construct shell commands. But, when possible, it is often better to avoid user input altogether. 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 by force deleting everything in the root directory.[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 4. Using os.path and Glob, write a Python function that accepts an integer n. Search the current directory and all subdirectories for the n largest files. Then sort the list of filenames from the largest to the smallest files. Next, write the line count of the smallest file to a file called `smallest.txt` into the current directory. Finally, return the list of filenames, including the file path, in order from largest to smallest.
(Hint: the shell commands `ls -s` shows the file size.)

As a note, same as in problem 3, to get this problem correct, you need to return the entire file path **starting from the directory that was searched and continuing to the name of the file**. Do not return just the filenames, or the complete file path. For example, if you are currently in the UnixShell2/ directory, meaning the next directory down to be searched will be Shell2, then Shell2/ should be the first part in the names of largest files returned by your function. More concretely, if `'data.txt'` is one files your function will return then instead of returning just `'data.txt'` or all of `'YourComputerSpecificFilePath/UnixShell2/Shell2/Files/Mar/docs/data.txt'` as part of your list, you would return only `'Shell2/Files/Mar/docs/data.txt'`. Notice the paths returned will vary based on both the current working directory you're in and what directories were searched. However, also make sure that your file paths do not begin with './' (Hint: To avoid the additional './' in your file path, use Glob instead of os.walk.)

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. This means that `wget` is capable of following links and directory structure when downloading content.

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 5. 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.

sed and awk

`sed` and `awk` are two different scripting languages in their own right. `sed` is a stream editor; it perfoms basic transformations on input text. `Awk` is a text processing language that manipulates and reports data. 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.

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 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 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.

```
$ cd Shell2/Documents
$ ls -l | awk ' {print $1, $9} '
total
-rw-r--r--. assignments.txt
```

```
-rw-r--r--. doc1.txt
-rw-r--r--. doc2.txt
-rw-r--r--. doc3.txt
-rw-r--r--. doc4.txt
-rw-r--r--. files.txt
-rw-r--r--. lines.txt
-rw-r--r--. newfiles.txt
-rw-r--r--. people.txt
-rw-r--r--. review.txt
-rw-r--r--. urls.txt
-rw-r--r--. words.txt
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we have to use quotation marks. 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.

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. This is done using the `%` command within the `printf` command, which allows us to edit how the relevant data is printed. Look at the last part of the example below to see how it is done.

```
# 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 space at the beginning of run using BEGIN
# 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 different ↩
    order
$ 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 justified, then two characters right justified, then print the last field to its full length.

Problem 6. 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 it to a new file in the current directory 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 of this can be accomplished using one command.
(Hint: change the field separator to account for tab-delimited files by setting `FS = "\t"` in the `BEGIN` 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`.

NOTE

Remember to archive and compress your `Shell2` directory before pushing it to your online repository for grading.

Additional Material

Customizing the Shell

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 with `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.

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.

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.