Introduction to Unix shell - NOTES

- 2015/16 Part III Systems Biology SEB module
- 10 Feb 2016, 10:00-13:00
- Bioinformatics Training Room, Craik-Marshall Building, Downing Site
- Avazeh Ghanbarian, Alexey Morgunov

Contents

Introduction

Unix shell is a command line interpreter that provides a user interface for directing the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands. In plain English, it is a powerful way of telling your computer what to do. You can read more about the history of Unix shell here http://www.softpanorama.org/People/Shell_giants/introduction.shtml.

Developing skills for coding in any language consists of the following components:

- *Logic* - understanding the syntax, how commands and scripts are structured and how components fit together. This is something one has to learn.
- *Awareness* - knowing what commands, methods and tricks exist and what they can be used for. This is like checking your inventory of LEGO bricks - you need to know what you have in order to start thinking how to put them together to build what you want.
- *Practice* - and a lot of practice. Learning how to combine the bricks together to solve increasingly more complex problems is best achieved through continuous practice.
- *Google and Stack Overflow* *http://stackoverflow.com/* - what coding really is about. It is likely that unless you are doing something very very novel, someone else has run

into the same problem and has a solution. Find it and use it, don't reinvent the wheel. This is an important part of the learning and practice process.

In this tutorial we focus on explaining the *Logic* component and on building some *Awareness* about existing commands and methods in Unix shell. Finally, we give some exercises for *Practice* and leave it up to you to familiarise yourself with how to search for answers if you get stuck.

Important! There are many links to extra resources included in this tutorial. They are for extra information only.

If you have previous experience with Unix shell. Skip to the *Exercises* Exercises.md section and try your skills at it.

Basics

The syntax of commands:

```
[command] -[options] [file or folder]
```

N.B. The angular brackets [] do not need to be typed. They are used here as a placeholder of specific type (e.g. a filename).

Very useful starting points:

```
man [command] #manual entry for the command ('q' to exit)
which [command] #locate the program aliased to the command
whatis [command] #one-line description
apropos [keyword] #match commands with keyword in their man pages
file [file] #reports the type of data contained in file
ls #list files in the directory
ls -l #long information
ls -lh #human readable format
ls -lht #sort by time
ls -A #include hidden files
pwd #print working directory
cd [folder] #change directory into folder
cd ~ #change to home folder
cd .. #move up a directory
```

Working with files and directories:

```
mkdir [name] #create a new directory
cp [file1] [file2] #copy file1 to location file2
cp [file] . #copy file from its location to working directory
mv [file1] [file2] #move file1 to location file2, e.g. rename
rm [file] #delete file
rmdir [directory] #delete directory
ln -s [file] [link] #create symbolic link to file
touch [file] #create file or update timestamp of file
```

Careful when using `rm` recursively (`rm -r`). It is better and safer to use `find` instead, e.g. to remove all files with `.pdf` as their extension in the current working directory: `find . -name '*.pdf' -delete` . (The star in `*.pdf` here means all files that end in `.pdf` .)

Searching for files:

```
find . -name file.txt -type f -print #find in current directory
by name and type (file), print path to file
```

There are plenty more usage examples of `find` - see here http://alvinalexander.com/unix/edu/examples/find.shtml.

Echo is a tool to print to standard output results of a command or just text:

```
echo 'Hello World!' #prints text
echo `ls` #command substitution: prints the output of the
command (backticks!)
echo $HOME #prints the value of the variable
```

File permissions:

