Life in a Shell: Getting the Most Out of Linux/Unix

Thomas Nichols, PhD
University of Warwick

9 October, 2014

Motivation

- Linux is crucial for Scientific Computing
  - Fast clusters use it (buster, minerva, etc)
- Need mastery of the command-line & scripts
  - A command-line environment is excellent for manipulating large numbers of files
  - But without basic skills, the command-line can be slow, result in errors, and can drive you nuts
  - Scripts crucial for efficiency & reproducibility

Long-Term Outline

- Life on the (tcsh) command line
  - Wildcards, pipes, essential commands,
- Basic bash scripting & SGE
  - Variables, if/then, for … do
  - How to submit jobs, manage them
- Intermediate bash scripting
  - Case statements, advanced tests

Life on the Command Line

- Shell basics
- Fundamental commands
- Wildcards
- Input/Output redirect
- Shell variables (local vs. environmental)
- Essential commands

Linux: How do I get there?

- Linux servers (e.g. buster)
  - No "head" with a keyboard/monitor
  - You connect remotely, with ssh "**S**ecure **Sh**ell"
- Windoze: Obtain terminal program & ssh
  - E.g. `putty` ssh client and terminal
  - E.g. `Cygwin` – Linux command line suite
    - Still need a terminal; use Dos command window, or mintty
    - Then ssh to linux host
- MacOS: Ready to roll!
  - "Terminal" terminal program & ssh installed
    - Bash shell by default
    - ssh to other Linux hosts

EXERCISE

- Log in to buster

  buster.stat**s**.warwick.ac.uk

don't forget the ***s****!*

- From a terminal command line (itself a shell!)

```
ssh <YourUserId>@buster.stats.warwick.ac.uk
```

  – E.g. my userid is **`essicd`**
  – Yours probably looks like **`str???`**

Shell Basics

- The Shell
  - Just another program
    - Accepts your keystrokes
    - Sends corresponding character to terminal
    - Runs programs on your behalf
  - But shells are *also* scripting language
    - Text file lists of commands
    - Complicated, looped, conditional programs

Shell Basics

- Different types of shells
  - sh “Bourne Shell”
    - Written by Steve Bourne at Bell Labs, c. 1974
    - Not so friendly on command line
    - On linux, now same as bash
  - bash “Bourne-Again Shell” *what we'll use!*
    - More friendly on command line
    - Regarded as best-practice scripting shell language
  - csh “c-shell”
    - Former standard shell
    - On linux, now same as tcsh
  - tcsh “Enhanced c-shell”
    - Enhanced c-shell, with tabular completion

Which Shell to Use?

- Interactive, on command line
  - bash
    - Most common; previously, tcsh was dominant
    - It’s the default
    - Changing the default is hard
- For scripting
  - bash
    - functions
    - Extensively used in FSL, elsewhere
    - See “Csh Programming Considered Harmful”

File Paths

- Hierarchical directory tree

`/`	“Root” directory
`/tmp`	Temporary files
`/home`	User files
`/etc`	System configuration files

Forward slash /
not
Backslash \

- Special directories

`.`	(period) references current directory
`..`	(period^{2}) references parent directory
`~`	Your home (& initial) directory
`~user`	Another user's home directory

Fundamental Commands

- `pwd` “Print working directory”
  - You are always *somewhere*
- `cd` “Change directory”
  - E.g. `cd ..` (go up one directory)
  - E.g. `cd ~/tmp` (go to my personal temp dir)
  - E.g. `cd ../../duh` (go up 2 dirs, then duh)
  - E.g. `cd ~` (go to your home directory)
  - E.g. `cd` (same)

Filenames

- Essentially no limit on filename length (256)
  - Though best to keep it reasonable <20 char
- Extensions meaningless to Linux itself
- But use them for humans' sake
  - Text files `.txt`
  - Data file `.dat` (generic)
  - Data file `.csv` (Comma separated)
  - Shell script `.sh` (bash/sh)
- Best to *not* use extensions in directory names

Command Parsing

- Each line entered broken into **white-space separated tokens**
  - White space = 1 or more space or tabs
  - E.g. `cd/to/my/directory` **Only 1 token!**
  - E.g. `cd /to/My Programs/Desktop` **3 tokens!**
- First token is the command
- Remaining tokens are arguments to the command
  - E.g. `cd /to/my/directory`
    - "cd" first token, the command
    - "/to/my/directory", argument for command “cd”
  - E.g. `cd "/to/My Programs/Desktop”`

Copy & Paste Danger!
Smart quotes don't work!
‘ ’ “”

Must use plain quotes ' "

Command Parsing: Escaping & Protecting

- How to deal with spaces?

