# 2023-09-18_BoC_LC-ShellLessonScript.pdf

## LEARNING GOALS

1. Get an introduction to what the shell is and a sample of what it can do
2. Jargon busting - learn some of the terminology used when talking about the shell
3. Learn a few simple and useful tasks that the shell can. For example:
   a. Exploring your directory structure and files
   b. Copying directory structures
   c. Batch creation of directory structures
   d. Batch renaming of files
   e. Counting, Sorting, and Filtering things
   f. Simple text mining (if we have time)
4. Inspire your imagination to recognise situations when you shell might be useful for your tasks
5. Build confidence about understanding the machine you use, probably every day
6. Have fun

## PREAMBLE

Normally, the LC Unix Shell lesson takes about 4 hours. The standard lesson covers (paste to etherpad):

1. What is the shell and some of what it can do for you. (5 mins)
2. Navigating the filesystem (30 mins)
3. Working with files and directories (30 mins)
-------------------------------------------------------
4. Loops (30 mins)
5. Counting, Sorting, and Filtering (90 mins)
-------------------------------------------------------
6. Working with free text (60 mins)

The last two topics take about 2.5 hrs. But given I have only 2 hrs for this introduction, I will cover at least the first 3 topics, and as much of 4 and 5 as time allows. It will be enough for you get a glimpse of what the shell can do, and for you to go forward to the Git and GitHub lesson tomorrow.

## What is the Shell?

Simplified schematic for the "layers" of the Unix cake from top to bottom [copy cake from EtherpadStuff.txt file]:

```
------------------------
You
------------------------
terminal emulator
("MINGW64" for Git Bash on PCs,"Terminal" for Macs)
------------------------
shell (Unix Bash)
------------------------
kernel
------------------------
hardware
------------------------
```

- interface/medium to interact with the Unix system/kernal
- gathers input from users, executes the program based on input, displays output
- an environment in which we run commands, programs, shell scripts
- "shell" which wraps around the kernel?

## Why Use the Shell?

- It is fast and has little problem handling large files.
    - For example, as of July 2023, Excel allows a maximum row number of 1,048,576 rows (roughly 1 million) and 16,384 columns per worksheet.(roughly 1.05 million) but dealing with such large files would be very slow. Also, in the sciences, 1 million may not be such a large number after all!

`<https://support.microsoft.com/en-gb/office/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3>  DOA: 17 Sep 2023`

- The shell's speed is achieved by processing large files **one line at a time**. I.e. the machine only has to see and keep in memory the next line, not the next 500 or 5000 line.

- Good at repetition, batch operations, and is also a scripting language. – reducing chances of human error at each iteration; reproducibility .

- It's used for communicating with remote machines and high-performance computers.

- But with great power comes great responsibilities and some danger. It is not idiot proof.
  - Unix and the shell assume you are intelligent and know what you want to do. There is no undelete or trash bin in the unix shell.
  - Always save a copy of your raw data
  - So go slow, be careful, build up larger scripts piece by piece
  - Use version control (like git, which we'll cover tomorrow).

- The Shell is better for certain types of tasks than for others. You need to use your knowledge and judgement about which tool to use for which task. I wouldn't use the shell or a text editor to write a report or create a visualization but I would use it for some quick and dirty tasks. (Stick shift vs auto trans?)

## Some Terminology and Basic Information

- The **shell** is a program that takes keyboard commands and passes them to the operating system to carry out. There are different flavours of shell. We will be working with a variation of the Unix shell called **bash**.
- The term, the **"command line",** is synonymous with the shell.
- The **terminal emulator** is a Graphical User Interface for the operating system (like Windows and the Mac OS) to interact with the shell. (In Macs, this

program is **Terminal** and comes with your OS. You find it in the Applications folder, then Utilities. In Windows, the emulator is **MINGW64.)** ['Minimum GNU for Windows 64 bit'. The name of a compiler used to build an extra copy of bash that "git for Windows" includes.]

- In a GUI, you are familiar with the metaphor from the desktop world, "folders." In the shell, we use the term "**directories**."
- Unless you specify differently, the standard (default) input and output for the shell is the terminal (i.e. your keyboard and screen).