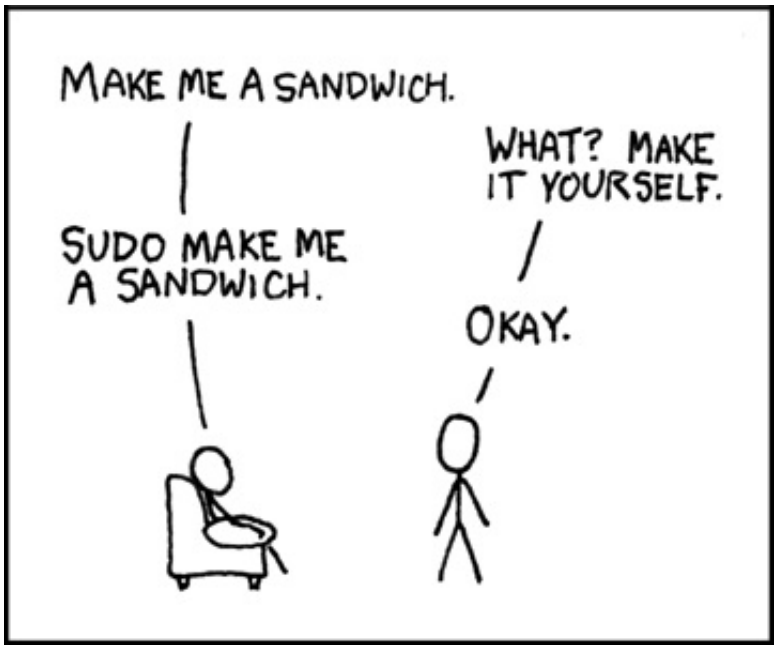
```
chmod 755 [file] #rwx for owner, rx for group and world
```

This works by adding permission codes to make an octal:

- 4 – read ®
- 2 – write (w)
- 1 – execute (x) Specified for owner, group and world, in that order.

There are alternative, non-octal options - see `man chmod`.

Superuser - prefixing the commands with `sudo` gives superuser permissions and requires password input.



Working with text files

Viewing file contents:

```
clear #clear the terminal screen
cat [file] #outputs file contents in terminal window
less [file] #one page at a time, space for next, (q)uit
gedit [file] #open text editor, also 'emacs', 'vi'
head -N [file] #display top N lines of file
tail -N [file] #display bottom N lines of file
wc [file] #word, line, character and byte count
```

Sorting:

```
sort [file] #sort alphabetically/numerically each line from file
sort -u [file] #sort unique entries
sort -r [file] #print reverse order
uniq [file] #print only unique lines
uniq -c [file] #show number of times the line occurs before each line
```

Operations on lines:

```
rev [file] #reverse the characters in each line of file
cut -c2 [file] #cut 2nd character from each line
cut -c3-5 [file] #cut 3rd, 4th and 5th characters
cut -c3- [file] #cut from 3rd character until end of line
cut -d':' -f2,5 [file] #cut the 2nd and 5th fields delimited by semicolons
join [file1] [file2] #join corresponding columns into one
paste [file] #same as `cat` without any options
paste -d',' -s [file] #join all lines in file into one using the delimiter
paste - - < [file] #paste the data in file into two columns
paste -d',:' - - - < [file] #three columns, two different delimiters
paste -d',' [file1] [file2] #join two files by columns, c.f. `join`
paste -d'\n' [file1] [file2] #read lines in both files alternatively
echo "hello" | tr l b #simple by-character substitution of input string (l for b)
```

`join` is a very useful tool - more tricks here http://www.albany.edu/~ig4895/join.htm.

Comparisons:

```
comm [file1] [file2] #outputs 3 columns: lines unique to file1, to file2, common
comm -12 [file1] [file2] #suppress output columns 1 and 2, i.e. show only common
diff [file1] [file2] #shows per line changes needed to make file1 into file2
sdiff [file1] [file2] #compare two files side-by-side
```

`diff` is another overloaded tool, check it out here http://www.computerhope.com/unix/udiff.htm.

`grep`:

```
grep [string] [file] #print lines in file containing string
```

```
grep 'multiple words' [file] #use quotes for phrases
grep -i [string] [file] #case-insensitive
grep -v [string] [file] #lines that DON'T match string
grep -n [string] [file] #show line numbers
grep -c [string] [file] #only total count of matching lines
```

`sed`:

```
sed 's/foo/bar/g' #replaces all instances of 'foo' with 'bar'
```

This is only the simplest and arguably the most useful usage case. There is much, much more to `sed`, start here https://www.digitalocean.com/community/tutorials/the-basics-of-using-the-sed-stream-editor-to-manipulate-text-in-linux and here http://www.thegeekstuff.com/tag/sed-tips-and-tricks/.

`awk`:

```
awk '{print $2,$5;}' [file] #print 2nd and 5th columns from file
awk 'BEGIN { Actions at the start ;} { Actions for every line ;}
END { Actions in the end ;}' [file] #do something, then act on
every line of the file, then do something else
```

`awk` has been called a language within a language. It has a set of useful inbuilt variables and it is very powerful. See more here http://www.thegeekstuff.com/2010/01/awk-introduction-tutorial-7-awk-print-examples/. It can even do its own control flow http://www.thegeekstuff.com/2010/02/unix-awk-do-while-for-loops/.

