Introduction to the Shell

How to become hackerman in less than 4 hours

Programming Practices for Economics Research

Department of Economics, Univeristy of Zurich

Fall 2018

Learning Objectives

At the end of the session you will:

1. Understand the structure of your computer a bit better
2. Do basic tasks on your computer using the Shell
3. Have an idea of the power of more advanced Shell commands
4. Know where to look up stuff

What is it?

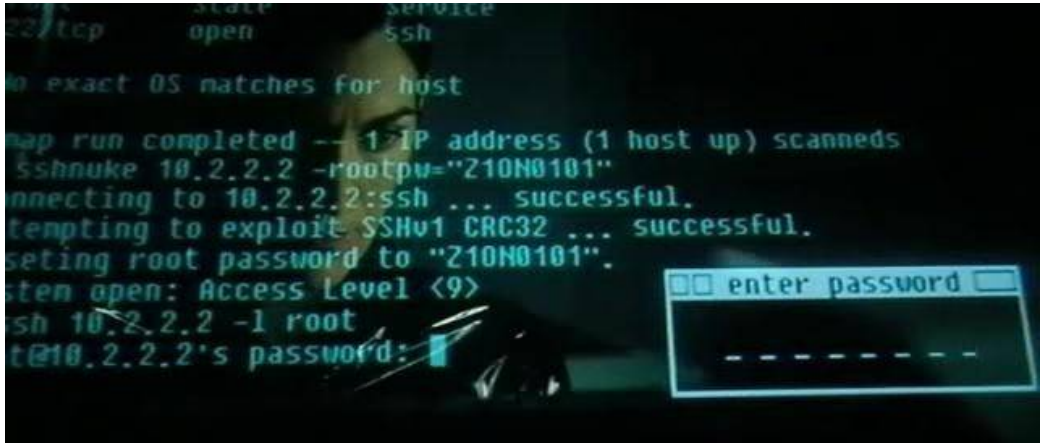

Figure 1: The Matrix

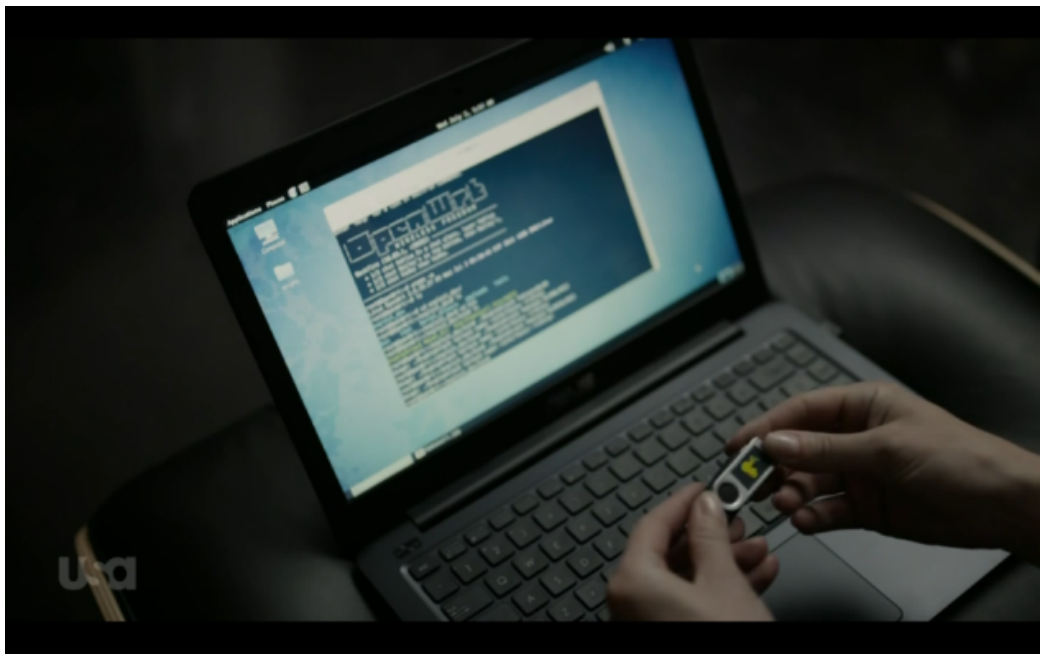

Figure 2: Mr Robot

Background

At a high level, computers do four things:

- run programs
- store data
- communicate with each other
- interact with us

Shell:

The shell is a **comand line interface (CLI)**

- offers a way to interact with the computer via text
- is a program like any other but it's main job is to run other programs
- The most popular Unix shell is bash (Bourne Again SHell).
- It works in a **read-evaluate-print loop (REPL)**.
- When you type a command and press `Return`,
  1. The shell **reads** your command
  2. The shell **evaluates** what it means and executes it
  3. The shell **prints** the output of the command

Why should you learn it?