## SHELL COMMAND SYNTAX (GRAMMAR)

The syntax of a shell command is the command followed by one or more option, followed by one or more argument.

```
command [-option(s)] [argument(s)]
verb   ~adverb          object
```

**Options** (or **flags**) modify the behaviour of a command. They are usually prefaced by a hyphen. Many commands allow multiple options to be strung together. Many versions of unix allow for "long options" in the format of two hyphens followed by a word (rather than a hyphen followed by a letter). See man pages for examples.

**Arguments** are things (often files) upon which the command acts.

## EXPLORING THE LANDSCAPE

### The Shell Prompt

- the **dollar sign** "$" is called the **prompt**. It indicates the shell is ready to receive a command from you. (Note: Most unix shells use the dollar sign as their prompt. You will likely see more text before the dollar sign. What text will depend on your shell's default setting.)

Sometimes you may see the **">" prompt**, which means the shell is waiting for further input from you (maybe you accidentally hit the ENTER key and you have not finished your command).

Ok, now let's see where you are, the “working directory,” the directory you are in right now. You do this with the **`pwd`** command, which means "print working directory."

```
$ pwd
```

You can also ask the shell who it thinks you are (user identity), which user by using the **`whoami`** command.

```
$ whoami
kathy
```

**NAVIGATION**

**cd -- change directory**

Use the **`cd`** command followed by the **pathname** of the target location/directory to change your working directory (where we are standing in our directory tree) to the target directory.

**Path names** can be specified in two ways, absolute address and relative address.

- **Relative** path names begin from where you are, your working directory (e.g. "./data" OR just "data")
- **Absolute** path names begin from the root directory. You must give your full address. (e.g. /c/Users/kathy/Desktop/libcarp/data/)
- Example: directions to washroom

**Useful `cd` Options**

| | |
|---|---|
| `cd ~` | Home directory, another way of going to the home directory |
| `cd` | Go to home directory (same as cd ~) |

|  |  |
|---|---|
| `cd ..` | Go up one level , the Parent directory |
| `cd .` | The symbol “.” stands for the working directory. |
| `cd -` | Go to previous directory (not one layer up but previous) |

**Speeding up your typing and accuracy**

- history - previous X commands (!xxx to execute the given command)
- UP arrow - previous command

Tab Completion

- Very useful. Good for reducing human error and speeding up your typing.
- if there are more than one item with starting characters, the shell will pause for you to choose

**MANIPULATING DIRECTORIES**

**mkdir -- Make Directory**

```
mkdir data2016
```

You can make more than one directory at a time.

```
mkdir data2016 data2017
```

Check with **`ls`** to verify what you've done.

**rmdir -- Remove (Delete) Directory, USE WITH CAUTION**

The is no "undelete" in the shell.

```
rmdir data2016
```

As a modicum of safety, the shell will not let you remove a directory unless it is empty.

To make it more safe, you can use the **-i** flag to invoke the **interactive mode**, requiring you to give permission for each item the shell deletes.

```
  rmdir -i
```

**mv -- move (rename) files or directories**

```
  mv 100kLines+Lables.tsv shelldata.tsv
```

This results in one file with a new name.

**cp -- Copy files or directories**

```
  cp shelldata.tsv shelldata-copy.tsv
```

This results in two copies of the same file, with different names.

**Note:** Unix is case sensitive but **bash is not case sensitive**, so unless you are using a linux machine, you usually don't have to worry about getting your case right.

**ls -- List**
If we want to know what is in our directory, we use the list ls command.

```
  ls
```

There are several useful flags we can use with list.
List all files, including hidden files

```
  ls -a
```

To find out what type of items are in the list, use the -F flag.

```
  ls -F
```

(Blank = file, / = directory, * = link, @ = executable file.)

To see more information about the files and directories, you want the long format -l

```
ls -l
```

This shows the size of your file in bytes. That's not very useful for most of us. So we can use the **-h** flag (**human readable**).

```
ls -lh
```

Now you see that the size if given in kilobyte and megabytes.