1. Don’t use them in file or directory names!!

2. **Escape** them, with backslash (\)

   E.g. `cd /to/My\ Programs/Desktop`

3. Protected them with quotes (' or ")

   E.g. `cd "/to/My Programs/Desktop"`

   E.g. `cd '/to/My Programs/Desktop'`

   - (more on single- vs double-quotes later)

Special Characters

These must be escaped or quoted to avoid their *special* meaning:

```
!  #  $  &
'  "  (  )
{  }  *  +
-  .  |  \
;  &  ~  ?
<  =  >  @
[  ]  ^  ␣
         (space)
```

There are more!

Command Parsing: Options

- Arguments vs. Options
  - Convention has it that optional arguments are preceded by a minus sign
  - E.g. `ls` (Show contents of current dir)
  - E.g. `ls /tmp` (Show contents of /tmp dir)
  - E.g. `ls -l /tmp` (Show detailed contents)

Fundamental Commands

- `ls` “List files”
  - E.g. `ls` (list files in current directory)
  - E.g. `ls .` (same)
  - Optional Arguments
    - `-l` (minus ell) Long listing, showing date, size
    - `-a` Include files beginning with . (dot)
    - `-t` Order by time of last modification (best w/ -l)
    - `-d` Do not list subdirectory contents
    - E.g. `ls /home/essicd`
      Shows contents of the directory
    - E.g. `ls -d /home/essicd`
      Shows info on the directory itself

Fundamental Commands

- `mkdir` <dirname>
  - Create a directory
- `rmdir` <dirname>
  - Remove a directory; must be empty
- `rm` <file>
  - Remove files
  - Optional Arguments
    - `-i` Interactive – ask if you're sure for each file
    - `-r` Recursive, delete directories and conents

Fundamental Commands

- `cp file1 file2`
  `cp file1 file2 file3 … directory`
  - Creates a copy of a file *(first form)*
  - Copies one or more files to a directory (second form)
  - Optional Arguments
    - `-i` Interactive, warn about over-writing
    - `-r` Recursive, copies directories and contents
    - `-p` Preserve file modification times (otherwise timestamp on new file is now)
- `mv file1 file2`
  `mv file1 file2 file3 directory`
  - Renames a files (i.e. "moves" it) *(first form)*
  - Moves one or more files to a directory *(second form)*
  - Optional Arguments
    - `-i` Interactive, warn about over-writing

EXERCISE

- Create two directories in your home, tmp & bin

```
mkdir tmp bin
```

- Copy my demo files to your tmp directory

```
cp -rp ~essicd/Sandbox ~/tmp
```

- Use the "touch" command to create some empty files, "ls" them, then delete them

```
touch duh
ls -l
rm duh
ls
```

Wildcards

- The shell "expands" each token, replacing "wildcards" with matching filenames
- `*` Matches any string
  - E.g. `ls *.txt` ...all files ending .txt
- `?` Matches any single character
  - E.g. `ls file?.txt`
  - E.g. `ls file??.txt`
- `[...]` Matches any one of the enclosed characters
  - E.g. `ls file[12].txt` matches `file1.txt` and/or `file2.txt`
  - E.g. `ls file*[12].txt` ... any file beginning `file` and ending `1.txt` or `2.txt`

EXERCISE

- In the `Sandbox` directory, try these wildcards… guess first what you think the outcome should be

```
ls -1 file*txt
```

… "*minus one*" all files in a column

```
ls file?.txt
ls file??.txt
ls file[12].txt
ls file*[128].txt
```

- Do you understand why you got each!?

