Post 1

Nadir Akhtar

October 31, 2017

UNIX and Bash

Unix Penguin Logo

As young incomers to the Computer Science and Statistics worlds, we take for granted how many problems our predecessors overcame to make development and progress much easier. We have the luxury of sharing information and software between machines with ease despite the fact that each machine has a different configuration. From the architecture of the hardware to communication protocols, each computer has its own set of internal rules yet maintains compatibility with others. UNIX was the first step towards a fully universal operation system, and Bash is the software and the language through which we interact with our computer.

Motivation

Imagine as a user having to learn a new set of instructions for every computer you bought or used. That's like having to learn a new set of levers and controls to use when buying a new car: most would not have the tolerance to relearn controls just to use a different machine, leading to inhibited growth due to difficulty of adoption. In such a field like Computer Science, facilitiating development and acceptance of new and improved technologies is crucial to progress.

On the other end, imagine having to design a new set of instructions every time your company releases a model of a computer. Again, that's like having to design a new control system for a car every time that a model is released. It would be much easier both for the user and the manufacturer to have some standard between car models, and even between car companies.

For these reasons, UNIX was made as the first portable operating system between machines. Now, every machine, no matter its architecture, could be run using this operating system. In order to establish a common language to interact with this operating system, Bash was created in conjunction. Theoretically, anyone knowing how to use Bash on one machine can now use *any* UNIX-based machine without retraining, a huge advancement from previous industry standards.

History

In the 1960s, every computer manufacturer had to make a customized operating system (OS) for their particular product. Each computer architecture model requires a unique language for communication, as the hardware has built-in interpreters of certain commands which most likely differ from other computers, paralleling how each human thinks in a unique way that would hardly work directly to another human being without some kind of translation. For this reason, information was siloed within computers, as only a particular computer knew how to interpret its own information.

Dennis Richie at Bell Laboratories, a subset of AT&T, decided to take on this problem by developing his own operating system built on C. As C is a portable language, meaning that code written for one machine can still be used on another, an operating system written using C is also portable, as all machines can execute the C code such that the logic will remain the same. Dennis named this project "unics," which stood for "**UN**iplexed **I**nformation and **C**omputing **S**ystem." "Uniplexed" refers to the aim for commonality across architectures. This acronym turned into "UNIX" over time.

Examples

Basic Commands

One of the most common examples of a bash command is `ls`. This command stands for "list," meaning that the command lists files in a directory. (By default, the current working directory.)

```
ls
```

```
## data
## post_files
## post.md
## post.Rmd
## test
```

One can give arguments to this command to specify what to operate on. The following will perform `ls` upon the `Code` directory.

```
ls test/
```

```
## names.txt
```

One can also add modifiers to adjust the behavior of the command. For example, the `-a` tag will cause `ls` to list *everything* within the directory, particularly hidden elements.

(Note: most elements beginning with a `.` are hidden by default because of the nature of their properties. For example, `.gitignore` is used to specify which files Git ignores but doesn't need to appear unless someone's actively looking for it.)

```
ls -a test/
```

```
## .
## ..
## .gitignore
## names.txt
```

We know how to see contents inside directories, but how do we move around? Well, we have the command `cd` which stands for "change directory." In UNIX, the `~` references the "home" directory (often the user directory unless otherwise specified). Before, we used `ls` with a directory as an argument; how about listing the info in `test/` without using any arguments to `ls`? Let's move into test!

```
cd test/
ls
```

```
## names.txt
```

We can also go up a directory by using the command `cd ..`, where `..` represents the parent directory. (Notice that it appears after running `ls -a`).

```
pwd # print working directory
cd ..
pwd
```

```
## /mnt/c/Users/User/Documents/R/stat133/stat133-hws-fall17/post1
## /mnt/c/Users/User/Documents/R/stat133/stat133-hws-fall17
```

The following versions are all equivalent (assuming that we are in a child folder of `~`):

```
cd ..
cd ~
cd
```

All of the above will return the user to `~`, or "home."