**Wildcards**

You can also use list with wildcards.

| | |
|---|---|
| * | Any characters |
| ? | Any one character |
| `[characters]` | Any character which is a member of the set `characters` |
| `[!characters]` | Any character which is a NOT member of the set `characters` |
| `[[:class:]]` | Any character that is a member of the specified `class` |

**Examples:**

| | |
|---|---|
| `ls *` | list all files |
| `ls g*` | list all files beginning with ‘g’ |
| `ls data???` | All files beginning with ‘data’ followed by exactly 3 characters |
| `[abc]*` | Any file beginning with a, b, or c |
| `[[:upper:]]*` | files beginning with uppercase letter |
| `*[[:lower]123]` | any file ending with a lowercase letter or the numerals 1, 2, or 3 |

**touch - create a blank file**
It creates an empty file of the name you give in the argument.

```
  touch file1 file2
```

## Counting, Sorting, Filtering, and Loops

### `echo` Command and Expansion

The echo command is very simple: it prints out its text arguments to standard output.

```
     echo Hello World
```

`echo` may be a simple command but it can be useful, esp. for batch processing (running a command on more than one argument/file/directory).

For example, since commands like `rm` and `cp` can be dangerous because `rm` will delete files and `cp` will overwrite existing files without asking "Are you sure?" You can use echo first and then change the command to rm or cp after you've checked those are the files you want to act upon.

```
  echo 2011*.txt
  rm 2011*.txt
```

**What really happens when you use wildcards:**
If you tell the shell:

```
     echo *
```

Why did you not just get the asterisk as the output?  Because expansions come ***before*** commands.

**Expansion**

- The shell expands wildcards *before* it carries out the command.  So, in this case, the shell expanded the asterisk (matching all filenames in our working directory) before it carried out the `echo` command.
- There are several types of expansions:
  - filename expansion with "*"   E.g.  echo *.txt
  - parameter expansion – what we did with echo $SP1 (variable names like $SP1 or $USER)
  - **brace expansion**

What happens if you want to prevent expansion?  Put things in quotes.  There are double quotes and single quotes.  We won't go into details here into the difference between the two but if you want the shell to take what you enter as an argument verbatim, use single quotes.

```
    echo this is  some               text
    echo ‘this is     some               text’
```

Some interesting things you can do with **brace expansion**

syntax:   ***preamble*{PATTERN}*postscript***

where PATTERN can be CSV list of strings or a range of integers or single characters. E.g. `{A, B, C}` or `{1..3}` or `{a..z}`

Try this out:

```
  echo {A, B, C}
  echo file0{1, 2, 3}.txt

  touch file0{1..3}.txt
  ls
```

**What is the use of this?**

- Say you are a photographer who wishes to organise your photos in a directory called <Pics> which contains sub-directories named in numeric year-month format. You want to create a bunch of sub-directories to hold the photos you took between 2015 to 2016. Can you think of a way to combine the make directory command with brace expansion to do this quickly and efficiently? Try this out line by line.

```
mkdir Pics
cd Pics
mkdir {2015..2016}-0{1-9} {2015..2016}-{10..12}
ls
```

**REDIRECTION, PIPES, AND FILTERS** (count, sort, filter, unique)

- Next, let's make a new directory where you are going to work and put your data files. You do this with the mkdir "make directory" command.
  mkdir datafiles
- Get some data
- Create a directory on your desktop called "shell-data".
- The file shelldate.tsv contains bibliographic data for articles published in academic history journals.

**FILE command**

What kind of file is it? Use the **file** command to find out:

```
file 2014.tsv
```

Try this out on different files on your desktop or in your folder.

**NOTE:** CSV and TSV file formats

`csv` stands for "comma separated value" and `tsv` stands for "tab separated value." When you export an Excel spreadsheet you can choose either of these format; you can also import these from these formats into your Excel program. The separator is also called a delimiter. CSV is probably more familiar to many of you. Can anyone think of a reason to use TSV?

**WC command**

First lets see how big the file is:

```
wc shelldata.tsv
```

wc is the word count command and will output to the screen (standard output) the number of lines, words, and characters in a file.

So, there are 50000 lines of data in this file!

**CAT command**

Let's take a look at what the data looks like. One command you can use is **cat**, which stands for **concatenate**. It both **prints** the file content to the screen and can be used to **join (concatenate)** files together.

```
cat shelldata.tsv
```

This is a good time to learn about the `CTRL-C` command, which cancels the action. So it's a huge file. We don't want to see all of it.   Let's look at just the first few lines.

