Controlling the World (with Computers): Introductory Linux Recipes, for Physicists

This is designed to give a flavour of how to use Linux/CLI, and what you can do with it, in ~ 2 hours, from scratch. The aim is to provide a jumping-off point, for demonstration and further exploration. Recipes are brief; see The Internet for elucidation. Written by Richard Neill (rn214), 2015-17, for his IA Nat-Sci Students at Magdalene Cambridge.

Contents: Command line – Files – Bash – Regexps – More – Networking – Mail – WWW – C – SQL – Git – Security.

Introduction

All scientists need to know how to control a computer. This is useful for data analysis (sometimes you have to write your own tools), controlling experiments (especially repeated ones with automated hardware), and general wizardry. Most inventions now have some element of software. It's also the case that "*the Geek shall inherit the Earth":* first-rate programmers can earn a *very* decent salary, while wielding great influence for good, and having *fun* doing it.

It is more likely that you will be productive if you are familiar with *Linux* than if you try to proceed with a consumer-only system such as OSX or MS Windows. Recommended: download and install *Ubuntu*, (probably in the XFCE version, for 64-bit systems named *Xubuntu*): it is *free* from xubuntu.org , and you can try it out from a USB key. Another system (designed for temporary use, directly booted from a USB) key is *Knoppix* from www.knoppix.net .
You can run most of these examples in a remote-shell. Text in `green` is a command to be typed; *italic* text like `the_file` or `user123` should be substituted.

Books and Resources

www.richardneill.org/teaching/	This document, and the previous term's Electronics (particularly Digital Logic).
www.cl.cam.ac.uk/teaching/1112/UnixTools	Markus Kuhn's **Unix Tools** course for the Cambridge Computer Laboratory.
linuxcommand.org	Introductory guide to the Linux command line
www.tldp.org/LDP/abs/html	Advanced Bash-Scripting Guide (very detailed, includes many examples)
ubuntuguide.org	A Guide to General Linux use (based on Ubuntu)
www.w3schools.com	Tutorials on programming for the Web (HTML, CSS, Javascript, etc)
www.php.net	The PHP (web) programming language (well documented)
www.catb.org/jargon/html	The Jargon file – a guide to hacker-culture and terminology. Entertaining.

Pre-requisites

A **Laptop** (with Wi-Fi, and a recent version of *Firefox* or *Chrome*). (Any operating-system will do, Linux ideally).
A **Linux server**: in Cambridge, we'll use the Student-Run Computing Facility, *SRCF*. Get a free **account** at www.srcf.net .
An **SSH** (secure shell) client. On Windows: get *PuTTY*, free, www.putty.org . Download *putty.exe* : the 1st link in Binaries.

Also useful:
An **SCP** (secure copy) client on your machine. Use the *FireFTP* addon for *Firefox*, the *SFTP client* for *Chrome*, or *WinSCP* (from winscp.net).
A **Text Editor** such as Notepad++, free from notepad-plus-plus.org . (A text-editor is for programming; it is *not* the same as a word-processor).
[Outside the scope of this course: note that the *Fink* project packages many Linux/Unix applications for MacOSX, while *Cygwin* does so for MS Windows.]

Terminology

Unix:	Operating system, designed in the 1970's (and still going strong) for *simplicity*, *elegance*, and *compactness*. *Unix*, *C*, and the *Internet* were born together. ← a pun on Multics (Multiplexed Information and Computing Service).
***nix:**	Any of the Unixes: *Solaris*, *AIX*, *OpenBSD*, *FreeBSD* (Berkeley Standard Distribution), *POSIX*, *HURD* *Darwin* (the core of MacOS X), *Linux*, *Android, ...*
GNU:	The GNU project is a free-software implementation of Unix. ← GNU = GNU's Not Unix (pronounced *G-nu*, not *Noo*).
Linux:	Linus Torvalds' Operating System (kernel), often generalised to mean the entire system of GNU/Linux.
Source-code:	The human-readable/editable form of software, which is then compiled to binary "object code" for the CPU.
Free-Software:	Free as in *Freedom*, share-alike. Copyleft. Software-Libre. See: GNU **GPL**. ← www.gnu.org/philosophy
Open-source:	An *engineering methodology* of open-development. Usually also Free-Software. FOSS or F/LOSS.
Distribution:	A collection of Linux+GNU+X+... E.g. *Debian*/*Ubuntu*/*Mint*/*Knoppix*, *RedHat*/*Fedora*/*SuSE*/*Mageia*, *Arch*.
CLI vs **GUI:**	Command-line (textual) interface vs Graphical User Interface. ← CLI is often referred to as the shell, terminal, or bash
Jargon:	See The Jargon File for explanations, context and history. ← catb.org/jargon

Linux is now everywhere: from most embedded devices (routers, modems, network printers) and the Internet of Things, to the majority of smart-phones (as *Android*), in education (*Raspberry Pi)*, the majority of servers, *CERN*, the *London Stock Exchange*, *Pixar*, *Facebook*, 494 of the World's top 500 supercomputers... ["Open source has won." - Martin Fink, CTO, HP]
Notable Free/Open-Source Software includes *Firefox*, *LibreOffice*, *Apache* webserver, *PostgreSQL* and *MariaDB* databases, *VLC*, *GCC*, *Busybox*, Perl, *Arduino*, and *Exim* (our own Cambridge email system, *Hermes*).
Free Software (as in Freedom, not just as in Beer) is important. See: youtube.com/watch?v=aQyZ5M96_CA (3 minutes).

The Shell: the Unix Command-Line Interface

The Linux command-line is exceptionally powerful. Because everything is text, the interface is simple, fast, predictable, and, most importantly, *scriptable.* It's very easy to repeat and automate complex processes, or to daisy-chain the output of one command into another. Each tool is designed to do *one simple thing well*, and for *reuse.* Most GUI tools are based on their CLI counterpart. Unix is expert-friendly; it is designed to make easy things fast, and hard things possible.

The shell is rather terse and arcane, but the incantations are powerful. For example, to list files, use `ls` , which is often abbreviated further to merely `l` , while for a long-format listing, use `ls -l` . A more powerful example: to create a music playlist of all Mozart files, type: `ls -1 mozart*.mp3 > playlist.m3u` .

To start the shell:

- On Linux/Mac, open the terminal program of your choice (*konsole, gnome-terminal, rxvt, xterm, Terminal.app* etc).
- On Windows, run *Putty.exe* and connect to *user123@*www.srcf.net ← www.srcf.net is the server; *user123* is **your** username.
  PuTTY may warn you that "the server's host key is not cached in the registry": this is normal if it's the *first* time you are connecting.
  When you type in your password, the characters are *not echoed* (i.e. there will be no visible response to your typing).
  An alternative to the SRCF is to use linux.pwf.cam.ac.uk in the same way. (PWF is the Cambridge Public Workstation Facility, and DS-Filestore).

This will give you a **Bash prompt**, that looks like this: ← bash = "bourne again shell" (original author: Stephen Bourne).

```
user123@pip:~$
```
← *user123, pip , ~* are username, hostname, current directory.

The **$ prompt** means "*what is your command*"? ← if the prompt were **"#"** , it would denote *root* i.e. administrator.

Type a command (such as `ls`), then [Enter]. Your command will run, then the prompt will *return*, ready for the next command.

Use the `passwd` command if you wish to change your password. ← type old password, correctly, then new one twice. Choose well.

Readline: Interactive Line-Editing

When interactively typing commands, you will find these shortcuts useful:

Ctrl-**A** Ctrl-**E** Alt-**B** Alt-**F**	Move cursor to: start-of-line, end-of-line, one word back, one forward (respectively)
Ctrl-**W** Ctrl-**U** Ctrl-**K** Ctrl-**Y**	Cut previous word. Cut to start of line. Cut to end of line. Paste the cut-buffer.
Up_Arrow and Down_Arrow	Retrieve previous command(s) from your history (ready to alter, then re-run).
Ctrl-**R** *text*	Search backwards through history for a command partially-matching *text*.
Ctrl-**L**	Clear the screen. (i.e. scroll to a blank screen).
Ctrl-**C** Ctrl-**D**	Cancel this operation. End-of-text: closes standard-input (e.g. your shell, cat, read, ...)
TAB	Auto-complete, as far as possible, else list options. This is the most useful one.
ENTER	Actually run the command. (The cursor needn't be at the end of the line to do this).

You should **enable full tab-completion** by running this command (type carefully!) the first time you log in to a new system:
`echo "set show-all-if-ambiguous on" > ~/.inputrc` *Then* start a new shell, with "`bash`".
[The SRCF's web-hosting has an activation quirk: save time *later* by now doing: `touch ~/public_html/index.html` .]

Now, type the following command: `echo Hello World` . Then, experiment with the above, moving around and using history.
Tab-completion: typing "ech**[TAB]**" completes to "echo", while "e**[TAB]**" presents many alternative choices.

Text Editing

Text editors edit text. Fixed-width fonts are used for clarity (indentation matters), syntax-highlighting automatically colourises.

In a remote-shell, use the *Nano* editor. Run `nano the_filename`. ← nano is the successor to *pico,* named from *pine,* which is not *elm.*
Common commands are shown at the bottom of the window, with "**^X"** meaning *Ctrl*-X, and "M-X" meaning *Meta*-X (i.e. *Alt*-X). For example:
^O (write out = save), **^X** (save/exit), **^K** (cut), **^U** (uncut i.e. paste), **^W** (where-is = find), **^W** then **^R** (search and replace), **^C** (current position)

Linux GUI: use *Kwrite*, *Gedit*, or *Emacs* (but not *Vi* / *Vim*). ← Emacs is for experts; allegedly named for "Escape-Meta-Alt-Control-Shift"!

