Introduction to Unix shell - NOTES

- 2016/17 Part II BBS Bioinformatics
- 16 Jan 2017, 15:00-17:00
- Bioinformatics Training Room, Craik-Marshall Building, Downing Site
- Alexey Morgunov

Contents

1. Introduction
2. Basics
3. Working with text files
4. Redirection & Pipes
5. Wildcards, special syntax and Regular Expressions
6. Miscellaneous

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.

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* - 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 I focus on explaining the *Logic* component and on building some *Awareness* about existing commands and methods in Unix shell. Finally, I 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* section and try your skills at it.

Basics

The syntax of commands:

```
[command] -[parameters] [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.

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:

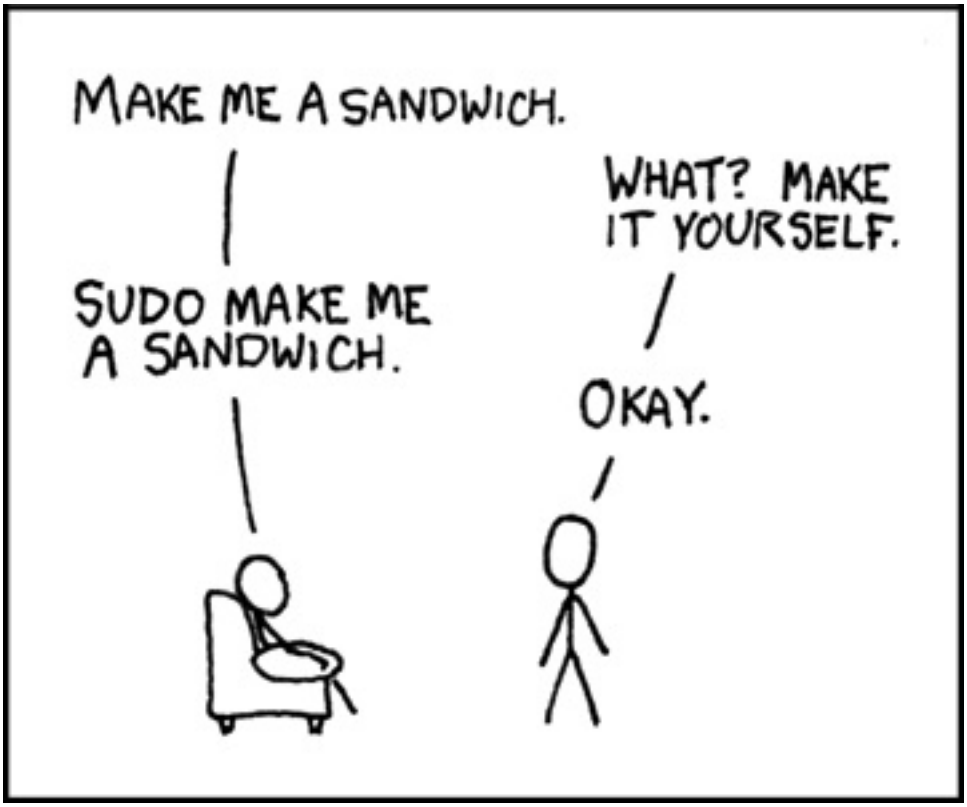
```
chmod 755 [file] #rwx for owner, rx for group and world
```

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
sort -n [file] #sort numerical values
uniq [file] #print only unique lines (detects only adjacent duplicates! sort first!)
uniq -c [file] #show number of times the line occurs before each line
```

Operations on strings:

```
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 r #simple by-character substitution of input string (l for r)
```

`join` is a very useful tool - more tricks here.

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.

`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 and here.

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 and here 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.

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.

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 in some specific cases, check out their usage by following the links!

Downloading files:

- `wget` - see here.
- `curl` - see here.

Checksum:

- `md5sum` - see here.

Calculator:

- `bc` - e.g. `echo '57+43' | bc`, see here.

License

Many of the shell scripting exercises are taken from Linux Shell Scripting Tutorial (LSST) v2.0 under a CC-BY-NC-SA license.

This material is released under a CC-BY-NC-SA license

.