Many programs and tools do not have a graphical interface. You run them via a shell command, e.g.

- apt-get / homebrew to quickly install software
- snakemake to automatically run a whole research project

Reproducing the first stages of a project (moving files / data by hand) is very hard

- The shell allows us to document the exact order in which every step was taken
- This is especially useful for public / messy data to make your work reproducible

Why should you learn it? #2

The command line is often the easiest way to interact with remote machines

- Allows you to execute projects in the cloud

The shell is very powerful

- allows you to combine existing tools with only a few keystrokes
- to set up pipelines
- to handle large volumes of data automatically

For Windows users

Shells of most Unix derivatives (Linux, OS X) are fairly similar and the basic tools are available

- Linux users just press `Crtl + Alt + T`
- Mac Users press `Cmd + Space` and start typing `terminal`

The Windows shell differs considerably from this

- Download cygwin to use the functionality and the commands of the Linux shell (see the installation guide)

Command structure and types

- `$ command -options (or --longoption) arguments`
- `$ which [executable]`; determines the exact location of an executable

Note:

- a `whitespace` on the command line is an argument separator,
- a `-` starts options,
- a `--` starts longoptions.
- but it's in the programers freedom to violate this standard

Starting and Exiting a program

Starting

- You can start programs in an interactive mode
- e.g. by typing `python`, you just started a python session
- Now you type python code, not shell code

Exiting

- It is therefore imporant to know how to exit a program.
- A program should tell you how, if not normally the following work:
  - The `exit command` for the program, e.g. `quit()` in python or R
  - Press `CTRL+C`
  - Press `CTRL+X`
  - Type `quit` or `q`

... Except for VIM

VIM:

- Press `ESC`, then type `:q!` (without saving) or `:x` (with saving)

Hands on

Getting help

- `$ whatis [command]`; display a brief description of a command
- `$ apropos [string]`; search the whatis database
- `$ man [executable]`; most executables provide a piece of documentation, called the manual page. Don't google a command, rtfm
- `$ help [builtins]`; help facility for shell builtins
- `$ [executable] --help`; option that displays a description of the command's supported syntax and options

Files and Directories

Before we get started...

- the part of the Operating System that handles files and directories is called the filesystem
- We differentiate between files which hold information and directories (or folders) which hold files
- A handful of commands are used frequently to interact with these structures. You will know them by the end of the lecture

Basic Bash

- The dollar sign stands for a prompt waiting for input

```
$
```

- Type `whoami` and press Enter to see how the current user is named

```
$ whoami
```

Basic Bash

- When type `whoami` the shell finds the program
- The program is run
- The output of the program is shown
- A new prompt is displayed, indicating that it's ready for new commands

Basic Bash

- To know where in the filesytem you are type `pwd` (print working directory)

```
$ pwd
```

Directory structure

- To understand what our home directory is, let's look at the directory structure
- It is organized as a tree with the root directory / at the very top
- Everything else is contained in it
- / refers to the leading slash in /Users/me (Mac and Linux) or /cygdrive (Windows with cygwin)

Directory structure

Mac and Linux:

- Underneath /Users the data of the other user accounts on the machine is stored
- E.g. /Users/someusername
- If we see /Users/me, we are inside /Users because of the first part of its name. Similarly, /Users resides in the root /

Windows:

- Underneath /cygdrive you find the drives of your system (i.e, C, D, etc.)

What files are in the directory?

- list directory contents

```
$ ls [directory ...]
```

- important options:
  - `-F` (for flag); distinguish directories ('/'), executables ('*'), symbolic links, etc.
  - `-a` (for all); include directory entries whose names begin with a dot (i.e., .git)
  - `-l` (for long); prints the output in the long format
  - `-h` (for human readable): prints filesize in KB, MB, GB, TB instead of #Bytes
  - `-d` (for directories): show directories only

How can I change my working directory?