Brace Expansion

- Wild cards will match if possible
  - Incomplete match no mention
  - No match at all, error
    - `ls file[128].txt` -> gives

      ```
      file1.txt file2.txt
      ```

    - `ls file[48].txt` -> gives

      ```
      ls: file*[48].txt: No such file or directory
      ```

- Braces { } do "hard" expansion (not "if possible")
  - Comma separated list of strings
    - `ls file{1,2,8}.txt` -> gives

      ```
      ls: file8.txt: No such file or directory
      file1.txt  file2.txt
      ```

      Just the same as typing `ls file1.txt file2.txt file8.txt`
    - `ls file.{dat,csv}` -> Incredibly useful case

Fundamental Commands

- `more` Show file, one screen at a time
- `head` Show first few lines of a file
- `tail` Show last few lines of a file
  - For both `head` & `tail`:
    - `-n #` Show # lines instead of default (10) num.
      e.g. `head -n 20 file.txt`
  - For `tail` only:
    - `-f` Show last 10 lines, then wait for file to grow, and show new lines as they appear
- `cat` Con***cat***enate files
  - E.g. `cat file1.txt file2.txt file3.txt`
- `wc` Line, ***w***ord & character ***c***ount
  - E.g. `wc file.txt`

EXERCISE

- In the Sandbox directory, use `more`, `head`, `tail` & wc to examine the (very long) `rgb.txt` file
  - What's the name of the first color?
  - What's the name of the last color?
  - How many colors are there?

In case you forget what a command does…

```
    man <command>
```

Gives help or **man**ual page for given command.

Essential reference, but not always so helpful. Instead, try Googling the command plus "unix".
E.g. Google "cat unix" (instead of Googling "cat")

Output Redirection

- Each program in Linux has three modes of input/ output
  - Standard input (normally what you type at the keyboard)
  - Standard output (normally what spits out on terminal)
  - Standard error (also dumped out on terminal, interleaved with stdout)
- Shell can redirect input or output

`<` Standard **input from** file (overwrite)

`>` Standard **output to** file (overwrite)

`>&` Standard **output** *and* **standard error to** file (overwrite)

`>>` Standard **output to** file – append

`>> file.out 2>&1`

Standard **output** *and* **standard error** to file – append (don't ask)

Output Redirection

- Can use alone or in combination

  E.g. `ls -l > ~/tmp/Listing.txt`

  E.g. `matlab < myscript.m >& myscript.log`

  * There are better ways of doing this!

- Pipe |
  - Connecting stdout of one program to stdin of another

    E.g. `ls -l | more`
  - Can have several pipes

    E.g. `ls -l | tail -100 | more`
  - Especially useful with "grep" command, a filter

    E.g. `ls -l | grep duh | more`

Output Redirection

- Useful Examples
  - Save directory listing

    -
      ```
      ls -l > FileList.txt
      ```
  - Look at long listing page at a time

    -
      ```
      ls -l | more
      ```
  - Look at the 10 most recently modified files

    -
      ```
      ls -lt | head
      ```
  - Look at only the 100 oldest files

    -
      ```
      ls -lt | tail -n 100 | more
      ```
  - Concatenate a bunch of files into a new one

    -
      ```
      cat file1.txt file2.txt > allfiles.txt
      ```

EXERCISE

- Save a long listing (`ls –l`) to a file; examine the file with `more`. Delete the file when you're done.
- Save the last 20 colors in `rgb.txt` to a new file, `rgb_last.txt`. Examine them with `head` (what is the 19th-to-last color?)

Shell Variables

- Behavior of the shell is modified by "shell variables"
- **Assign** variables with equal sign =

```
NextSim=Prog4
```

- **Dereference** with dollar sign $

```
echo $NextSim
```

  … just shows "`Prog4`"

- Protect dereferencing with brackets

```
echo $NextSim_1
```

  …no output, variable "NextSim_1" is undefined

```
echo ${NextSim}_1
```

  … shows "`Prog4_1`"

The simplest shell command: `echo`
Just 'echoes' the command line

Vital Shell Variables

- `USER`
  - Your user name
- `HOME`
  - Your home directory, same as ~
- `PS1`
  - Prompt string. Try…

```
PS1="Your wish is my command> "
```

Shell Variables: Local vs Global

- **Local variables** do not get passed on to child processes

```
NextSim=TestProg
bash
```

Start a new shell! Yes, you can do that any time.

```
echo $NextSim
```

… no output

- **Global variables** passed to 'child' processes
  - Mark global variable with "export"

```
export NextSim=TestProg
bash
echo $NextSim
```

… shows "TextProg"

  - By convention (only) global variables are capitalised

Most Important Shell Variable

- `PATH`
  - Colon-separated list of directories

```
echo $PATH
```

… might show something like

```
/usr/local/bin:/usr/bin:/bin
```

  - These are the directories searched when you type a command.
  - If you type "`ls`", the shell will first look in /usr/local/bin for a program named "`ls`" and then run it; if it isn't there, it will look in "`/usr/bin`", and then "`/bin`", etc.
  - Finally, if it doesn't find it, you get "`bash: ls: command not found`"