Redirection & Pipes

To take keyboard input and put it into a file, we can use `cat > file1.txt`. Type as many lines as you like to put into the text file (press `<Enter>` to start a new line) and when done finish with `<CTRL+D>`.

To load the contents of the file, use `cat file1.txt`. To append the contents, e.g. taking contents of a different file `file2.txt` and adding them to the end of `file1.txt`, use `cat file2.txt >> file1.txt`.

To combine (i.e. to concatenate) two files, use

`cat file1.txt file2.txt > long_file.txt`.

N.B. `>` overwrites existing files, `>>` only appends to the end.

To take input from `file1.txt`, sort it and output it as `file2.txt`:

```
sort < file1.txt > file2.txt #using redirection into command, then into file
cat file1.txt | sort > file2.txt #using | to pipe output as next input
```

Using piping, many commands can be joined together, e.g.:

```
cat file1.txt | cut -d',' -f2 | sort -u | wc -l #number of unique entries in second column (as delimited by commas) of file1.txt
```

Wildcards, special syntax and Regular Expressions

To match none or more characters in a file name, a wildcard `*.pdf` can be used, as seen above. Some more examples of wildcards:

```
*ouse #any number or none: matches GRouse, House, Mouse and ouse
?ouse #only one character: matches House and Mouse
^mouse #only at the beginning of line
mouse$ #only at the end of line
```

- *Backslash* works as an escape character:
- *Single quotes* quote everything inside them as is, no need for escape characters.
- *Backticks* work as command substitution.
- *Double quotes* preserve everything except variables and backquoted expressions.
- *Double parentheses* are used for arithmetic operations.
- *Braces* are used for parameter expansion, and also to identify variables unambiguously (among other things).
- *Brackets* are a bit more complicated. *Single brackets* use builtin simple test evaluation, while *Double brackets* are more modern and generally more compatible but not all shells have them. See Shell scripting for more examples and here http://stackoverflow.com/questions/2188199/how-to-use-double-or-single-bracket-

parentheses-curly-braces and here http://serverfault.com/questions/52034/what-is-the-difference-between-double-and-single-square-brackets-in-bash for some discussion.

```
echo $HOME #what you expect, evaluates the variable
echo \$HOME #just a string, escapes the variable
echo \\$HOME #escapes the backslash, evaluates the variable
echo \\\$HOME #if you want both escaped
echo '$HOME | ls' #as is
echo `$HOME | ls` #command substitution, executes and then
displays
echo "$HOME | ls" #variable evaluated, then everything is
displayed without execution
echo $((42+42)) #dollar sign is needed to treat it as variable
echo f{oo,ee,e}d #displays all possible variants
```

The `$IFS` (Internal Field Separator) variable is very important as it determines how shell does word splitting. More info here http://unix.stackexchange.com/questions/184863/what-is-the-meaning-of-ifs-n-in-bash-scripting.

Regular Expressions are sets of characters and/or metacharacters that match (or specify) patterns. It is a world of both wonder and pain. For a brief introduction, if you dare, see here http://www.tldp.org/LDP/abs/html/x17129.html.

Shell scripting

This is where things get real. Shell can be used as a general purpose interpreted scripting language(-ish) with many of the associated features. It is especially powerful for processing text data and passing it between programs, as well as file handling. This it does easier and often faster (citation needed) than Python/Perl/Ruby equivalents.

Shell scripts are usually stored as text files with `.sh` as the extension and starting with a shebang https://en.wikipedia.org/wiki/Shebang_(Unix) consisting of `#!/bin/sh`.

Variables are prefixed with a dollar sign when called, e.g. `$HOME` and assigned with a single equals sign, e.g. `my_variable=42`. To avoid ambiguity in scripts, it is often customary to surround variable names with braces, e.g. `${HOME}`. More on variables here

http://steve-parker.org/sh/variables1.shtml.