- to change your working directory

```
$ cd [directory]
```

- some shortcuts:
  - change to the current directory: `$ cd .`
  - change to the parent directory: `$ cd ..`
  - change to the home directory: `$ cd ~` | `cd`
  - change to previous directory: `$ cd -`
  - tab completion (press TAB once, twice, ALT+*)

How can I view the content of a file?

- View the file in the shell

```
$ less [filename]
```

NOTE: `man` uses `less` to show the manual page

- to navigate in less:
  - space: jump a page
  - b: jump a page back
  - /: search and highlight string in file/manualpage
  - q: quit
- Print out the file into the shell

```
$ cat [filename]
$ tail [filename]
$ head [filename]
$ more [filename] # less is more more
```

In action...

- Navigate to your home directory
- list the files in your home directory
- go to Nelle's Data, read some of her .txt files
- read the haiku.txt file in Nelle's writing folder

Creating Stuff

The Atom editor

What you should have got from the installation guide:

- download Atom
- the command palette: `CMD+SHIFT+P`

Add `atom` command to the shell (Mac and Linux)

- run `atom --help`
- enter the following commands to make Atom your default editor:

```
$ export EDITOR='atom -w
$ export TEXEDIT='atom'
$ alias atom="atom --new-window"
```

- those settings will only be active for the current session. If you want to make them persistent, you can copy those terms into your `.bash_profile` in your home directory

Create a new file

```
$ touch [filename]
$ touch myproject/data.txt
```

Remove a file or a directory

```
$ rm [filename | directory]
```

- there is **no undelete**
- important options
  - `-i` (for interactive); request confirmation before removing
  - `-v` (for verbose); show files which are being removed
  - `-r` (for recursive); required for directories; attempt to remove the file hierarchy rooted in each file argument
- Exmaples:

```
$ rm somefile.txt
$ rm some-subfolder/somefile.txt
$ rm -r some-directory/
```

Create or remove an empty directory

```
$ mkdir [directory] | rmdir [directory]
```

Copy file, or copy files to directory

```
$ cp [source file ...] [target file | target directory]
```

- important options
  - `-i` ; ask for permission before overwriting
  - `-r` ; required for directories
  - `-u` (for update); copy files that don't exist or are modified than in the existing directory
  - `-v` ; display messages
- Examples:

```
$ cp somefile.txt ../some-other-directory/samename.txt
```

Rename files and directories, or move files to directory

```
$ mv [filename ...] [target file | target directory]
```

- important options
  - `-i` ; ask for permission before overwriting
  - `-v` ; display messages
- Examples:

```
$ mv somefile.txt someothername.txt
$ mv data.{csv,backup}
```

Wildcards

- Working with shell commands becomes powerful when you work with wildcards
- Wildcards are special characters that help you to rapidly specify groups of filenames
- Four important wildcards are:
  - any character: `*`
  - any single character: `?`
  - any character that is a member of the set characters: `[characters]`
  - any character that is *not* a member of the set characters: `[!characters]`

Wildcards Examples

- Here are some examples

type	results
`*`	all files
`g*`	any file beginning with `g`
`b*.txt`	Any file beginning with `b` followed by any characters and ending with `.txt`
`Data???`	Any file beginning with `Data` followed by exactly three characters
`[abc]*`	Any file beginning with either `a`, `b`, or `c`

In action...

- In your home folder, create a new folder, create a .txt file, open the file using atom, type some stuff, save it, rename the file, delete the file.
- use wildcards to copy all .txt files from the exercise folder to your folder
- rename some files, create some backups
- delete the folder you created

Redirections, Pipes, and Filters

I/O redirections

- Most of our programs read your input, execute it, and print output
- We call the input facility *standard input*, which by default is your keyboard
- Our programs send their results to a special file called *standard output*, which by default is print to the screen and not saved into the hard disk
- I/O redirection allows us to redefine where standard output goes. For example,
  - redirect the output to a file instead of to the screen

```
$ ls -l [directory] > [filename.txt]
```

- or redirect the output to a file and appends instead of rewriting it

```
$ ls -l [directory] >> [filename.txt]
```