In the Linux-GUI: selecting text automatically copies it, while middle-click pastes. Use *Klipper* or *Parcellite* for clipboard-history.
Meta is a synonym for *Alt.* *AltGr* (alternate graphic) gives symbols such as **µ** (AltGr + m), or **É** (AltGr + ; , then Shift-E).
[For MS Windows, the inbuilt NotePad editor is too basic; the free *Notepad++* editor (download from notepad-plus-plus.org) is a good choice.]

Optionally, customise your session by editing your .bashrc file: `nano ~/.bashrc` . I recommend *appending* these lines:
[Type carefully, then save the file and exit nano with **^X**, **Y**, **ENTER** (i.e. save, yes, confirm filename). Then run `bash` again; if there are errors, fix them.]

```
alias l="ls"              #List files
alias ll="ls -l -k"       #List detailed.
alias la="ls -a"          #List all files, inc hidden files beginning with a dot.
alias lsd="ls -d */"      #List only directories.
alias s="cd .."           #Up a directory
alias p="cd -"            #Previous directory
alias cp="cp -i"          #Copy, but ask before overwriting.
alias mv="mv -i"          #Move, but ask before overwriting.
alias rm="rm -i"          #Delete, but with confirmation.
export PATH=$PATH:~/bin   #Add your own script directory to the search PATH
```

Files, Directories, and the File System – a Reference

On Unix, *everything* is a file:

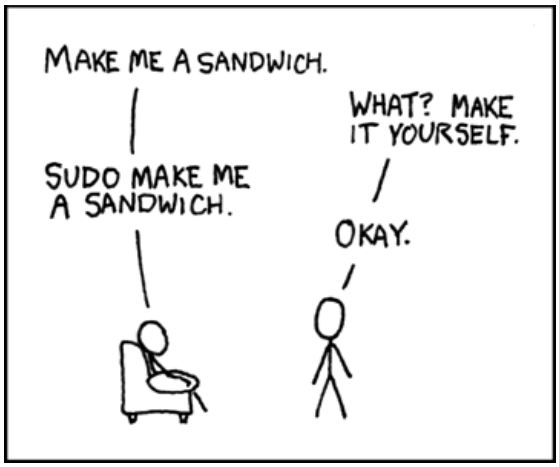
- Even a directory (aka "folder") is just a special type of file, named "`.`" or the parent directory is named "`..`" .
- Special files include `/dev/sda` (your disk), `/dev/random` (a source of randomness) or `/dev/null` (the "bit-bucket").
- File names *can* contain any character (except "`/`", which is the directory separator). But please don't: you should really only use the letters, numbers, and underscore, dot and dash. Lower-case is conventional.
- Avoid spaces in file names, because you will get confused (e.g. is "apple pie" one file, or two?) and you have to be fussy with quoting names or *escaping* the spaces like this "`apple\ pie`". Bash considers space to be a *delimiter*.)
- The directory-separator is forward-slash "`/`" ← URLs are copied from Unix; DOS got it wrong, so Windows still uses backslash.
- Hidden files (whose names start with a dot) are not displayed unless you use `ls -a` to show *all*. e.g. `~/.inputrc` .
- File paths are either *absolute* with respect to the root directory (starting with a "`/`", e.g. `/home/abc123/myfile.txt`), or *relative* to the current directory (no leading slash, e.g. `../otherfile.txt`). "`..`" means "parent" /"up one level".
- Your own files live in `/home/your_userid` , sometimes shortened to just "`~`" (the tilde-character).
- Your own shell-scripts and binaries go in `~/bin/` ← which is in your $PATH (the set of directories bash searches for commands).
- Unix has always been multi-user, so files have *ownerships* and *permissions* to control whether the {*owner*, *group*, *world*} can {*read*, *write*, *execute*} them. E.g., this file, which is owner read/write, group readable, world readable is listed as: `-rw-r--r-- 1 rjn rjn 277432 Feb 26 18:28 computing_supervision.odt` .
- Use the chown and chmod commands to change owner/group and mode (permissions).
- ls shows different colours for different types of files: directory, symlink, text, executable, device, special, broken.

All files exist in a *single* **tree** structure, under one *root* directory, "`/`". The main *branches* of the filesystem are:

- `/` - the root directory. This is the root of the file-system tree. It contains the following (and a few others).
- `/bin` - essential binary files, e.g. `ls`. N.B. bin is ***not*** your trash!
- `/boot` - the kernel (and initrd) live here, to boot ("bootstrap") from. (historically, /boot was a separate tape drive).
- `/dev` - special device files, e.g. disks, memory, serial-ports, usb-interfaces.
- `/etc` - configuration files (all in text format: "editable text configuration", this may be a backronym)
- `/home` - users' data. Your data is in `/home/user123/` (obviously, replace user123 by your *own* userid).
- `/lib` - library files for `/bin` and `/sbin` and kernel modules. (but most libraries live in `/usr/lib/`).
- `/mnt` - mount-point for removable devices (CD,USB-key etc). Linux doesn't use "drive letters". Also "`/media`".
- `/proc` - information on processes. E.g. look at `/proc/self/` or `/proc/cpuinfo`
- `/root` - the home directory for the system administrator, also named "root". (***Not*** the root directory, "`/`").
- `/run` - temporary lock-files used for process-coordination at run time (often as tmpfs).
- `/sbin` - system-administration binaries. (Some distributions now unify `/bin`, `/usr/bin`, and `/sbin` .)
- `/sys` - special files for interacting with the system: kernel and hardware. (e.g. `/sys/class/gpio`).
- `/tmp` - temporary files (automatically deleted at reboot).
- `/usr` - *most* programs and resources live here ("Unix system resources"). Data files live in `/usr/share/` .
- `/var` - variable data: logfiles, databases, website content etc. E.g. `/var/log/syslog` or `/var/www/html/` .

Users and Root

Normal users can only write to their *own* files. By convention, most files are world-readable, unless explicitly set otherwise. You can safely explore the rest of the system, you can't break it. Daemons (e.g. Apache webserver) are also "users". The system administrator, *root*, is all-powerful. To switch user, use the `su` command, or you may `sudo` ("super-user do").



XKCD: 149

Basic Commands and File Manipulation

Most Unix commands follow the pattern: **command -option(s) argument1 argument 2 …**
Example: `ls -la kitten.jpg cat.jpg` ← The options "-la" can be written separately as: "-l -a"
The *manual* page for a command lists its options. For `ls`, use `man ls`. (Q to quit). Most commands also take `-h` for help.
Try typing the following, pressing **[Enter]** after each line. Feel free to experiment, and use tab-completion to speed up typing:

```
#Testing                    ← Anything following "#" is a comment (for the benefit of the human operator), and has no effect.
echo Hello                  ← The echo command prints its arguments. This prints "hello" and returns the prompt.
echo Hello     World        ← Note that there are multiple spaces between "hello" and "world". But echo just sees 2 arguments to print.
echo "Hello    World"       ← Now, we have quoted the string, so it is a single argument. Echo prints the spaces as we expect.
touch foo bar baz           ← The touch command updates the timestamp on a file, creating an empty file if it's not already there.
ls                          ← You should now see files called "foo", "bar" and "baz"  [known as "canonical metasyntactic variables"].
mv foo wibble               ← This moves (renames) "foo" to "wibble". Unix isn't verbose: there is no (extraneous) confirmation message.
ls                          ← You can now see that this worked.
cp wibble wombat            ← Copy the file "wibble" to "wombat".
ls -l                       ← List the files, now in a long format. The first column, such as -rwx-rw-r-- , shows the file permissions.
rm wombat                   ← Delete (remove) the file "wombat". Note that "rm" without an "rm -i" alias deletes without prompting!
echo hello > foo            ← Writes the string "hello", redirect by the ">" operator to "foo". The file "foo" now contains the word "hello".
cat foo                     ← The cat command concatenates the file "foo" to the screen. i.e. print the contents.
file foo                    ← The file command prints information on what a particular file actually is. This one contains ASCII text.
ln -s foo qux               ← Create a "symbolic link" to "foo" by another file named "qux". Symlinks act as signposts, and can be chained.
ls -l foo qux               ← List the files. ls -l  shows that "qux → foo".  It also uses a different colour for qux, denoting a symlink.
readlink -f foo qux         ← Readlink canonicalises the full, absolute path, resolving (i.e. following) symlinks when needed.
pwd                         ← Print working directory. Shows which directory you are currently in. In this case,  "/home/userid"
cd /                        ← Change directory to the root directory, "/".  Try listing it ("ls") to see the file-tree in the previous page.
pwd                         ← Shows where we are again.
cd                          ← Default destination (when cd has no argument) is your home-dir. ["cd -" means "back"; "cd .." means "up"]
pwd                         ← We are back in  "/home/userid"
mkdir test                  ← Make a new empty directory, called test.  (Can create multiple directories with  "mkdir -p one/two/three/").
cd te[TAB_KEY]              ← Change into it.  Use tab-completion to save typing, i.e. type "te" and TAB which autocompletes to "test".
pwd                         ← We are now in  "/home/userid/test"
touch a b c                 ← Create empty files "a", "b" and "c" within our new directory.
cd ..                       ← Go up one directory.  [NB: in bash, if you'd followed symlinks, "cd .." goes "back-up", not "straight-up".]
ls test                     ← List the contents of test/   You should see "a b c" listed.
rmdir test                  ← Delete (remove) the directory test. This will fail because rmdir only works on non-empty directories.
rm -rf test                 ← Remove, recursively, forcefully! Deletes without confirmation (there is no recycle bin; it is now gone).
rm absent.txt               ← The file doesn't exist, so this will fail, printing an error message. Unix says nothing unless it needs to.
rm foo bar baz wibble qux   ← Clean up: delete the temporary files we just created.
```

Try these. Use `man the_command_name` to see what they do, and what options they can use (in man, "/" to search, Q to quit).