EXERCISE

- The `env` command shows all variables; try:

```
env | more
```

- Examine your `PAT`

```
echo $PATH
```

- Compare these two commands

```
echo "My user name is $USER"
echo 'My user name is $USER'
echo "My user name is '$USER'"
echo 'My user name is '$USER
```

Shell Variables and Double vs Single Quotes

- Both types of quotes protect white space and special characters. But single quotes protects *more* special characters.
- **Partial Quoting** with Double quotes (")
  - Protects white space, and most characters, but allows shell variables, e.g. $USER, to be expanded into their value.
- **Full Quoting** with Single quotes (')
  - Protects all special characters including $
    - So no shell variable expansion

Setting Shell Variables Permanently

- Configuration Files

  `~/.profile`
  Run each time you **log in**

  `~/.bashrc`
  Run each time you start a new **interactive shell**
- Login-only?
  - E.g. when SGE runs programs on your behalf
- Interactive shell?
  - E.g. whenever you ssh, or start a new shell with "bash"
- Change your `PATH` in `.profile`
- Change command-line goodies in `.bashrc`
  - e.g. `PS1`, aliases

EXERCISE

- File editing practice
- Use `nano`, a simple text edit that works in a terminal (no graphics!)
  - `nano test.txt`
  - Write some text
  - Save with `^o` (specify name, press [return])
  - Exit with `^x`
- Other useful nano commands
  - ^K "cut line"
  - ^U "uncut line"

Convention for Describing Keyboard Shortcuts: "^X" means "Control+x"

Most keyboard shortcuts in Linux consist of holding the [control] key while pressing another key.

By convention this is denoted by a up-caret (^) and the character – in capitals (as it appears on the keyboard). ^X *does not mean*, [control]+[shift]+x

bash aliases

- Best way to make shortcuts for frequently used commands
  - Instead of every day typing

    ```
    cd /storage/myid/very/long/path/to/my/project
    ```

  - You could type

    ```
    cdmyproj
    ```

  - Syntax

    alias <AliasName>=<Command>

    E.g. `alias cdmyproj="cd /storage/myid/very/long/path/to/my/project"`
- Quiz!
  - Where should you add aliases, to `.profile` or `.bashrc`?

Essential Aliases

- IMHO, everyone should have these 3 aliases

```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

- Prevents you from accidently overwriting a file
- What if you *do* have lots of files to delete? Make a special "really delete" command

```
alias trash='rm –f'
```

Editing Configuration Files SAFELY!

- Editing .profile and .bashrc is **dangerous**!
  - If you introduce an error to .profile, you might not be able to log in!!
  - Be careful! Always use two terminal windows!
- Terminal Window 1
  - Make a backup-copy
    - `cp .bashrc .bashrc_safe`
    - `cp .profile .profile_safe`
  - Open a text editor; make edit to .profile/.bashrc
- Terminal Window 2
  - After making edit, try running a new shell
    - `bash`
  - **ALSO**, log out, and try logging back in
    - `exit`
    - `ssh buster.stats.warwick.ac.uk`
- If you *can't* login or get errors
  - Fix them *before* closing the editor and Terminal 1!!!
  - Worst case, restore safe version
    - `cp .bashrc_safe .bashrc`
    - `cp .profile_safe .profile` … and double check can run bash and login!!!

"Power User" Terminal Text Editors

emacs – Hard to learn, but incredibly powerful. Can be endlessly modified (using lisp-based configuration files)

vim - Emacs' arch enemy. Don't use. ☺

EXERCISE

- Using the safe method for editing .bashrc, add the rm, mv and cp aliases.

```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

- Be sure to use the 'safe' editing method!

EXERCISE

- Add your personal binary directory `~/bin` to your path in `~/.profile`, with this line

```
export PATH="$HOME/bin:$PATH"
```

- Crucial details!!!
  - Must use quotes, in case existing path has white space in it
  - Must *add* to existing path
    - If you simply did

```
export PATH=$HOME/bin
```

don't use this!!!

… your shell would break; no `ls` or any command!!

Process Control - Intro

- The essential aspect of Linux/Unix is that it multi-user, multi-tasking
- The Linux host is always doing more than 1 thing
- You can do more than one thing at once with the shell!