**HEAD command**

We can use the head command. Without any options, it will display the first 10 lines.

```
head 20NumberedLines.txt
```

But you can specify how many lines head prints:

```
  head -n 3 20NumberedLines.txt OR head -3 20NumberedLines.txt
  head -n 1 20NumberedLines.txt OR head -1 20NumberedLines.txt
```

Can anyone guess what command you would use to see the last 10 or last 3 commands?

Yup, the **TAIL command**:

```
  $ tail -3 20NumberedLines.txt
```

**CHALLENGE:**
Using pen and paper, come up with a sequence of HEAD and TAIL commands to extract lines 6-12 of a 20 line file? (There are more than one way to do this.) Put your answer on the etherpad.

ANS:

```
  head -12 20NumberedLines.txt > output.txt
              # this should give me lines 1-12
  tail -7 output.txt          # this returns lines 6-12
```

**LESS command**
We can also see the output one page or screen at a time, with `less` command:

```
  less Names.txt
  less 100kLines+Labels.tsv
```

We can try this out on a data set. But we need to issue a series of commands which modify the data set. Before we do that, we need to know about **pipes** and **redirection**.

| | |
|---|---|
| `first \| second` | "Pipe" output of first command as input of second command |

|  |  |
|---|---|
| `> file` | Direct output to file |
| `>> file` | Append output into file |

ANS:

```
  head -12 20lines.txt > output.txt
  tail -7 output.txt
```

OR

```
  head -12 20lines.txt | tail -7
  head -12 20lines.txt | tail -7 > output.txt
```

**Finally, two more filters which are useful:**
We'll use <Names.txt> file for this

**SORT command** - sort the input

|  |  |
|---|---|
| `sort` |  |
| `sort -b` | ignore leading blanks |
| `sort -f` | case insensitive |
| `sort -n` | numeric sort rather than alphabetic sort (for numbers) |

`sort -k field1[,field2]` OR `sort - -key=field1[,field2]`
  sort based on a key field located from field1 to field 2 rather than entire line

`sort -t ':'` OR `sort - -field-separator=':'`
  define field separator. (Default separators are spaces or tabs.)

Let's try this out on our names file.

```
  sort Names.txt
```

Now let's sort by profession.

```
  sort -k 3,3 Names.txt
```

Why didn't sort -k 3,3 work?  Sort treated the space as the field separator.

We need to define the field separator as a comma:

```
  sort -t ‘,’ -k 3,3 Names.txt
```

That now works!

**UNIQ command** - report or omit repeated lines.

Let’s create a file to try out the command:

```
nano fruits.txt

apple
pear
orange
apple
orange
apple
pear
```

Write Out, Exit

Some useful options:

`uniq [-u]`    Output uniq lines. This is the default.

```
  uniq fruits.txt
```

What happened?
**uniq** only removes duplicate lines which are adjacent to each other. That’s why it’s usually coupled with <sort>.

**SORT, then UNIQ**

```
sort fruits.txt | uniq
```

We need to sort first, and then run unique because the shell processes files line by line. This is one of the features which allows it to be so fast. It is only working with one line at a time and doesn't have to hold a lot in memory and search through thousands of line. It only compares one line with the next line.

sort fruits.txt > sorted_fruits.txt

| | |
|---|---|
| `uniq -c` | Output a list of duplicate lines preceded by number of times line occurs. |
| `uniq -i` | Case-insensitive. Ignore case during line comparison. |

Try it out with <fruits.txt>

---

**SIMPLE LOOPS**

Syntax for a Unix Shell loops is:

```
for thing in list_of_things
do
  Operation_using $thing   # indentation optional
done
```

For example:

```
$ for file in *.txt    # wildcard * means one or more
> do
>   echo “$file”
>    cp “$file” backup_”$file”
```

```
  > done
```

The variable name in double-quotes accounts for possibility of spaces in filenames. If we don't use double-quotes shell will treat spaces at separators. Safer to use double quotes at all times. (Also safer not to use spaces in filenames!)

Also can do the script in one line, separating commands with a semi-colon:

```
for file in *.txt; do; echo "$file"; cp "$file"
backup_"$file"; done
```

What if you want to use the commands again in the future? —> **Shell Scripts**

---

**SHELL SCRIPT**

Let's start by going back to file <20lines.txt>. First let's create our script.