House

We now know how to list information within a director and we know how to move between directories, but how do we actually do anything meaningful with these files?

Below are a few relevant commands:

Command	Action
touch	create a file of this name
vim	see and manipulate the contents of this file
cat	display the contents of this file inside the terminal
echo	display a line of text
grep	search for lines matching some pattern
sed	edit a stream to manipulate text
awk	a “pattern scanning and processing language”

Let’s see what’s in `names.txt` and if we can do anything interesting with it.

```
cd test/
cat names.txt # display the information within this file
echo "Nadir Akhtar" > names.txt
cat names.txt
```

```
##
## Nadir Akhtar
```

This final will replace the contents of names.txt with what is echoed – the > operator means that the StdOut of the echo command is going into the file on the right.

We can use the same logic to wipe the information in that file as well:

```
echo > test/names.txt
cat test/names.txt
```

By echoing nothing, we now overwrite the contents of the file. Neat, huh?

Piping

Last important topic is piping. Bash commands can take in a `StdIn` (Standard Input) input and may, depending on their functionality, produce some `StdOut` (Standard Output). `StdOut` for one function can always be `StdIn` for another by connecting the two commands using the pipe character, `|`.

For example, let's say that I want to search through all directories for some files for all `.txt` files. I can do the following:

```
ls -R | grep .txt$
```

```
## nba2017-stats.txt
## names.txt
```

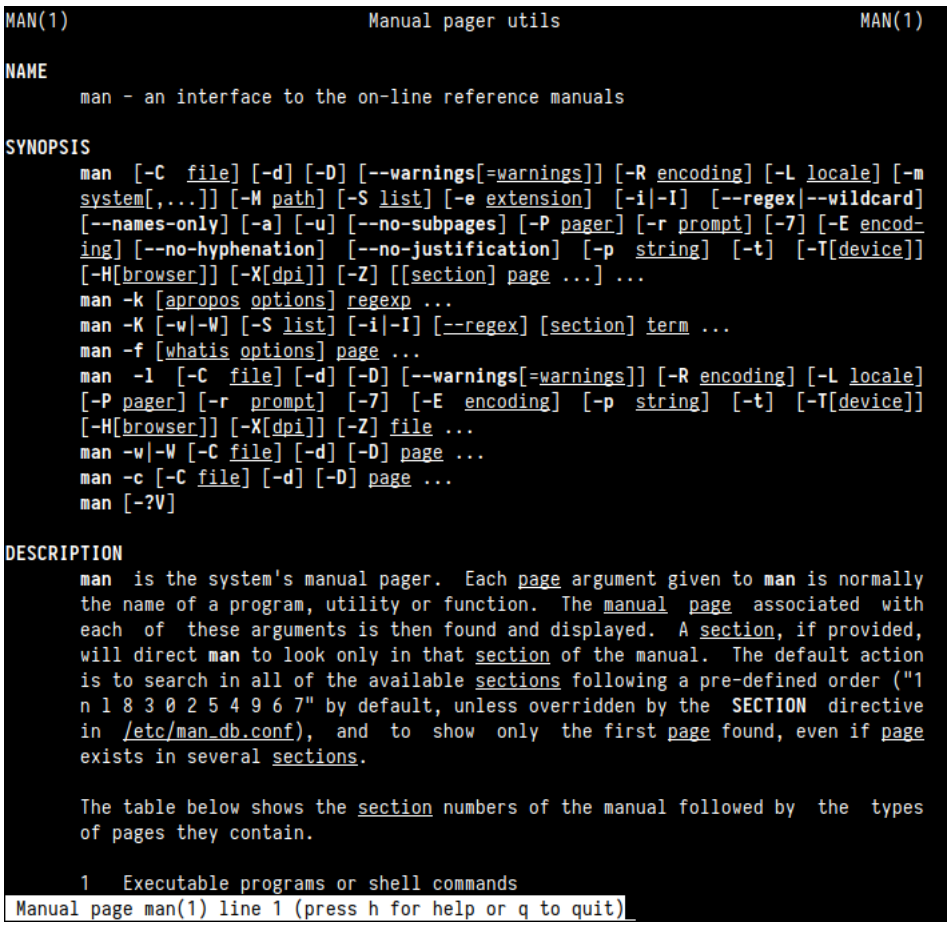
Pipelining is an incredible way to combine simple commands to create a long circuit which is built out of familiar commands. This power of bash scripts allows for the computer to deal with the complicated issues, requiring the user only to think about how the commands work, abstracting away lots of technicality and also allowing users to make subcircuits which can be inserted into larger circuits through further pipelining!