Conditional statements:

```
if  [ condition ]; then
 echo "Something"
 elif [ another_condition ]; then
   echo "Something else"
 else
   echo "None of the above"
fi

#example

#!/bin/sh
if [ $1 -gt 0 ]; then
  echo "$1 number is positive"
  else
    echo "$1 number is negative"
fi
```

Note that in the above example, special variable `$1` refers to the first argument supplied when running the script. Save the script to a file named `my_script.sh` and run it using the command `./my_script.sh [number]`.

Case statements:

```
case $variable in
  "pattern1") echo "pattern1";;
  "pattern2") echo "pattern2";;
  *) echo "none of the above";;
esac

#example

#!/bin/sh
echo "car, van, jeep, bicycle or something entirely different?"
read rental
case $rental in
   "car") echo "For $rental £30 per day";;
```

```
    "van") echo "For $rental £50 per day";;
    "jeep") echo "For $rental £60 per day";;
    "bicycle") echo "For $rental £7 per day";;
    *) echo "Sorry, I can not get a $rental for you";;
esac
```

While loops:

```
while [ condition ]; do
  echo "Something"
done

#example

#!/bin/sh
n=$1
i=1
while [ $i -le 10 ]
do
  echo "$n * $i = `expr $i \* $n`"
  i=`expr $i + 1`
done
```

For loops:

```
for $variable in {list}; do
  echo "Something"
done

#example

#!/bin/sh
for ((  i = 0 ;  i <= 5;  i++  )); do
  echo "Welcome $i times"
done
```

For most modern shells, the use of double brackets is recommended instead of single brackets in conditional statements.

There exist too many comparison operators to list here. Use this resource

http://www.tldp.org/LDP/abs/html/tests.html for reference on how to build up conditional statements.

Some further useful things:

```
expr $x +1 #evaluate expression and print results
read [variable] #interactively supply value
printf "%s\t%d\t%f\t%.1f\n" "${string}" "${integer}" "${float}"
"${float.0}" #prints tab separated variables (note the
specifiers) and a new line character
```

Miscellaneous

Compressing files:

```
gzip [file] #creates a .gz archive
gunzip [archive] #unpacks a .gz archive
zcat [archive] #reads a .gz archive without unpacking
tar -cf [archive] [file1] [file2] #create a tarball
tar -xf [archive] #extract the tarball
```

Compiling and running software packages:

```
tar -xf [archive] #unpack the archive
cd [folder] #enter the folder (if necessary)
./configure --prefix=$HOME/[folder] #create the Makefile and
configures installation path
make #build the package
make check #optional: check it compiled correctly
make install #install the package
cd ~/[folder]/bin #go to where the binaries usually are
./[binary] #run the program
```

The following commands and tools are useful ins some specific cases, check out their usage by following the links!

Downloading files:

- `wget` - see here http://www.computerhope.com/unix/wget.htm.

- `curl` - see here http://www.computerhope.com/unix/curl.htm.

Checksum:

- `md5sum` - see here http://linux.101hacks.com/unix/md5sum/.

Calculator:

- `bc` - e.g. `echo '57+43' | bc`, see here http://www.basicallytech.com/blog/archive/23/command-line-calculations-using-bc/.

There are some other things you should definitely know about if you are going anywhere near programming:

Git and GitHub allow version tracking and collaborative development. Don't worry, nobody understands it http://xkcd.com/1597/. But it is still incredibly useful. Start here https://try.github.io/levels/1/challenges/1 and here https://guides.github.com/.

Make is a peculiar little tool that is worth understanding as it can save lots of time and pain by automating very tedious tasks. Check out the shorter http://mrbook.org/blog/tutorials/make/ and longer http://www.tutorialspoint.com/makefile/ tutorials. A good example of what it can do is to automate git processes for a repository to one command `make all` if the following is saved in the `Makefile`:

```
.PHONY: all

all:
 git add --all
 git commit -a -m "pushed through make"
 git pull
 git push
```

Some excellent resources for much more in-depth shell:

- Advanced Bash-Scripting Guide http://www.tldp.org/LDP/abs/html/index.html.
- Short-ish tutorial in bash scripting http://steve-parker.org/sh/sh.shtml.
- Linux Shell Scripting Tutorial (LSST) v1.05r3 http://www.freeos.com/guides/lsst/.
- Linux Shell Scripting Tutorial (LSST) v2.0 https://bash.cyberciti.biz/guide/Main_Page.

This tutorial was written in `Markdown`, which is also a useful thing to learn. Start here https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet.

License