`whoami`	(prints your username)	`who`	(who else is logged in at the moment)	`uptime`	(time since boot)
`beep`	(beeps. Try "-f 1000" option)	`date`	(current date, various formats)	`cal`	(prints a calendar)
`du`	(disk-usage in this directory)	`df`	(disk-free-space. Try "-h" option)	`fortune`	(print a fortune cookie)
`sleep 3`	(pause for N seconds)	`uname -a`	(system information)	`lsb_release -d`	(O.S. version)

Redirection of stdin/stdout with: "|" ">" ">>" and "<"

Each command has 3 standard streams, **input** (*stdin*, 0, default:keyboard*);* **output** (*stdout*, 1, default:screen*);* **error** (*stderr*, 2, default:screen).They can be chained together by pipes or redirected to/from files. A **pipe** "**|**" connects stdout → stdin.
Redirection: "**>**" overwrites, "**>>**" appends; "**<**" reads from file. [`2> /dev/null` discards stderr, sending it to the null device.]

```
fortune  | rev                ← Reverse each line (character-wise). [The pipe symbol is on the backslash key with shift.]
fortune  | tac                ← Print the lines, in reverse order ("tac" is "cat" backwards).
dmesg    | less               ← Display kernel messages, but use the "less" pager to scroll. (Use arrows to scroll; Q to exit).
fortune  | tac | tr abc xyz   ← Multiple pipes. tr transliterates characters in the first set to the second  (a → x,  b → y, c → z).
echo hello    >  test.txt      ← Redirect the output (stdout) of echo to the file "test.txt". Check the contents with "cat test.txt".
echo hi       >  test.txt      ← test.txt now contains "hi". The single > over-writes anything that was there before.
echo bonjour  >> test.txt      ← test.txt now contains "hi newline bonjour".  The double >> appends to the file.
cat           <  test.txt      ← redirect the input (stdin) of "cat" to take input from the file "test.txt".
```

Many commands can read data *either* from a specified file, *or* from a pipe. These four all do exactly the same thing:

```
cat text.txt ← open a file      cat < test.txt  ← redirection from file      cat test.txt | cat  ← pipe stdout to stdin.
cat test.txt > cat.txt; cat cat.txt | cat | tac | cat | cat | tac | cat    ← silly, useless use of cat!
```

Globbing: matching wildcards

A glob is the shell's expansion of special wildcard characters within filenames. E.g. "*.jpg".

*****	matches any number of characters (including zero). (If there is no match, * becomes literal.)
******	like *, but includes subdirectories
a[bcd]e	matches any *one of the characters within the brackets*, i.e. abe, ace, ade
a?c	matches any *single character*. e.g. axc or ayc, but not axxc. (If there is no match, **?** becomes literal.)
a{bb,cc}d	expands all the *alternatives* within the {}, i.e. abbd and accd.

```
mkdir globtest ; cd globtest                ← Create a temporary directory and change into it
touch apple banana bAnana pineapple         ← Create some empty files with interesting names

echo *                                      ← "*" by itself matches every file
echo *ple                                   ← "*" matches zero or more characters, followed by "ple". i.e. apple and pineapple
echo b?nana                                 ← The "?" matches any single character. In this case, banana and bAnana.
echo a[pqr]ple                              ← The [] give a choice of p,q,r. This would match apple/aqple/arple, but only apple exists.

cd; rm -rf globtest                         ← Clean up afterwards, get rid of the temporary files. (-rf means recursive, forced, delete),
```

Variables and the "$" operator

Variables store data (integers, floating point, strings, arrays). A variable is created when data is assigned to it.
Bash's use of $ is slightly quirky. Try: ← **$** is not a *sigil* (as in PHP/Perl), but a ***unary operator*** whose main purpose is "get the value of".

```
x=42                                ← create the variable x and assigns the value of 42 to it. N.B. no spaces around the equals sign.
echo $x                             ← prints the value 42, which is the contents of x.
name="Albert Einstein"              ← create a variable name and assign the value Albert Einstein. Quotes are needed because of the embedded space.
echo "Hello $name"                  ← prints "Hello Albert Einstein". Note that $ is interpolated within double-quotes. Try now with your own name.
echo 'Hello $name'                  ← prints "Hello $name" (literally). Single quotes are literal, and the $ is not special.

x = 42                              ← This fails: with "x: command not found". Spaces are not allowed around the equals.
name=Albert Einstein                ← This fails with "Einstein: command not found". Bash treats spaces as a separator, (unless you quote them).

y=$x                                ← y is now also 42.
z=$((2*x +1))                       ← z is now 85. $(( )) does arithmetic evaluation. See man bash and look under "Arithmetic Evaluation".
a=$((x/5))                          ← a is now 8. The / operator does integer division. Watch out for this one. [For remainder, use the % operator.]
echo $x $y $z $a                    ← 42 42 85 8

echo ${name,,}                      ← ${} has many special tricks. E.g. ",," means lower-case the string. Try a single comma.
echo ${name^^}                      ← ^^ means upper-case. For a full list, see man bash and look under "Parameter Expansion".
echo ${#name}                       ← prints 15, i.e. the number of characters in the variable name.
echo ${name:2:5}                    ← prints "bert". Substring, from offset of 2, for length 5. Note that offset is zero-based.
echo ${name//[aeiou]/Z}             ← searches for pattern (in this case, [aeiou]) and replaces it (in this case, by Z).

list=(General Relativity 1915)      ← list is now a 3-element array. See: http://tldp.org/LDP/abs/html/arrays.html
echo ${list[1]}                     ← access an array element by its index (starting from 0). To count the elements, use: ${#list[@]}

if [ $x == 42 ] ; then echo yes; fi ← test value of a variable. Spaces are critical. "==" is used for testing (vs. "=" for assignment).
```

Bash also has some special variables that are predefined for you. Notably:

- **$0**, **$1**, **$2**, … the arguments to the script. (**$0** is the name of the script itself). `shift` moves $n ← $n+1
- **$#** tells you how many arguments there are (not counting **$0**).
- **$USER** and **$HOME** important "environment variables", containing the username and home-directory.
- **$PWD** the current directory. Try: `echo $PWD` . See also `$PATH` .
- **$RANDOM** a random integer between 0 ... 32767. It changes each time you read it. (32767 is 2^{15} -1).

Quoting: Single and Double-Quotes

Single quotes are literal. *Anything* inside single-quotes is treated exactly how it appears. (This also means that you can't include a single-quote within single-quotes.) Try: `echo 'It'"'"'s not easy'` ← i.e. 'It' then "'" then 's not easy'

Double quotes get interpreted. Variables are expanded, and *backticks* "`" are evaluated for command-substitution. To escape (make literal) a special character, prefix it with backslash "\" i.e. `\$` (← for $) `\`` (← for `) `\"` (← for ") `\\` (← for \).
E.g. `echo "On `date +%A`, $name said \"I owe you \$$x\"."` ← On Thursday, Albert Einstein said "I owe you $42".
[Note that the date command is surrounded by backtick characters, meaning "the-result-of" (not single-quotes). The backtick key is to the left of the "1" key.]

Shell Scripts: Automating Repetitive Tasks

Rather than re-typing many commands, you can save a sequence of them into a *script*, and use the script instead.
Shell scripts range from 3-line utilities to 5000-line monsters. Many system commands are actually scripts, e.g. `/bin/zcat`.

- Shell scripts must have this *magic* first line: `#!/bin/bash` ← #! is a "shebang". It tells the *kernel* to run /bin/bash..
- Comments begin with `#` and describe what the script does.
- Commands go on separate lines (or separated by ";"). ← Bash doesn't *need* a semicolon at the end of every line.
- Parameters are `$0`, `$1`, `$2`.... (there are `$#` of them). ← $0 is the scriptname. $# doesn't count $0. Can *shift* .
- End with an exit code, typically `exit 0` . `0` for success, `1` for error. ← not strictly required, but good practice.
- Make the script executable with `chmod +x scriptname.sh` ← "change mode, set the executable bit"
- Run it with `./scriptname.sh` (or put it in your `$PATH`).

Create/edit the file hello.sh (`nano hello.sh`) with the following contents: ← nano will helpfully syntax-highlight while you write.

```
#!/bin/bash
#This shell script is an example
echo "Hello World"
date
```

Save the file (in nano, use **^X** to save and exit); then make the file executable: `chmod +x hello.sh`
Now try it out: `./hello.sh` . It should print "Hello World" and the date. ← specify the *path* explicitly, with the "./" prefix.

Logic and Conditionals

Each *command* has a return value (or exit status). If it *succeeds*, the status is `0`; while if it *fails* the error number can convey the type of failure (though it is typically 1). The return value (retval) of the *last* command is stored in `$?` . `!` inverts the result.

```
true                          ← the command "true" does nothing, and returns success.
echo $?                       ← return value of the previous command is 0
false                         ← the command "false" does nothing, and fails.
echo $?                       ← 1
! false ; echo $?             ← 0. "! cmd " carries out command, then inverts the return status.
! echo hi ; echo $?           ← hi, 1   (why?)
```

Now, we can test variables and commands, and respond accordingly. There are 3 main ways to write an if clause:

```
if CONDITION ; then CMD  ; fi                                  ← "fi" is "if" backwards.
if CONDITION ; then CMD1 ; else CMD2 ; fi                      ← if … then … else … fi
if CONDITION ; then CMD1 ; elif CONDITION2 ; then CMD2 ; else CMD3 ; fi   ← if … else if … then … else … fi
```

Try it out, by experimenting with variants of: `if true; then echo YES; else echo NO; fi`

For testing, use the **`test`** , or **`[`** comand (described in `help [`). The test builtin can variously compare strings and integers, check for empty or non empty strings, and whether something is a file, directory, executable, newer, older etc. Watch *spaces*.