Documentation

The question arises quickly: How does one quickly retrieve information about these various commands? Surely, no one would take the time to remember all this information; that's as inconvenient as relearning a new language.

To answer this question, UNIX has a built-in `man` command, which will allow a user to view built-in documentation about the purpose, syntax, and various "flags" (modifiers) of each of these commands. Below, we see the result of calling the `man` command on the argument `man`, meaning that we will see the manual for how to use the `man` command. (Yes, quite mind-boggling.)

```
man man
```



Man Reference of "Man"

There is also lots of documentation online about UNIX commands, particularly in Stack Overflow.

UNIX + R

Let's say that we have some `data/` folder that we need to sanitize before using because we heard that some evil proprietary mastermind changed all the commmas to colons and all the file extensions to `.txt`. We first list the contents of that folder, then we grep for anything that matches a `.txt` file extension to deal with it.

```
cd data
ls | grep .txt$
```

```
## nba2017-stats.txt
```

We should notice that we have an `nba2017-stats.txt` file. Let's quickly see what's inside:

```
ls | grep .txt$ | xargs cat | head
```

Let's make a small script:

```
cd data/;
for file in $(ls | grep .txt$); do
  newname=$(echo $file | sed 's/.txt/.csv/');
  cat file | sed 's/:/,/g' > newname;
done
ls | grep .csv$ | xargs cat | head
```

```
## "player","games_played","minutes","field_goals_made","field_goals_atts","field_goals_perc","points3_made","poin
ts3_atts","points3_perc","points2_made","points2_atts","points2_perc","points1_made","points1_atts","points1_perc"
,"off_rebounds","def_rebounds","assists","steals","blocks","turnovers","fouls"
## "Al Horford",68,2193,379,801,0.473,86,242,0.355,293,559,0.524,108,135,0.8,95,369,337,52,87,116,138
## "Amir Johnson",80,1608,213,370,0.576,27,66,0.409,186,304,0.612,67,100,0.67,117,248,140,52,62,77,211
## "Avery Bradley",55,1835,359,775,0.463,108,277,0.39,251,498,0.504,68,93,0.731,65,269,121,68,11,88,141
## "Demetrius Jackson",5,17,3,4,0.75,1,1,1,2,3,0.667,3,6,0.5,2,2,3,0,0,0,0
## "Gerald Green",47,538,95,232,0.409,39,111,0.351,56,121,0.463,33,41,0.805,17,68,33,9,7,25,48
## "Isaiah Thomas",76,2569,682,1473,0.463,245,646,0.379,437,827,0.528,590,649,0.909,43,162,449,70,13,210,167
## "Jae Crowder",72,2335,333,720,0.463,157,394,0.398,176,326,0.54,176,217,0.811,48,367,155,72,23,79,161
## "James Young",29,220,25,58,0.431,12,35,0.343,13,23,0.565,6,9,0.667,6,20,4,10,2,4,15
## "Jaylen Brown",78,1341,192,423,0.454,46,135,0.341,146,288,0.507,85,124,0.685,45,175,64,35,18,68,142
```

Excellent! Let's do some ggplotting to celebrate! Let's compare the number of fouls per player to the number of minutes they've played.