Process Control: Foreground vs Background

- Commands can run in "foreground" or "background"
- Foreground
  - The default; shell does nothing, waits for command to finish
  - Use `^c` to terminate the current foreground job
- Background
  - A job started in the background returns control to the shell, let's you do other things
  - Once running in the background, you can manipulate the job (e.g. suspend it, kill it, etc)

Process Control: Starting a job in the background

- Use & (ampersand)
  - E.g. `MyLongJob &`
    Shell will return two numbers, one in brackets, e.g.

    ```
     [1] 5764
    ```
  - Number in brackets is the short hand job id, relevant to your shell only
  - Other number is process id, the unique identifier for the entire system

Process Control:
Put a foreground job in background

- Start a job in the foreground (i.e. as usual)
  - E.g. `MyLongJob` *(no ampersand!)*
    Shell waits for job to finish
- Suspend it with `^Z`
  - This stops (does not kill) a job; shell says, e.g.

```
[1]+  Stopped                 MyLongJob
```

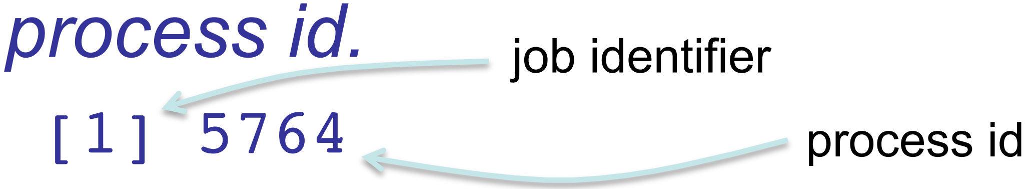
- Put it in the background with `bg`
  - The job resumes, but you get the shell back

```
[1]+ MyLongJob &
```

- Show all background jobs with `jobs`

Process Control: Manipulating Background Jobs

- Kill a background job
  - You must know the either the *job identifier* or *process id.*



```
[1] 5764
```

job identifier

process id

  - Job identifiers require a % prefix
  - Use `kill` command

    `kill %1` same as `kill 5764`

  - This is the same as ^C. Sometimes it doesn't work, then must do

    `kill -9 %1` same as kill -9 5764

Process Control: Killing Me Softly

- Some processes know how to die gracefully
  - E.g. Most text editors will do an auto-save then die
- But `kill -9` prevents any such gentle death!
- Use `kill -HUP` to send a "hang up", and trigger a (hopefully) organized termination
  - `gedit MyThesis.tex &`
  - `kill -9 %1` Nothing saved!
  - `gedit MyThesis.tex &`
  - `kill -HUP %1` Hopefully something saved

EXERCISE

- In Sandbox, run my test program

```
./SlowProg.sh
```

- Kill it with `^C`
- Run it again, with its output sent to a file

```
./SlowProg.sh > SlowProg.log
```

- Suspend it with `^Z`
- Put it into the background with `bg`
- Look at the output with `tail -f`

```
tail -f SlowProg.log
```

- Kill tail (from foreground) with `^C`
- Kill `SlowProg.sh` from background with kill

```
kill %1
```

- How long did it run?
- Try this again, but this time use `&` to start it in the background
  - What happens if you do: `./SlowProg.sh &` (i.e. don't use redirection)?

SGE Preview

- "buster" isn't the server host itself, but just the "head node"
- It is impolite to run anything but short, 'test' jobs on buster
- To run jobs interactively use

  ```
  qlogin
  ```

  That will connect you to a "compute node"
- Queueing non-interactive jobs is the next topic

Supplementary Material (covered next time in depth)

EXERCISE
Basic Scripting Preview

- Try these examples (in `Sandbox`) and see if you can figure out what they do

```
grep blue rgb.txt | head
grep blue rgb.txt | wc
grep -i blue rgb.txt | wc          what does -i do?
sed 's/imgdir/dir/' file.txt
sed 's/i/X/' file.txt          and compare to…
sed 's/i/X/g' file.txt          what does /g do?
awk '{print $2}' file.txt    and also try $1 instead of $2
awk -F, '{print $2}' file.csv    also try dropping the -F,
awk -F, '(NR>1){print $2}' file.csv
awk -F, '($1<10){print $0}' file.csv
```