Read files sequentially and print the output in a file

```
$ cat table0* > table.txt
```

Pipelines

- The standard output of one command can be *piped* into the standard input of another using the pipe operator |
- The general structure is

```
$ command | command
```

- For example,

```
$ ls -l Data | less
$ history | grep cp
```

- Pipes allow you to do complex data manipulations in one line, the pipeline

Filters

- When working with pipelines, it is often useful to use filters
- Filters take input, change it somehow, and then output it
- Some useful filters are the following:
  - `$ sort`
  - `$ uniq`
  - `$ wc`
  - `$ head` and `$ tail`

sort

- sort lines of text files and writes to standard output; it does not change the file

```
$ sort [filename]
```

- some options:
  - `-f` (for fold); fold lower case to upper case characters
  - `-n` (for numerical); compare according to string numerical value
  - `-r` ; reverse the result of comparisons

uniq

- report or filter out repeated lines in a file

```
$ uniq [input file] [ouput file]
```

- often used with `sort`

```
$ ls file1 data/file2  | sort | uniq | less
```

```
* `-d` (for duplicates); print list of duplicates
```

wc

```
$ wc [file] ...
```

- count number of words, lines, characters, and bytes count
  - `-w`: words
  - `-l`: lines
  - `-m`: characters
- example:

```
ls file1 data/file2 | sort | uniq | wc -l > lines.txt
```

head and tail

- print first / last part of files; by default 10 lines

```
$ head [file ...]
```

and

```
$ tail [file ...]
```

- `-n [count]`; determines the number of lines you want to print
- `-f [follow]`; display the file and update if the files get updated

In action...

- Make a subdirectory, navigate to it, copy the data .txt files from Nelle's Data into it.
- Create a file that contains the line counts of planets.txt
- how many unique salmons are in the salmon.txt file

print out the the argument on standard output

```
$ echo
```

- print out hello world

```
$ echo hello world
```

- pathname expansion; print any file in the working directory

```
$ echo *
```

- print all hidden files

```
$ echo .*
```

- parameter expansion; print the variable `USER`

```
$ echo $USER
```

- command substitution; print the output of `ls`

```
$ echo $(ls)
```

A note on naming files

- consider the file *two words.txt*. If you use this on the command line, the shell will treat this as two separate arguments

```
$ ls -l two words.txt
```

- use double quotes to suppress word splitting.

```
$ ls -l "two words.txt"
```

- best practice:

```
$ mv "two words.txt" two_words.txt
```

Troubleshooting: spacing, double quotes "", and escaping characters

- consider `$ echo this is a      test`.
  - the Shell removes the extra whitespace
  - use `$ echo "this is a      test"`
- consider `$ echo The total is $100`
  - the Shell views $1 as a parameter and, by parameter expansion, substitutes an empty string
  - use `$ echo The total is \$100`
  - **NOTE**: the \ backslash starts the so called escape sequence, e.g. for whitspace \

A note on quotes and expansion

- `$ echo text $USER has files in ~/* directory`
- `$ echo "text $USER has files in ~/* directory"`
- `$ echo 'text $USER has files in ~/* directory'`
- with each level of quoting, more and more expansion will be suppressed.

View the list of your last 500 commands

```
$ history
```

- `!4` ; the Shell expands this into the content of the 4th line in the history list and repeats it
- `!!` ; or arrow up and `ENTER` to repeat the last command
- `sudo !!` ; to give elevated privileges to command
- `!$` ; last argument, e.g.
- `mkdir test;`
- `cd !$;`

Keyboard shortcuts

- `CTRL-A` ; move the cursor to the beginning of the line
- `CTRL-E` ; move the cursor to the end of the line
- `CTRL-K` ; delete everything to the left
- `CTRL-U` ; delete everything to the right, and paste it on `CTRL-Y`
- `CTRL-C` ; abort current execution of running process
- `CTRL-R` ; reverse search through command history
- `CTRL-X,E` ; open and edit current command in an editor, execute on editor close

Shell Scripts

Writing and running a bash script

- write your script in the atom editor, selecting the shell syntax, or start an editor from the shell:

```
$ atom somescript.sh
```

- start with the “shebang”

```
#!/usr/bin/env bash
```

- run the script

```
$ bash somescript.sh
```

- to check the content

```
$ cat somescript.sh
```

Some notation: `$1`, `$@`, and

- when the script contains `$1`, then `$ bash somescript.sh file.txt` will use the first file or parameter on the command line
- when the script contains `$@`, then `$ bash somescript.sh *.txt` will be use all files or parameters on the command line
- do your future self a favour, comment your script using #

Write a useful script. . .