When saved as a shell script (rather than typed at the prompt), we *indent with tabs* for clarity. Nested "if"s indent again. E.g.

```
#!/bin/bash
#A shell script to ask the final question pertaining to human experience.
echo -n "What is the ultimate answer?: "
read input
if  [ "$input" == 42 ] ; then                          #be careful, typing the spaces for "[".
       echo "Yes, $input is the Answer to Life, the Universe and Everything."
       exit 0
else
       echo -n "What do you get when you multiply six by nine?: "
       read input
       if  [ "$input" == 42 ] ; then
              echo "Yes, $input is the Answer to Life, the Universe and Everything."
              exit 0
       fi
       echo "You should re-read Douglas Adams."
       exit 1
fi
```

Optional Exercise: write a script that prompts for a shape, and dimensions, and calculates its moment-of-inertia. MOI formulae for various shapes are:
point-mass: $I = m.x^2$; hoop: $I = m.r^2$; disc: $I = 1/2\ m.r^2$; sphere: $I = 2/5\ m.r^2$; rod (about middle): $I = 1/12\ m.l^2$; rod (about end): $I = 1/3\ m.l^2$.
For calculations in bash use: $(( )), e.g. i=$((m*r**2)) . This is described in the "Arithmetic Evaluation" section of the bash manpage.

More Shell Syntax

A brief summary, by example. Try `help the_keyword`, or `man bash` or see: www.gnu.org/software/bash/manual/bashref.html

```
echo hi && echo there                                  ← prints hi, AND prints there. (&& is a short-circuit operator).
echo hi || echo there                                  ← prints hi OR prints there. (|| is a short-circuit operator).

for fruit in apple orange kiwi; do echo "tasty $fruit"; done   ← for-loop: for item_name in list, do ...
for ((i=0; i<10; i++)); do echo "i is $i"; done        ← for loop. Initialise i to 0 ; test i < 10 ; iterate by adding 1 each time.
while : ; do echo nag; done                            ← infinite while loop (":" is a no-op). Stop it with a break, or Ctrl-C.
while : ; do echo once; break ; done                   ← break out of a for, while, select, or case statement before its end.
exit N                                                 ← exit the script (or the shell) with an exit code. N is the next $? .

x=`command`                                            ← get the output of a command and assign to variable x.
x=$(command)                                           ← another way to write this, but several $() can be nested.

[ -f my_filename ]    ; echo $?                        ← test if my_filename exists, and is an ordinary file.
[ -d a_directory ]    ; echo $?                        ← test if a_directory exists, and is a directory. See: "help [ ".
[ -z "$variable" ]    ; echo $?                        ← test if variable is an empty string. N.B. the quotes are required.
[ -n "$variable" ]    ; echo $?                        ← test if variable is not empty.  N.B. the quotes are required.

read -ep "pick a number: " num    ; echo $num          ← prompt the user to enter a value, to be read into variable num.
echo -e "\a\tHello\033[031mRed\033[0m\n\n"             ← echo -e supports escape codes. \a,\t,\n are bell,tab,newline. ANSI.

select decay in alpha beta gamma; do echo $decay; done ← a multiple-choice menu. Use break (or Ctrl-C) to exit the menu.
case "b" in a) echo AA;; b) echo BB;; *) echo def;; esac ← case is another way to write multiple-choice "if"s.

function myfn() { echo hello ;}     ; myfn             ← define a function. Parameters are passed in as $1, $2 etc.
```

Special Characters – a Reference

Virtually every character has a special use or seven! Here's a brief *summary*, for future reference. See also `man bash`.

Sym	Name	Description	Example
!	(bang)	begin script with a shebang. history.	#!/bin/bash
$	(dollar)	get the value of a variable. modify it.	$HOME ${#HOME}
$()	(dollar)	get the result of a command.	X=$(date)
`	(backtick)	get (or interpolate) the result of a command.	X=`date`
%	(percent)	formatting a string. modulus operator.	date +%Y-%m-%d
^	(caret)	bitwise xor operator, or substitution.	${HOME^^} $((3^2))
&	(ampersand)	background a command, bitwise, short-circuit and.	xclock &
*****	(star)	globbing (pattern matching). multiply.	*.jpg
( )	(parentheses)	arrays, maths	$((6*7))
[ ]	(brackets)	arrays, globbing, tests	${PIPESTATUS[0]}
{ }	(braces)	variables, group commands, or globbing	{ cmd1; cmd2; } | cmd3
< >	(angle brackets)	redirection of input or output.	cmd >> file
_	(underscore)	*Not* special, used as alphanumeric.	file_name_with_underscore
-	(dash or minus)	command options. maths.	ls -l
=	(equals)	= for assignment (== for comparison)	a=b sets a equal to b.
+	(plus)	maths.	let i++
;	(semicolon)	command-separator.	date ; fortune ; du
:	(colon)	no-operation, or "true"	while : ; do ...
'	(single-quote)	literal quoting.	echo 'Literal $ sign'
"	(double-quote)	quoting with evaluation.	cost=3; echo "Price is £$cost"
#	(hash)	comment sign. part of a shebang.	#ignored
~	(tilde)	home directory, or regex match operator.	cd ~
.	(dot)	include ("source") a file into this context.	. library_file
/	(forward-slash)	directory separator, integer-division.	$((7/3)) is 2
****	(backslash)	escaping another character (special ↔ literal).	\n \$ filename\ with\ spaces
|	(pipe)	chain commands, bitwise, short-circuit "or".	cmd1 | cmd2 true || false
?	(question mark)	globbing to match one character.	abc?d
	(space)	separator between arguments (any number).	cmd arg1 arg2 arg3
\n	(newline)	used to separate records in files.	echo "Hello"$'\n'"World"
\t	(tab)	used to separate fields within records.	[tab delimited data]
\033[31m	(ANSI)	colour codes, to change text colour.	echo -e "\033[31mRED\033[0m"

Regular Expressions (regexps and grep)

A regular expression is an elaborate pattern used to match parts of a text we are interested in (similar to how "globs" can match filenames). See: tldp.org/LDP/abs/html/regexp.html . REs are extremely *powerful* (xkcd.com/208/) , and sometimes *confusing* ("Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems!" - Zawinski).

The tool is `grep` ("global search regular expression and print"). It scans the input, and prints matching lines (or parts of lines). Useful flags to grep are: **-i** (case-insensitive) **-r** (recursive) **-n** (number lines) **-v** (invert match) **-E** (extended RE) **-o** (only print the matching part of line) **-C** (lines of context). Some demonstrations of what regexps can do:

1. Find full name of a user: `grep rn214 /etc/passwd` ← i.e. find lines matching "rn214" in "/etc/passwd", the user-list for the system.

2. Consider a crossword puzzle, where you know that the clue is 16 characters, in the form "E_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ M". Linux's spellchecker wordlist is in `/usr/share/dict/words`. So, do: `grep -iE '^e.{14}m$' /usr/share/dict/words`
Grep flags: `-i` (case insensitive) `-E` (extended) **Regex:** `^` (start of line) `e` (literal) `.` (any char) `{14}` (repeat 14x) `m` (literal) `$` (end of line).

3. Consider another file containing data in the form: ← a copy of this is conveniently already on the SRCF, at ~rn214/composers.dat

```
Johann Sebastian Bach           1685-1750       johann@leipzig.de          Baroque
Ludwig van Beethoven            1770-1827       ludwig@bonn.de             Classical
Wolfgang Amadeus Mozart         1756-1791       mozart@vienna.org          Classical
Jean Sibelius                   1865-1957       karelia@finland.fi         Romantic
Gustav Mahler                   1860-1911       gustav@metropolitan.com    Modern
```

and you want to get a list of email addresses, ready for blind-carbon copy into an email. The solution is:
`cat composers.dat | grep -oE '[a-z0-9_.-]+@[a-z0-9_.-]+' | tr '\n' , | head -c -1`
Explanation: `-o` (only print matching part) `[...]` (set of characters) `+` (one or more) `tr` (newline to comma) `head` (remove last comma)

4. Chained greps: `grep -inrE '\.php|\.html' * | grep -v https`
Explanation: given a website project directory, search recursively for all links to php/html pages, filter out https (secure) links, to find insecure links.

Bash also has regular-expression matching: see the `[[ value =~ regex ]]` syntax and `${BASH_REMATCH[n]}` E.g.:

```
mass="16.3 kg"
[[ "$mass" =~ ([0-9.]+)\ (g|mg|kg) ]] && echo "valid"   ← Does "$mass" match the expected pattern?
echo ${BASH_REMATCH[0]}                                 ← contains the entire matched string, "16.3 kg"
echo ${BASH_REMATCH[1]}                                 ← contains the 1st parenthesised sub-expression, the number, "16.3"
echo ${BASH_REMATCH[2]}                                 ← contains the 2nd p.s.e., the unit, "kg" ("g" and "mg" also allowed).
```

RegEx Syntax – a brief, *incomplete*, Reference

Regular Expression Syntax isn't that difficult, once you break it down logically. Most programming languages (*Perl*, *Python*, *Javascript*, *PHP*, *grep -E, sed -E* [note the -E for "extended regex format", which is now considered standard] use the same rules, known as PCRE (Perl-compatible Regular Expressions), see: wikipedia.org/wiki/Regular_expression and `man pcre` .