`nano first_bash_script.sh` # shell scripts end with file extension **`.sh`**

First line contains the Shebang (#!) followed by path interpreter/program which will run the rest of the lines.

Second line is a comment. Comments are prefaced by the hash symbol and are not acted upon by the program. Useful for the Future You.

```
    #!/bin/bash
    # This script loops through .txt files, returns the file
    name, first line, and last line of the file.
    #
    for file in *.txt
    do
         echo "$file"
         head -n 1 "$file"
```

```
        tail -n 1 “$file”
    done
```

We save the file (CTRL-O in nano) and exit the text editor (CTRL-X in nano). You can check that you have such a file and its content with the `ls` and `cat` commands. To run the command, we use the command:

```
bash first_bash_script.sh
```

## ---------- END OF LESSON -----------

## EXTRAS

**FINDING THINGS**

**- - help option** gives information about how to use a bash command

```
sort - - help
```

**man Command** gives the manual page for the given command. May not be available in some versions of Unix. If that is the case, google it.

```
man sort
```

**find Command** – has lots of options and we will not go into it today

**grep command**

GREP stands for "global/regular expression/print".
It looks in files and returns lines of text which contain a given string pattern (or "regular expression")

We will work with the file "chap01.txt" in folder dogbook/ which you can download from my GitHub repo here:
https://github.com/chungkky/2023-09-18_lc_BoC/tree/main/Shell-Data

First, let's look at the contents of "chap01.txt"

```
cat chap01.txt
```

You can see there are seven lines of text.

Say we want to find occurrences of the pattern "dog" in the file. We would type:

```
grep dog chap01.txt
```

GREP prints out the four lines which contain the pattern "dog".

Note, it would help to know the **line numbers** of our results. We can do this by adding the parameter -n, after the word GREP:

```
grep -n dog chap01.txt
```

We now see that "dog" appears in lines 1, 3, 4, and 6.

Notice GREP returned **ALL** occurrences of the string pattern "d-o-g". Not just "dog" but "dogs", "dogma", and "policedog".
If we want to find just the word "dog", we use the parameter -w, for word:

```
grep -w dog chap01.txt
```

We can also **combine** parameters:

```
grep -n -w dog chap01.txt
```

to get the numbered lines where the word "dog" occurs. (By the way, it doesn't matter which order you put the parameters.)

We are not limited to working on just one file at a time with UNIX command lines. For example: our directory <dogbook> has 7 files:

```
ls
```

a JPEG and 6 text files.

A friend points out to us that there are typos in our work. Sometimes, we've typed "policedog" as one word when it is two words, "police [space] dog". We can search through all the text files for the typo at once, using the familiar wildcard “*.txt” for the text files:

```
grep -n policedog *.txt
```

GREP returns four lines in three files: chapter 1, line 6; chapter 2, lines 3 & 6; and the introduction, line 3.

It's always a good idea to look at your results before making any changes to your files (especially anything like a global change). In this case, notice that we would not want to change chapter 2, line 3 because "policedog" here is part of a file name and we do not want to introduce any spaces in a file name.

Finally, you can use more complex **regular expresses** with GREP. For example, we can use the command:

```
grep -n -E '^[A-Z][a-z]* cat' chap01.txt
```

to find where the word "cat" is the second word of a line in <chap01.txt>

The regular expression pattern tells GREP to look for:
# the beginning of a line, caret (^)
# followed by one upper case letter, [A-Z]
# followed by one or more lower case letters, [a-z]*
# followed by space-c-a-t, which matches itself
(and we put all this in quotes so Unix does not attempt to expand any wildcards, like '*', before it runs the command)

The result is line #6, where indeed, "cat" is the second word of the line.

That's a short introduction to GREP.

--------------------------------------------------------------------------------

**Putting It All Together**
(Courtesy of Library Carpentry Shell Lesson.)

We will work on the <shelldata.tsv> file which you can download from my GitHub repo here:
https://github.com/chungkky/2023-09-18_lc_BoC/tree/main/Shell-Data

I'll show you the command and then explain each component of it.