The Holy Trinity

- grep
  - Prints lines that match general regular expressions
- sed
  - Stream editor
- awk
  - A full programming language, brilliant for handling structured data files (e.g. tab or comma separated)

grep

- grep <pattern> <files>
  - Will print all lines in files that match the pattern
  - Key options
    - `-i` Ignores case
    - `-l` Only print file name when a match found
    - `-v` Print lines where match does *not* occur
    - `-n` Show line number where match occurs
    - `-r` Work recursively
- Ex: What aliases do I have?
  - `grep alias ~/.bashrc`

sed

- sed <command> <files>
- There is basically only kind of command you'll use, the "search" command
  - `sed 's/dir/DIR/' file.txt`
  - `sed 's/dir/DIR/g' file.txt` <- Use global option
  - `sed 's/dir/DIR/g;s/img/hdr/' file1.txt` <- stack commands w/ ;
  - `sed -n p3 file.txt` print only 3rd line of file

awk

- Basically a full programming language
  - Not really advised… see Python
- However, indispensible for text processing
- Line-by-line processing
  - Each line broken into tokens
    - By default, white-space-separated tokens
    - Change "Input Field Separator" with `-F` option
      - Most typically `-F,` to work with CSV files
  - Each element accessible with $1, $2, etc.
    - E.g. `awk -F, '{print $2,$1}' test.csv`
      …print 2nd and then 1st entry of CSV file
  - $0 means "The entire input line"

awk

- Commands must be in braces {}'s.
  - Braces are special shell characters, so **always must single-quote** (`'`) awk commands
- A conditional test can precede the commands
  - E.g. `awk -F, '($1>10){print $0}' test.csv` …print entire line only only if first value > 10
- Special variables
  - NR tells you the current row number
    - E.g. `awk -F, '(NR>2){print $0}' test.csv` …print entire line only for 3rd and greater lines
  - NF is the number of fields in the current line
    - E.g. `awk -F, '{print $NF, $(NF-1)}' test.csv` …print last field (`$NF`) and then next-to-last field (`$(NF-1)`)

Other Important Commands

- `man` Show "manual" pages
  - Gives (some) help on commands
- `sort`
  - Key options
    - `-r` Reverse the order of the sort
    - `-n` Try to sort numbers correctly (e.g. 2 < 10)
- `du` "Disk usage"
  - Key options
    - `-s` Silent, only report summary
- `df` Show file system usage

Very Basic Shell Scripting

- Making a script
  - Make sure you have a `~/bin` directory
  - Make sure `~/bin` directory is in your path
  - Create your script in `~/bin`

    ```
    emacs ~/bin/myscript.sh
    ```

    First line must be

    ```
    #!/bin/bash
    ```
  - Make it executable

    ```
    chmod +x emacs ~/bin/myscript.sh
    ```
- Magic!!!
  - Now anytime, anywhere that you type `myscript.sh` it will run!

.sh extension

There is no requirement to use .sh extension on shell scripts.

I like the convention, as it reminds me what is a script and what isn't. (e.g. vs. .R .m etc)

Special Variables in Scripts

- Command line "positional" arguments
  - `$0` Name of the program run
  - `$1` Frist argument, `$2` second argument, etc.
  - `$#` Number of arguments
  - `"$@"` All arguments
    - Later we'll see that the quotes important to deal with white space correctly

```
#!/bin/bash

echo "This is the program name: $0"
echo "There are $# args"
echo "This is the first arg: $1"
echo "All args: $@"
```

Looping

- For loops

```
for <var> in <a list of stuff>  ;  do
     command1
     command2
done
```

- Most typically over arguments…

```
#!/bin/bash

for f in "$@" ; do
  echo "This is an argument '$f'"
done
```

Integer Arithmetic

- Bash can natively handle integer variables and do simple arithmetic
- Double parenthesis mark "math mode"

  `((i=1+2))` … but if just assigning, no need for (( ))… `i=1`

  `((j=3))`

  `((k=i+j))`
- Special for loops available for math mode

```
#!/bin/bash

n=10
for ((i=n;i>0;i--)) ; do
  echo -n "$i "
done
echo "Lift off"
```

Bash Functions

- Essential for scripts and command line

```
functname() {
   Commands
}
```

- I have 2 shell functions I can't live without

```
lsh() {
     ls -lat "$@" | head
}
lsm() {
     ls -lat "$@" | less
}
```

  - What do these do?!
  - Are they in my .bashrc or .profile?