Symbol	Example	Explanation
`abc123_`	`cat`	*Literal* characters. The lower/upper-case characters, digits, space and underscore are literal.
`.`	`c.t`	Dot means *any single character*. Here, "c.t" would match e.g. "cat", "cot", or "c3t".
`?`	`ab?c`	Quantifier: 0 or 1 of the previous atom. This matches "abc" or "ac".
`+`	`ab*c`	Quantifier: 1 or more of the previous atom. This matches "abc", "abbc", "abbbc" etc
`*`	`ab+c`	Quantifier: 0 or more of the previous atom. This matches "ac", "abc", "abbc", "abbbc" etc
`{n,m}`	`ab{3,5}c`	Min/max quantifier: n to m. This matches "abbbc", "abbbbc", "abbbbbc".
`\`	`\.`	Backslash *escapes* the next symbol, to make it literal. "\." means "an actual dot character".
`\\`	`a\\3`	Of course, that means that to explicitly have a single backslash, you must type it twice.
`\d`	`1\d2`	\d means "any digit". So "1\d2" could match "102", "112", … "192".
`\n`	`hi\nthere`	\n, \t are newlines and tabs. Backslash makes literals special, and specials literal!
`[...]`	`[a-f]`	Square brackets enclose a character-class (range). This matches any character from a – f.
`[...]`	`[a-z0-9_-]`	Combined ranges: any letter, number, or the underscore, or a dash (if it's last, it's literal).
`[^...]`	`[^>]`	Inverted character class. Anything that isn't this. E.g. "</?[^/>]+>" matches HTML tags.
`()`	`a(bc){3}d`	Group atoms together in a subexpression. This matches "abcbcbcd".
`|`	`cat|dog`	Alternative branches. This matches "cat" or "dog". Compare: "ca(t|d)og".
`^`	`^ten`	Anchor: assert that this is the start of text. So this would match "tennis ..." but not "kitten"
`$`	`cat$`	Anchor: assert that this is the end of text. So this would match "… wildcat" but not "cats"
`/i`	`cat/i`	Pattern modifiers. Notably, "/i" makes the whole thing case-insensitive.

Using this reference should now allow you to "decrypt" the examples above. This covers the most common examples of RE, though there are many more sophisticated uses, such as "backreferences", "look {ahead/behind} {assertions,negative-assertions}", and more sophisticated assertions (e.g. "word-boundaries", "\b") and character classes e.g. [:print:]. The stream-editor, *sed* is really useful for regex search-and-replace, within pipes or files.

Some Selected Commands

There are about 50,000 commands on a fully-loaded Ubuntu system, and about 500 that are frequently useful. Here are some:

Command:	Example:	Description:
head	`| head -n 5`	Read only the first n lines of a file, or stdin.
tail	`tail -f filename`	Read only the last 10 lines, and *follow* the file as it grows.
sort	`| sort`	Sort lines in alphabetical order.
uniq	`| uniq`	Unique: remove duplicate lines *if adjacent.*
wc	`wc -l`	Word count (-l for lines, -w for words, -c for characters)
cut	`cut -d : -f 2-4`	Cut into columns, delimited by : , and output only columns 2-4.
paste	`paste file1 file2`	Paste files together horizontally.
diff	`diff file1 file2`	Show the lines where files differ. (see also patch, and kdiff3)
sed	`sed -e 's/search/replace/g'`	Stream-editor, many examples: sed.sourceforge.net/sed1line.txt
top, **htop**	`top`	Table of processes: view what is currently running. Q quits.
ps	`ps -aux`	List all processes (ps has powerful set of filters, pstree shows hierarchy).
kill, **killall**	`killall -9 yes`	Kill a process by PID or by name. kill -l lists available signals.
ps*	`ps2pdf in.ps out.pdf`	Various postscript manipulation tools. Use lyx or latex to generate.
gv	`gv file.ps`	Ghostview: display postscript and PDF files. (Also, try evince, okular .)
pdf*	`pdflatex in.tex`	Various PDF manipulation tools (e.g. extract, merge, typeset).
lpr	`lpr *.pdf`	Print files. -P sets destination device. lpq for queue. cancel -a stops them.
find	`find . -name '*.JPG'`	Search (recursively) all files with certain properties (name, size, date, etc)
locate	`locate -i part_of_filename`	Find all filenames containing this string. (the database is updated nightly)
qmv	`qmv *.html`	Use editor to quickly batch rename files
xclock	`xclock -geometry 100x150+7+9`	Show a GUI clock, with specified window size and position.
ascii	`ascii`	Print the character names and meanings and hex codes.
xxd	`| xxd`	Hexadecimal dump of input data.
strings	`| strings`	Print every group of at least 4 consecutive human readable characters.
dd	`dd if=/dev/zero bs=1 count=5`	"dd" stands for copy and convert. ("cc" was taken by the C compiler).
bc	`echo "scale=2; 4/3" | bc`	Binary calculator. This example gives 1.33.
which	`which ls`	Finds the full path of the command that is run. See also `apropos`, `whatis`.
time	`time some_command`	Prints the time taken (CPU time, and wall-clock time) for a command.
tar, **bzip2**, **gzip**	`tar xvzf file.tar.gz`	tar is the "tape archiver", and gzip/bzip2 are compression utilities.
mount, **umount**	`mount /dev/sdX1 /mnt`	Mount a device into the filesystem tree. See also eject.
dpkg	`dpkg -l`	Debian package manager. (There is a GUI equivalent, synaptic).
apt-get	`apt-get install firefox`	Advanced package tool. Download, install firefox. Really easy.
cowsay, **figlet**	`cowsay $(fortune)`	Fancy text formatting. Try piping: `fortune | figlet | lolcat -a`
units	`units 10kg mg`	Versatile unit convertor. Try '100 degcelsius' and 'tempcelsius(100)'.
qrencode	`qrencode -o qr.png "Data"`	Encode "Data" in a QR (quick response) code.
xli, **qiv**	`qiv qr.png`	Quick image-viewer. Use "?" to list shortcut keys. Q quits.
festival	`echo hello | festival –tts`	Festival is a speech synthesis program.
play	`play sound.wav`	Play any audio file (with various effects).
mplayer	`mplayer video.avi`	Play almost any audio/video file, with many, many options.
sox	`sox file.wav file.mp3`	Sound exchange: convert and filter all sound formats.
convert	`convert photo.jpg photo.png`	Convert and process images. ImageMagick is *amazingly* powerful.
ffmpeg	`ffmpeg video.avi video.mpg`	Convert and process video. ffmpeg is another swiss-army-knife.
gphoto2	`gphoto2 -P`	Control a digital camera: most cameras can be triggered over USB.
zenity	`zenity --question --text "Happy?"`	GUI dialog boxes for script interaction and messages.
perl	`perl -pie 's/change this/to that/g' file1 file2`	Perl: another scripting language.
sqlite3	`sqlite3 database.db`	Use the SQLite database program to open the file database.db.
psql	`psql database_name`	Connect to a PostgreSQL database.
git	`git clone repository_name`	Clone a source-code repository. pull, commit, push. See gitref.org .
gcc, **make**	`gcc -Wall -o hello hello.c`	Compile a program, using the GNU C compiler

Bash One-Liners - Some Examples and Inspirations

A one-liner is a short, temporary script. Try these (some will only work on your local machine), create your own, or see: www.bashoneliners.com . If something is useful to you, save the file in your `~/bin` directory, and you can use it again.

#Alarm clock: snooze 10 minutes, then speak "wake up" repeatedly (also try the "`beep`" command):
```
sleep 600 ; while : ; do echo "wake up" | festival --tts ; sleep 2; done
```

#Download all system updates, and install them (This is very useful for system-administrators as a shell alias):
```
sudo apt-get update && sudo apt-get dist-upgrade
```

#Synchronise local work with a directory in the office. Be careful about trailing slashes, or using the `--delete` option:
```
rsync -avz -e ssh /home/user123/myproject/  laboratory_pc:myproject/
```

#Trashcan function. Use "`cn`" instead of "`rm`" as a safety measure. Put this in your `~/.bashrc` .
```
function cn(){ /bin/mv -f --backup=numbered -- "$@" $HOME/.local/share/Trash/files ; }
```

#Clipboard sync. Get the clipboard from another machine copied to this one. (See also `x2x`).
```
function ccc(){ ssh other_machine "DISPLAY=:0 xclip -o"  | xclip -i ; }
```

#SSH forwarding: outbound mail and web-proxying via a machine you trust. (Also enable SOCKS v5 in Firefox).
```
ssh -L 8025:localhost:25 -D 1080 www.your.proxy.org
```

#Make a temporary music or video playlist.
```
mplayer file1.mp3;  mplayer file2.wav;  mplayer file3.ogg
```

#Synthesise sounds using sox (or play). This produces 2 sine waves superposed, at 440 and 660 Hz. Try 440 + 445 Hz.
```
play -n -c1 synth sin 440 sin 660 fade h 0.1 2 0.1
```
#Now experiment with chords in Pythagorean tuning vs. Equal-temperament: can you hear the differences in intonation?