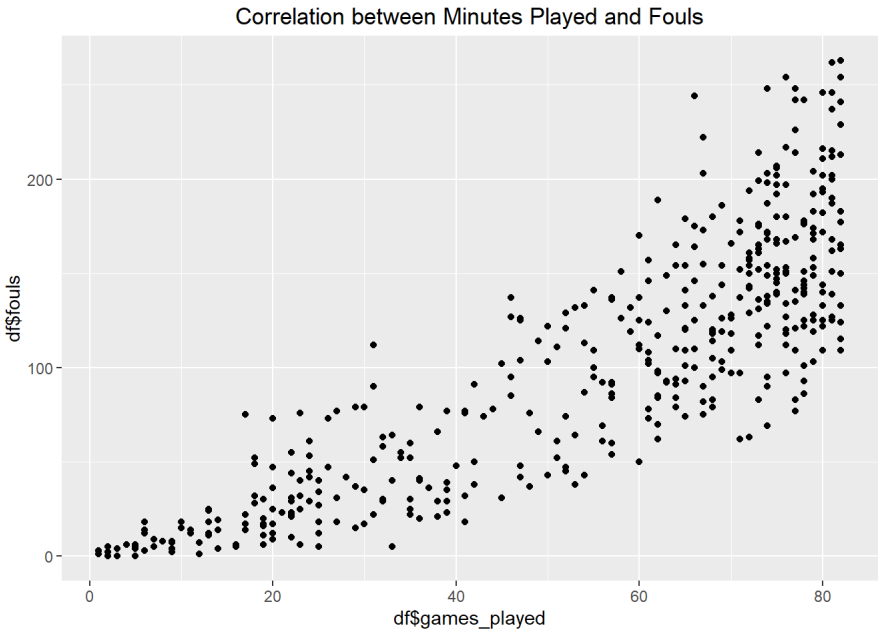
```
library(ggplot2)
library(readr)

df <- read_csv('data/nba2017-stats.csv')
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   player = col_character(),
##   field_goals_perc = col_double(),
##   points3_perc = col_double(),
##   points2_perc = col_double(),
##   points1_perc = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

```
ggplot(data = df) + geom_point(aes(x = df$games_played, y = df$fouls)) + ggtitle("Correlation between Minutes Play
ed and Fouls")
```



Look at that! What a correlation!

Discussion

Bash was required to be a command line interface before the advent of GUIs, which allowed for visual and more intuitive interfaces. However, in spite of these conventionally attractive interfaces, most experienced programmers still opt for using the terminal. Although the learning curve is higher to learn how to use the terminal, the terminal is far more powerful and faster than graphical interfaces. Graphical interfaces are limited by screen space, but the command line is only limited by what is programmable and what the user can remember or quickly access. For more advanced users, using CLI often speeds up their workflow.

Conclusions

UNIX revolutionized the way that computer sciencists and computer engineers approached both design and interoperability. Having the assumption of portability gives software engineers the power to create just one application which can work across any device using UNIX. And as a tool for file management and data analysis, it has more than enough functionality fo A standard language at the computer and the human level, what better way to ensure clear communication and ease of access?

References

A few references to important resources which helped with the development of this post:

- Gaston Sanchez's lecture on Unix and Bash basics: https://github.com/ucb-stat133/stat133-fall-2017/blob/master/slides/07-shell-basics.pdf
- An overview of the history behind UNIX: http://www.engineering-bachelors-degree.com/remote-server-administration-tools/uncategorized/unix-primermotivation-and-unix-environment/
- A graphic depicting the history of UNIX from 1969 to the present: http://opengroup.org/sites/default/files/contentimages/Brochures/unix-info-web.pdf
- A post distinguishing the operating system from the kernel: https://stackoverflow.com/questions/3315730/what-is-the-difference-between-the-operating-system-and-the-kernel
- An overview of using `cd`: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Step_by_Step_Guide/s1-navigating-cd.html
- How to use sed: http://www.folkstalk.com/2012/01/sed-command-in-unix-examples.html
- Pipes in UNIX: https://www.tutorialspoint.com/unix/unix-pipes-filters.htm