. . . that automates a tedious task for you.

- for example, write a shell script that creates a backup of Nelle's folder

Finding Stuff & REGEX

find files in path and below which match an expression

```
$ find [path] [expression]
```

- helpful versions:
  - `$ find . -type d`; find directories in current working directory
  - `$ find . -type f`; find files in current working directory
  - `$ find . -maxdepth 1 -type f`; restrict the depth of search to current level
  - `$ find . -mindepth 2 -type f`; find all files that are two or more levels below
  - `$ find . -name *.txt`; find all txt files

Print lines which match a pattern

```
$ grep [pattern] [file ...]
```

- example: print lines containing “beta”:

```
$ grep beta results.txt
$ history | grep find
```

- Options include:
  - `-w` word; restrict matches to lines containing the word on its own (i.e., if beta, not beta1)
  - `-i` insensitive; makes search case-insensitive
  - `-n` number; number the lines that match
  - `-v` invert; print the lines that do not match
  - with "" phrase;
- check `man grep`

Regular expressions

- `grep` becomes powerful when combined with *regular expressions*
- Regex are used to identify regular strings; this can be exceptionally handy for quickly scanning datasets to look for specific strings, i.e., phone numbers or email addresses.

Regular expressions

- What is a regular string? It's any string that can be generated by a series of linear rules, such as:
  1. Write the letter "a" at least once.
  2. Append to this the letter "b" exactly five times.
  3. Append to this the letter "c" any even number of times.
  4. Optionally, write the letter "d" at the end.
- Strings that follow these rules are: "aaaabbbbbccccd," "aabbbbbcc," and so on (there are an infinite number of variations). Regular Expressions are a shorthand way of expressing these sets of rules, here:

```
aa*bbbbb(cc)*c(d | )
```

Regular expressions

- Regex are supported by many command-line tools and byt most programming languages, however not all regular expressions are the same; they vary slightly from tool to tool and from programming language to programming language.
- Understand the concept, get manual for specific implementation

Classic example: identify email addresses

- Rule 1: The first part of an email address contains at least one of the following: uppercase letters, lowercase letters, the numbers 0-9, periods (.), plus signs (+), or underscores (_).
- Rule 2: After this, the email address contains the @ symbol.
- Rule 3: The email address then must contain at least one uppercase or lowercase letter.
- Rule 4: This is followed by a period (.).
- Rule 5: Finally, the email address ends with *com, org, edu,* or *net* (in reality, there are many possible top-level domains, but, these four should suffice for the sake of example).