Tuning	Perfect 5th	Major 3rd	Minor 3rd
Equal temperament:	440, 659.26	440, 554.37	440, 523.35
Pythagorean:	440, 660	440, 550	440, 528

#Create a set of QR codes for conference attendees. This one is typed on several lines for ease of reading, but you can enter it
#in one line (in which case, the semi-colons are all essential), or escape the line-breaks by `\` . This one is really a better case
#for writing as a proper shell-script file, so you can add comments to explain it. This approach is much faster than making 100
#QR codes separately, especially, if you decide later that you need to re-do some of them!
```
mkdir badges; cd badges;
for name in "Albert Einsten" "Richard Feynman" "Niels Bohr" "Erwin Schrödinger" "James Maxwell"; do
   filename=$(echo -n ${name,,} | tr -sc '[:alnum:]' _);
   echo -e "$name\nMy Conference\n$(date "+%a %d %b")" | qrencode -o ${filename}_qr.png;
done;
cd .. ;
eog badges
```

#Convert an HTML document to a list of tags and a plain text. Get the list of tags with *grep*, while the plaintext filter removes
#them using *sed*. Both cases use the regular expressions above, in "extended" mode. The grep is easy to follow, searching for
#an opening < then an optional / then one or more characters that are neither > nor /, then the closing > . The sed is similar,
#doing a search and replace (the replacement is the empty string between the final //), with escaping of the forward-slashes.
```
htmlfile="myfile.html" ;
cat $htmlfile | grep -oE '</?[^/>]+>'         >  tags.txt ;
cat $htmlfile | sed   -E 's/<\/?[^\/>]+>//g'  >  text.txt ;
```

#Get the latest news from the BBC, format it on one page and print it. This uses the API at https://newsapi.org/bbc-news-api
#and an API key, which you can register for free. We then download the JSON (Javascript Object Notation) data format with
#CURL, and format it with jq. Then use a2ps (Any to Postscript) to print it, though you could just use "lpr". Try scheduling
#this with "at". You can the be woken up by the noise of the printer… and have your news digest ready to read!
```
api_key="29e4507430884e589e5f6ceabf3e3bee";
url="https://newsapi.org/v1/articles?source=bbc-news&sortBy=top&ApiKey=$api_key";
curl -s "$url" | jq -r .articles  | a2ps --stdin="BBC NEWS"
```

Networking

Each computer (host) has at least one **DNS** (domain name system) entry, such as "www.magd.cam.ac.uk", corresponding to one of more **IP** (internet protocol) addresses such as "`128.232.235.115`". Each network protocol (such as HTTP, HTTPS, SSH) connects to a specific port number on that IP (standardised as respectively, `80`, `443`, `22`). In addition, every machine has a special name for itself, `localhost`, or `127.0.0.1` . If you want a domain of your own, you can register one; I recommend www.gandi.net .

```
ifconfig  or /sbin/ifconfig          ← shows your IP address (and other interface info). Look at the inet addr for eth0 (first ethernet port).

ping    www.cam.ac.uk                ← send packets to www.cam.ac.uk. How fast does it respond. Any dropped? Ctrl-C to stop.
fping   www.cam.ac.uk                ← is a host alive? fping is designed to be simple to use in scripts.
traceroute www.cam.ac.uk             ← trace the network route from here to there, one hop at a time.
whois   cam.ac.uk                    ← do a WhoIs lookup, to find out about the domain owner and registrar.

telnet towel.blinkenlights.nl        ← use telnet to connect to another machine. Wait and watch. To exit, Ctrl-] then type quit
```

Netcat is used for all sorts of scripted network operations. Here is a simple one, that allows you to **chat** across the network.

- One person should set up netcat to **listen** for incoming connections: `netcat -l -p 10000` , where the chosen port (in this case, 10000) can be anything between 1025 – 40000 that isn't already in use. (Ctrl-C to quit).
- The other should then try to **connect** to it: `netcat localhost 10000` and then you can type back and forth.
- If you are on different IP addresses, then use the IP instead of "`localhost`". [Intervening firewalls may prevent this.]

Network monitoring. Use `tcpdump` (CLI) or `wireshark` (GUI) to see the packets as they travel. Needs to be run as root. For example, you can monitor the passing traffic of the above chat-session with: `sudo tcpdump -vvv -X port 10000`

E-Mail

Email, from first principles. We can "speak" **SMTP** (simple mail transfer protocol) directly to most mail-servers. Provided that we are *within* the cam.ac.uk domain, the outgoing relay, ppsw.cam.ac.uk will trust us implicitly. Try the following. What you type is in green, while explanations are in blue. Change it to suit yourself. SMTP has very simple commands (HELO, MAIL FROM:, etc) and responds with a numeric code and a textual explanation. Your email client normally does this for you.

```
user123@pip:~$ telnet ppsw.cam.ac.uk 25          ← connect to the mail server on the SMTP port, 25
Trying 131.111.8.139...                          } the telnet program is
Connected to ppsw.cam.ac.uk.                     } saying what's happening
Escape character is '^]'.                        } use Ctrl-] to control telnet itself.
220 ppsw-52.csi.cam.ac.uk (ppsw.cam.ac.uk [131.111.8.139]:25)
ESMTP Exim 4.82_3-c0e5623+ppsw+2 Wed, 18 Feb 2015 15:00:44 +0000
HELO www.srcf.net                                ← We are connecting from the machine www.srcf.net
250 ppsw-52.csi.cam.ac.uk Hello pip.srcf.societies.cam.ac.uk [131.111.179.83]
MAIL FROM:user123@cam.ac.uk                      ← Begin message, from <sender>. No space after FROM:
250 OK
RCPT TO:user123@cam.ac.uk                        ← Please transfer this message to <recipient>
250 Accepted
DATA                                             ← This is the text of the message
354 Enter message, ending with "." on a line by itself
This is a test message                           } The text of the message goes here for as long as needed
Bye for now                                      } [attachments are MIME-encoded, and included here]
.                                                ← Don't forget the single trailing dot.
250 OK id=1YO67u-0005hH-E3
QUIT                                             ← We're done. Or you can start again with MAIL FROM:
221 ppsw-52.csi.cam.ac.uk closing connection
Connection closed by foreign host.               ← telnet is telling us that the remote-end disconnected.
```

Note: SMTP will usually only relay mail from IPs within the same domain. There is *no* verification(!) of the sender's email address: it is often a free choice.

Now, just use the **mail** command: `echo "Your msg text" | mail -s "The subject" recipient@cam.ac.uk`
To attach files, use `mutt` (rather than `mail`); see manpage. [Avoid the temptation to repeat a message 100 x in a loop.]

Local mail can be *forwarded*: put the destination address in your `~/.forward` and then `chmod 600 ~/.forward`

To read mail on **Hermes**, you can `ssh user123@hermes.cam.ac.uk` and use `Alpine`, which is *fast*, with *practice*.

To look up a user in the University email directory, use **LDAP** (lightweight directory access protocol) – in one line:
`ldapsearch -x -LLL -H ldap://ldap.lookup.cam.ac.uk -b "ou=people, o=University of Cambridge,dc=cam,dc=ac,dc=uk" "(uid=rn214)" uid cn` ← or similarly, use jackdaw.cam.ac.uk/mailsearch

A simple shell script to email a daily fortune to yourself. An exercise for the reader...

1. Begin the script as above, name it `cookie.sh` . (N.B. the "magic" first line, and a comment about what it does).
2. Use a long fortune, `/usr/games/fortune -l` , for the source of the text, and pipe it to `mail` (as above).
   [N.B. you need to specify the *full* path to fortune; i.e. /usr/games/fortune (find it with `which`), because cron's $PATH doesn't include /usr/games/ .]
3. Test it with `./cookie.sh` (remember to chmod it executable first).
4. To automate it, make use of the **cron daemon**, which runs scheduled commands. [A daemon is a Unix background process, named after classical mythology, or Maxwell's daemon. Crond (pronounced "Cron-D") is a sophisticated and reliable timer.]
5. Add the scheduling rule to your cron table with `crontab -e` (this opens in nano; append to the *end* of the file).
6. The `crontab` file format is self-documented (or see `man 5 crontab` for examples); you will need a line such as:
   `15 07 * * * /home/user123/cookie.sh` ← minute, hour, day, month, weekday, full path to *your* script.
7. To stop the automated messages, remove the line from your crontab, or comment it out with `#` .

SSH Wizardry

The Secure Shell, SSH is *amazing*: it uses *Public Key Cryptography* to allow secure remote logins. If you set up a **key-pair** between machines, then you only have to type your passphrase once per session, and everything is *seamless*.
Public/Private key crypto is brilliantly simple, elegant, and powerful. PuTTY uses SSH, but SSH can do so much more.

- **Key Generation.** Create a keypair with `ssh-keygen -t rsa` . Use a passphrase if it's important.
- **Authorise your key.** Use `ssh-copy-id user123@www.srcf.net` . Now, you don't have to type your password.
- **Shortcuts.** Create a `.~/ssh/config` file containing a `Host/Hostname/Username` stanza. Now simply `ssh srcf`
- **SCP** (secure copy) copies files/directories. You can tab-complete: `scp remoteserver:path/to/file localfile`
- **RSYNC** (remote sync) keeps local and remote directories sync'd, transferring only parts of files that changed. It's really fast, powerful, can tunnel over SSH, and you can now say goodbye to USB keys and *Dropbox* !
- **File Access**. In Cambridge, your PWF (public workstation files) are accessible at: linux.pwf.cam.ac.uk
- **X11**, **VNC**, or **XPRA**: desktop and application forwarding: run GUI applications remotely. `ssh -X` .
- **SSHFS** (ssh filesystem) mounts a remote directory as if it were local. Simply: `sshfs servername: /mnt/localdir`
- **Tunneling**: access one remote system via another, through a firewall: `ssh -L 8080:internal_host:80 gateway` .
- **Remote commands** in one line: on your local machine, run `ssh srcf who` to list the remote users.
- **Printing**. Cat, pipe over ssh, and print with lpr: `cat somefile.pdf | ssh servername lpr -P printername` .
- **Run your own** SSH server (or ssh-daemon): `apt-get install openssh-server` .
- **Encrypting and decrypting** files manually: http://krisjordan.com/essays/encrypting-with-rsa-key-pairs
- For more on SSH setup, see: richardneill.org/a22p-mdk11-0.php#ssh

Miscellany:

- SRCF remote desktop: uses VNC, runs in a browser with Javascript: www.srcf.net/desktop
- *Twitter* has a CLI interface, such as this one: github.com/sferik/t
- Automation services include **Huginn**, **IFTTT**. Data services include **Phant** and **ThingSpeak**.
- Many other web-services have a scriptable API (application programming interface) e.g. to look up an ISBN number and the book information, use the API is described here: isbndb.com/api/v2/docs
- If there isn't an API, you can usually get away with a mix of `curl` and `grep`, for example, to download XKCD.