[Remember, here is what one line of the TSV data file looks like:

```
History_3-rdf.tsv       Nostrand, R. L. 3       34      JOURNAL OF THE
WEST    0022-5169        (Uk)ZT002486131 JOURNAL OF THE WEST 34(3), 82.
```

```
(1995)   The Spread of Spanish Settlement in Greater New Mexico: An
Isochronic Map, 1610-1890   xxu            KANSAS STATE UNIVERSITY 1995
```

If we ran a **wc** command, we'll find that it is a huge file with 5000 lines!

```
 wc shelldata.tv
   50000  1728738 12846120 shelldata.tsv
```

Now, here is our command line:

```
grep 2009 shelldata.tsv | grep INTERNATIONAL | awk -F'\t'
'{print $5}' | sort | uniq -c
```

Notice that it is just one line of text!  Let's see what we just did.

```
grep 2009 shelldata.tsv
```

The **grep** command here searches line by line for the string "2009" in the file.

The **vertical bar** is the **pipe** command. It tells the shell to use the output of the previous command as the input of the next command; "piping" the output to another command.

```
grep INTERNATIONAL
```

then searches the resulting lines for those containing the word "INTERNATIONAL" and pipes the output to the next command:

```
awk -F'\t' '{print $5}'
```

**awk** is a powerful search and replace command which we will not go into today but what this part does is tell the shell we have a Tab Separated File (-F, like sort's -t defines the field separator, which in this case is a tab, represented by \t – we will go into this when we look at Regex) and print the 5th column in the file, which contains the journal title.

Next we tell the shell to **sort** the lines.

```
sort
```

Then we tell it to print out only **unique** titles, and a count of how many lines of the same title there are.

```
uniq -c
```

We need to sort first, and then run unique because the shell processes files line by line. This is one of the features which allows the bash shell to be so fast. It is only working with one line at a time and doesn't have to hold a lot in memory and search through thousands of line. It only compares one line with the next line.

We can add one final term to the command sequence. Printing output to the screen is useful for quick checks and demos but often we want to save output to a file so we can reuse it later.

```
> output.txt
```

So, we have the command line:

```
grep 2009 shelldata.tsv | grep INTERNATIONAL | awk -F'\t'
'{print $5}' | sort | uniq -c > output.txt
```

Remember, each record gives the filename of an article and its bibliographic data. The content of the output.txt file is a count of how many records contain both the text “2009" and “INTERNATIONAL” and which journals these records refer to ((i.e. articles published in 2009 and has the word “INTERNATIONAL” anywhere in the record in the record [i.e. in the article title, journal title, or publisher name] and their associated journal title.

```
      2 AFRICA -LONDON- INTERNATIONAL AFRICAN INSTITUTE-
      3 AUSTRALIAN JOURNAL OF INTERNATIONAL AFFAIRS
    106 BAR INTERNATIONAL SERIES
     15 CHINA REVIEW INTERNATIONAL
      8 FOREIGN POLICY -WASHINGTON-
      2 INDONESIAN QUARTERLY
     27 INTERNATIONAL HISTORY REVIEW
     10 INTERNATIONAL JOURNAL OF AFRICAN HISTORICAL STUDIES
      1 INTERNATIONAL JOURNAL OF AFRICAN RENAISSANCE STUDIES
      4 INTERNATIONAL JOURNAL OF CONTEMPORARY IRAQI STUDIES
      3 INTERNATIONAL JOURNAL OF HISTORICAL ARCHAEOLOGY
      1 INTERNATIONAL JOURNAL OF IBERIAN STUDIES
     22 INTERNATIONAL JOURNAL OF MARITIME HISTORY
     16 INTERNATIONAL JOURNAL OF MIDDLE EAST STUDIES
     14 INTERNATIONAL JOURNAL OF NAUTICAL ARCHAEOLOGY
      6 INTERNATIONAL JOURNAL OF OSTEOARCHAEOLOGY
      5 INTERNATIONAL JOURNAL OF REGIONAL AND LOCAL STUDIES
      3 INTERNATIONAL JOURNAL OF THE CLASSICAL TRADITION
      4 INTERNATIONAL REVIEW OF SOCIAL HISTORY
      5 INTERNATIONALES ASIENFORUM
      4 JEUNE AFRIQUE
      5 JOURNAL OF CONTEMPORARY AFRICAN STUDIES
      1 RESEARCH IN INTERNATIONAL STUDIES SOUTHEAST ASIA SERIES
```