Web Browsing

Web browsing, from first principles. We can also speak the **HTTP** protocol. Type *fast*: Apache closes idle connections:

```
telnet www.example.org 80          ← Connect to the webserver on the HTTP port, 80. [Actually use the domain "example.org" here].
HEAD /index.html HTTP/1.0          ← Request the document HEADer, for file /index.html with protocol HTTP version 1.0 (or 1.1)
[ENTER]                            ← Needs a double-newline. [You may need to reconnect with telnet, if the server has Keepalive off.]
GET /index.html HTTP/1.0           ← Now get the document body (the full html document, not the same as <body>...<./body>).
[ENTER]                            ← Another double-newline.
Ctrl-]  quit                       ← (For extra fun, observe the network traffic with wireshark while you run this process).
```

What you will see is the raw HTML (hypertext mark-up language), and some HTTP status codes. Try again with a different site, such as www.bbc.co.uk. You can also download files with `wget` or `curl`, and do command-line browsing with `lynx`, `links`, or `w3m`. Sometimes it's useful to do FTP (file transfer protocol): use `lftp`, e.g. `lftp mirrorservice.org`.

Creating Web Pages: HTML and PHP

Websites are a structured set of files and links, written in HTML (*hypertext markup language*). HTML files are made available as web-pages, via a *webserver*, usually *Apache,* though you can open them directly in *Firefox*, or with a text-editor.

On the SRCF, any files in your `~/public_html` directory will be served at http://userid.user.srcf.net . Alternatively, you can put content into your local `/var/www/html/` (or use a symlink), and access at http://localhost . (More advanced sites get their own configuration file, within `/etc/apache2/sites-enabled/` and their own DocumentRoot within `/var/www/`).

Now, create a **static web page**, `nano ~/public_html/index.html` to make the most basic valid **HTML** document:

```
<html>          ← Begin html document. The special characters <> denote an html tag.
<body>          ← Begin body of document.
Hello World     ← Actual text.   [To write a literal <, >, or &, use the entities: &lt; , &gt; , &amp; respectively.]
</body>         ← Closing body tag. "/" denotes that the tag is a closing-tag.
</html>         ← Closing html tag. Always close tags in the reverse order of opening: mis-nesting can cause weird results.
```

Now, visit the URL: *user123*.user.srcf.net/index.html (change *user123* to your own id) in your web-browser, and you'll see it!
You can watch Apache's log files: `tail -f /var/log/apache/user/user123/access.log` and re-load the page.
Note: if a *directory* is requested (e.g. *user123*.user.srcf.net/), then the file **index.html** (if it exists) is the *default*.
For more on HTML, and Web Design (CSS, JavaScript), see the tutorials at www.w3schools.com .

A dynamic web-page, with PHP. PHP scripting is widely used, e.g. by *Facebook* . See php.net ← PHP Hypertext Preprocessor.
Create this file, with `nano ~/public_html/calc.php` (or copy it from `~rn214/public_html/calc.php`):

```
<html>                                                  ← Begin html document as usual. (Nano will colour-highlight).
<head><title>Calculator</title></head>                  ← Set the page title
<body>
<h1>Calculator</h1>                                     ← Headline sized (h1).

<?php                                                   ← Begin PHP interpreter for everything from <?php to ?> .
$x = floatval ($_GET['x']);                             ← Variables in the URL are now in the array $_GET[]
$y = floatval ($_GET['y']);                             ← floatval() sanitises them (for safety). [search: Cross-Site Scripting]
$op = $_GET['op'];                                      ← 'op', 'x', 'y' are the same names as the inputs below.

if ($op){                                               ← If a button was pressed (i.e. $op is non-empty) ...
        if ($op == "add"){                              ← Test the value of $op … [N.B. double-equals for comparisons]
                $ans = $x + $y;     $sym="+";           ← Calculate the answer.
        }elseif ($op == "subtract"){                    ← NB, in PHP, "$" is a "sigil" which just means "this is a variable",
                $ans = $x - $y;     $sym="-";           ←  whereas "$" is a unary operator in bash.
        }elseif ($op == "multiply"){
                $ans = $x * $y;     $sym="*";
        }elseif ($op == "divide"){
                $ans = $x / $y;     $sym="/";           ← PHP does "proper" division, not just integers.
        }else{
                $ans = "ERR";       $sym="?";           ← Remember to handle the unexpected.
        }

        echo "<p>Question: <b>$x $sym $y</b><br>";      ← Print out the question and answer.
        echo "Answer: <b>$ans</b></p>";                 ← Note that we are quoting, and mixing HTML in too.
}
?>                                                      ← End PHP interpreter.

<form method=get action=calc.php>                       ← HTML form for inputs and buttons.
X: <input name=x value=<?=$x?>>                         ← An input field, named x, whose value defaults to
Y: <input name=y value=<?=$y?>>                         ←  the previous values of x and y (via embedded PHP).
<input type=submit name=op value=add>                   ← A submit input is a button.
<input type=submit name=op value=subtract>
<input type=submit name=op value=multiply>
<input type=submit name=op value=divide>
</form>                                                 ← Close out all the tags, in order.

</body>
</html>
```

Now, test: *user123*.user.srcf.net/calc.php . Errors are at: `tail -f /var/log/apache2/error.log | grep user123`
Try adding an extra operator, such as `%` (for remainder), or `sqrt()` (for square-root).
What if the user tries to do "5 / 0" ? This is a **bug**: you should detect the attempt to divide by zero and warn.

Internet of Things. The IoT is becoming reality at last, thanks to the prevalence of *very* cheap hardware, and ubiquitous wireless networking. The available modules are increasingly cheap/tiny, e.g. the £1.86 ESP8266 Wi-Fi module, the C.H.I.P. (**getchip.com**) and the Raspberry Pi Zero W. Here is a simple IoT device/application which is well documented, made from a Raspberry Pi: you may find it a useful prototype/starting point: **richardneill.org/src/dinnerdog** .

Programming In C

C is the most fundamental language of computing (for example, both `bash` and `PHP` are actually written in C). Unlike the others, it has to be compiled before it is run. A short example to demonstrate. Try the following: `nano hello.c`

```
/* Hello World program */                    ← shown with syntax-highlighting colours applied, for clarity.
#include <stdio.h>                           ← stdio is the standard input/output library.
int main () {                                ← C programs always start in the function called main() .
        printf ("Hello World!\n");           ← print, formatted..
        return (0);                          ← return code, on exit. The type is int (integer).
}
```

Now compile it with `gcc -Wall -o hello hello.c` ← Wall enables all warnings; hopefully there were no compiler errors.
Run it with `./hello` ← or, in a single step: `make hello && ./hello`
Take a look at the object file: `xxd hello` ← The hex codes in the object file are CPU instructions.

A more complex program, which connects a command-line to a physics experiment is richardneill.org/src/arduino_delay
Of course, the most complex C program is the Linux Kernel itself: over 15 million lines of code!

Databases and SQL (Structured Query Language)

Databases store data in a structured way. They are widely useful, from configuration files to massive research projects.
For a simple workloads, use *SQLite*, while for more complex tasks, use *PostgreSQL*. (*Never* use a spreadsheet!)
Here is a *very* short demonstration of SQLite:

```
sqlite3 fruit.db                                                    ← Open (or create) an SQLite database file.

create table tbl_fruit (id integer pkey, name text, color text);    ← Create a 3 column table (colname, datatype)
                                                                    ← The Primary Key is a unique integer.
insert into tbl_fruit (name, color) values ("strawberry", "red");
insert into tbl_fruit (name, color) values ("orange",     "orange");
insert into tbl_fruit (name, color) values ("banana",     "yellow");  ← Insert some values into the table.
insert into tbl_fruit (name, color) values ("apple",      "green");   ← Table names begin "tbl_" by convention.
insert into tbl_fruit (name, color) values ("avocado",    "green");   ← The spaces are just for readability.
insert into tbl_fruit (name, color) values ("blueberry",  "blue");    ← Each statements must end with a ";"

select * from tbl_fruit;                                            ← A simple select statement. Gets everything.
select * from tbl_fruit where color == "green";                     ← Select, with rules. Gets apple and avocado.
update tbl_fruit set color = "purple" where name = "blueberry";     ← Update a row (or rows), matching pattern.
delete from tbl_fruit where name = "banana";                        ← Delete matching row(s).
select color, name from tbl_fruit;                                  ← Select specific columns.

.exit                                                               ← Exit the SQLite shell. (or use Ctrl-D).
```

You can also interface directly to SQLite from a shell-script (or from most other languages such as PHP).

```
ingredient=$(echo "select name from tbl_fruit where color='purple';" | sqlite3 fruit.db)
echo $ingredient
```

For more, including *joins*, *foreign-keys, indexes*, *sequences*, *types*, and *constraints*, see sqlite.org .

And now a *very, very* quick start on **PostgreSQL**. This is an extremely powerful, industrial-grade database, see postgresql.org .
1. Installation:

```
sudo apt-get install postgresql postgresql-contrib
```

2. Create a new database-user "*testuser*" and a database "*testdb*" owned by that user:

```
sudo su postgres sh -c "createuser -d testuser"
sudo su postgres sh -c "createdb -U testuser testdb"
```

3. If necessary, allow access. Edit the file: `/etc/postgresql/9.6/main/pg_hba.conf` and add the line:

```
local   all   all   trust
```

and then restart postgresql: `sudo service postgresql restart`

4. Connect to the database. Use `\h` for help on SQL commands, and `\?` for help on the psql interface. Try a command.

```
psql -U testuser testdb
SELECT now() AS date, 'Unipart Digital' AS team, 6*7 AS answer;
SELECT * FROM pg_database;
\q
```

Git: Source Control (Source-Code-Management, SCM)

Git is a tool for sharing *repositories* of source code, such that multiple people can collaboratively edit them, *tracking* and merging changes. Git handles ownership, branches, merge-conflicts, and the abilty to *revert* a change, or view history. You can use Git on your own machines, but it's most useful for teams. See: http://rogerdudler.github.io/git-guide . Here is an overview:

1. Ensure you have ssh enabled to get to the server, and that your user has read/write access to the scm directory on the server.

2. New projects: create an empty central repo on the server: `git init --bare --shared /home/scm/repositoryname.git`

3. Locally, clone the server's repo: `git clone your.gitserver.com/path/to/respositoryname.git` . This will create a new directory, *`repositoryname`* into which a copy of the project source-code has been checked out. It will, also contain a (hidden) `.git/` subdirectory, containing the local git data, and your configuration in `.git/config` .

4. To make a change locally, edit the files as normal, then *add* them to the change-set, then *commit* them, with a helpful commit message. This commit message is really important for large projects. Then *push* your changes to the central server:
```
git add file1 file2 ...  ;  git commit -m "This summarises what/why you changed." ;  git push
```

5. To fetch and apply all changes from other users, pull the changeset from the server: `git pull` .It's possible to do this in 2 steps: `git fetch` , then `git merge` ; this is useful if a merge somehow *conflicts.*

6. Other important git commands (in each case, see e.g. `man git-status` for `git status`) are:

`git mv old_name new_name`	to move/rename a file, while keeping track of the change. Similarly, `git rm` .
`git checkout filename`	to check-out the saved version of a file, *discarding* local uncommitted changes.
`git diff --cached`	to show the changes waiting to be included in the next "git commit".
`git status`	show current sync-state of the local/remote repositories. Also, `git log` and `git blame` .

7. Other concepts: .gitignore, branching, git hooks (e.g. automatically build/test/deploy), git-web (www source browser).

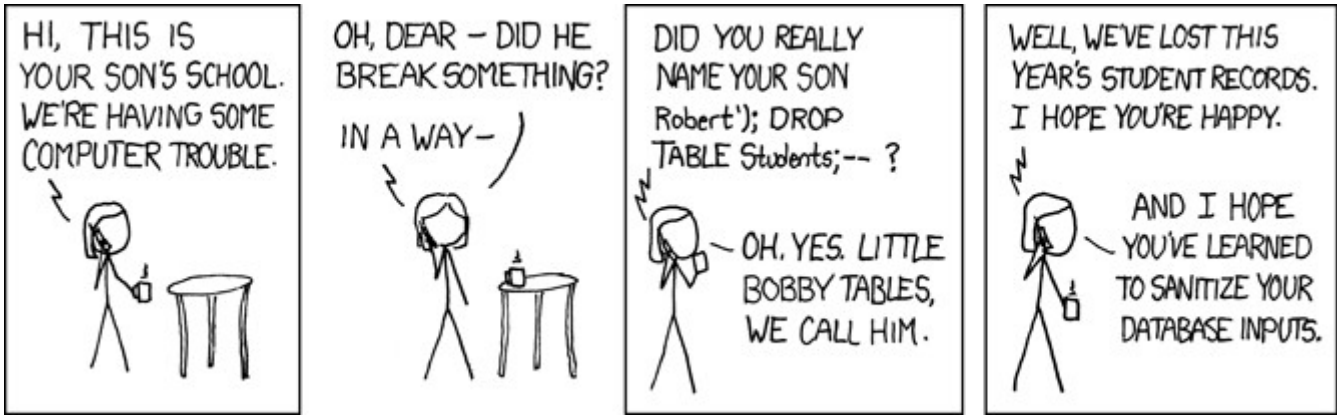
Computer Security

"*If builders built houses the way programmers built programs, the first woodpecker to come along would destroy civilization.*"
- Gerald Weinberg

Cybersecurity has never been more important, and nor has it ever been so **precarious**. The gulf between "best practice" and "actually secure enough" is rather large. In mid-2017, there are really only 2 alternatives: ***Pretend***, and ***Panic***. There are massive technical problems (some example bugs include, "HeartBleed", "GotoFail", "ShellShock"), and this is made worse by wholesale deliberate undermining of our security infrastructure by the malefactors at the NSA, GCHQ etc (see Snowden) and the careless way that the CIA created a suite of cyberweapons and then lost control of them (see "Vault7"). Cloud computing concentrates "eggs" in few "baskets", and often undermines privacy. Also, there are the idiots (e.g. Lenovo/Superfish), the incompetents (insecure IOT enabled in a DDOS against Dyn; Intel Active Management Technology (AMT) accepting empty passwords) and the crooks (e.g. CryptoLocker). Linux is somewhat less vulnerable than Windows. The IETF community has begun to deal with the obvious problems (buffer overflow, SQL injection etc), but the task is vast. See schneier.com and ted.com/talks/mikko_hypponen_how_the_nsa_betrayed_the_world_s_trust_time_to_act .

*A **trusted system*** is one whose failure may *break* your security policy. (i.e. you *must* trust it; it is not necessarily trust*worthy*).

A common example of a failure is when mishandling user-inputs. Always take care with *untrusted user input*: it could be **malicious**. SQL injection is explained further at: bobby-tables.com :



XKCD: 327

And Lastly...

"*Any sufficiently advanced technology is indistinguishable from magic*."
- Arthur C Clarke.

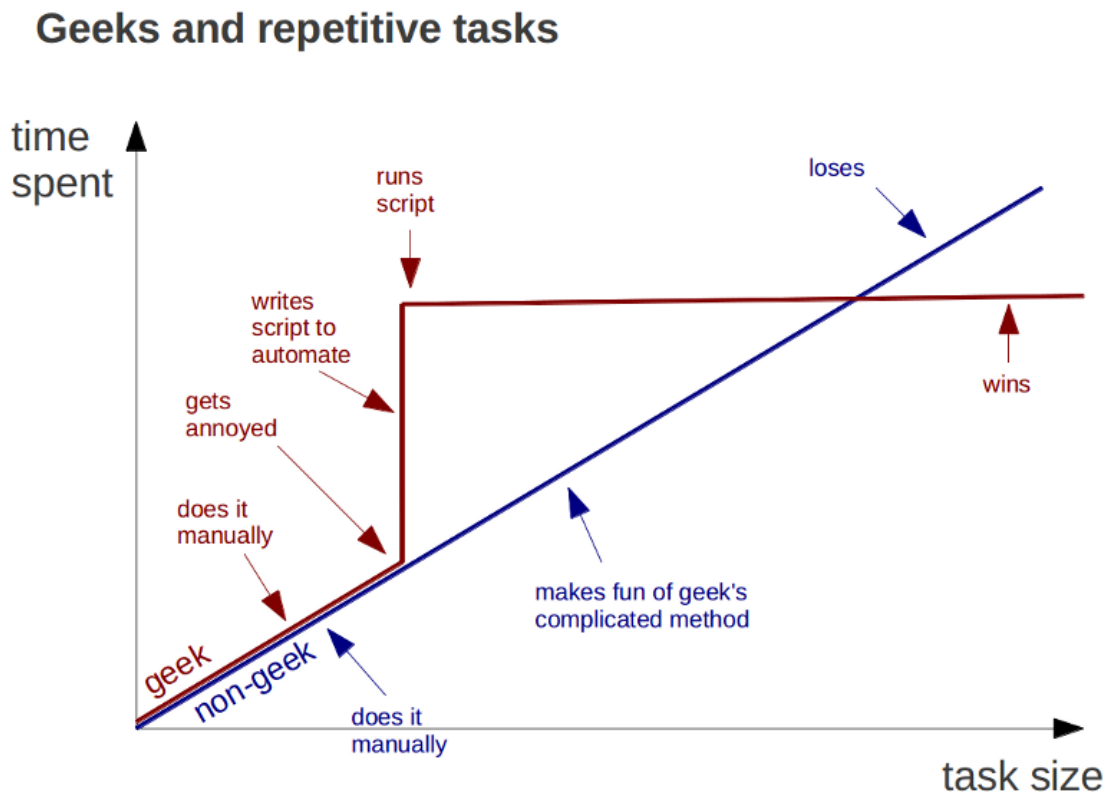
→ Now, you too are in possession of a wand. Use it well.

"Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- Brian Kernighan

→ What this means is that you should ensure your code is elegant, clear, well-structured, and well-commented.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
- C.A.R. Hoare

And, for **amusement**:
GNU Humour: various jokes are at: www.gnu.org/fun
The Jargon File: Unix terms, history and culture: www.catb.org/jargon
Silly programming languages: LOLCODE, INTERCAL, Whitespace, International Obfuscated C Code Contest.
Silly editors: try *Vigor* (inspired by UserFriendly.org) or Butterflies (XKCD #378).
Silly users (PEBKAC, ID-ten-T): www.rinkworks.com/stupid
Silly businesses and developers: thedailywtf.com and the BOFH (see: TheRegister.co.uk)
The Internet Oracle: collaborative humour. internetoracle.org



Source: Bruno Oliveira

An example of Magic: consider the *Hailo* application which basically does "*Accio taxi*". This unites a phenomenal array of dependencies [theory of relativity, space-flight, atomic clocks, microelectronics, GPS, GPS receivers (a few pence per chip, to do billionth-of-second timing on a trillionth of a milliwatt of signal), GCC, libC, the Linux kernel (10k man-years of work) + Android, and an entire industry + supply-chain] – and *then* Hailo write their application on top. Remember quite how amazing this is... and that, if you can program, you too can create amazing things.

~ The End ~