Solution:

```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
*	Matches the preceding character, subexpression, or bracketed character, 0 or more times	a*b*	aaaaaaaa, aaabbbbb, bbbbbb
+	Matches the preceding character, subexpression, or bracketed character, 1 or more times	a+b+	aaaaaaaab, aaabbbbb, abbbbbb
[]	Matches any character within the brackets (i.e., "Pick any one of these things")	[A-Z]*	APPLE, CAPITALS, QWERTY

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`()`	A grouped subexpression (these are evaluated first, in the "order of operations" of regular expressions)	`(a*b)*`	`aaabaab`, `abaaab`, `ababaaaaab`
`{m, n}`	Matches the preceding character, subexpression, or bracketed character between m and n times (inclusive)	`a{2,3}b{2,3}`	`aabbb`, `aaabbb`, `aabb`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`[^]`	Matches any single character that is not in the brackets	`[^A-Z]*`	`apple`, `lowercase`, `qwerty`
`|`	Matches any character, string of characters, or subexpression, separated by the "|" (a vertical bar, or "pipe," not a capital "i")	`b(a|i|e)d`	`bad`, `bid`, `bed`
`.`	Matches any single character (including symbols, numbers, a space, etc.)	`b.d`	`bad`, `bzd`, `b$d`, `b d`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
^	Indicates that a character or subexpression occurs at the beginning of a string	`^a`	`apple, asdf, a`
\	An escape character (this allows you to use "special" characters as their literal meaning)	`\. | \\`	`. | \`

Commonly used regular expressions

Symbols	Meaning	Example	Ex Matches
`$`	Often used at the end of a regular expression, it means "match this up to the end of the string." Without it, every regular expression has a defacto `".*"` at the end of it, accepting strings where only the first part of the string matches.	`[A-Z]*[a-z]*$`	`ABCabc`, `zzzyx`, `Bob`

Commonly used regular expressions

Symb	Meaning	Example	Ex Matches
`?!`	"Does not contain." This pairing of symbols, immediately preceding a character (or regular expression), indicates that that character should not be found in that specific place in the larger string. If trying to eliminate a character entirely, use in conjunction with a ^ and $ at either end.	`^((?![A-Z]).)*$`	`no-caps-here`, `$ymb0ls a4e f!ne`

Let's practice

Go to Nelle's Data

- find a file which matches a pattern
- print lines which match a pattern
- play with regex

Where to dig deeper?

- Here are two good books to look up stuff:
  - Newham and Rosenblatt (2005)
  - Shotts Jr (2012)

Recap

1. Do you understand the tree structure of your operating system?
2. Do you value the potential of the shell?
3. Can you do simple stuff using the shell?
4. Do know where to look if you want to learn more?

Acknowledgements

- This course is designed after and borrows a lot from:
  - Effective Programming Practices for Economists, a course by Hans-Martin von Gaudecker
  - Software Carpentry and Data Carpentry designed by Greg Wilson
  - Shotts, W.E. (2012). The Linux Command Line. San Francisco: No Starch Press.
- The course material from above sources is made available under a Creative Commons Attribution License, as is this courses material.

Programming Practices Team

Programming Practices for Economics Research was created by

* Lachlan Deer
* Adrian Etter
* Julian Langer
* Max Winkler

at the Department of Economics, University of Zurich. These slides are from the 2017 edition.

Bash Cheat Sheet

cd

- `cd` change working directory. Without options: go to home directory
- `cd dir` change into dir
- `cd ..` change to parent

ls

- `ls` list contents of current directory
- `ls dir` list contents of dir
- `ls -l` list in long format
- `ls -a` list all files
- `ls -R` recursively list files in subdirectories
- `ls -d` don't go into subdirectories, just list them
- `ls -S` list by size
- `ls -t` list by modification date

manpage aka rtfm

- `man cmd` get help for command cmd

create / manipulate timestamp

- `touch f` if f exists: update modification date. Otherwise create a new empty file f

copy

- `cp` copies files
- `cp a b` copy file a to b
- `cp a b c dir/` copy files abc into `dir/`
- `cp -R old new` recursively copies directory old into new
- `cp -i a b` ask before overwriting files

move

- `mv` moves files
- `mv a b` move file a to b
- `mv a b cdir/` move files abc into `dir/`
- `mv -i a b` ask before overwriting files
- `rm` removes files
- `mv a` remove file a
- `rm -r dir/`recursively delete directory dir and all its contents
- `rm -i a` ask before removing files

create / delete direcotries

- `mkdir d` create directory d
- `rmdir d` remove directory d (only works on empty directories)

check file content

- `cat f` write f to screen
- `less f` display contents of f, with paging, keys: space for next page, b goes up, q for exit, I to search
- `open f` open file with associated program (Mac OS only)

reset

- `reset` terminal if messed up by eg binary output

wildcards

are replaced by bash by matching filenames

- `*` matches any string
  - `*txt` matches all .txt files
  - `a*` matches all files starting with a
- `?` matches a single character
  - `doc_v?.txt` matches doc_vl.txt, doc_v2,txt, doc_va.txt etc.
- `[ac5]` matches one of a, c, or 5
- `[a-z]` matches a lowercase letter
- `[a-zA-Z]` matches any letter
- `[0-9]` matches any digit
- `[^A0-9]` carets inverts meaning: this matches any character that is not a digit

braces

use this to generate strings

- `c{a,u}t` expanded to `cat cut`
- `c{1..4}t` range: expanded to `c1t c2t c3t c4t`

Tip: use the `echo` command to try out wildcards/braces.

output redirection

send output to a file

- `>` overwrite
- `>>` append
- `ld > f` saves output to file f. If it exists, f will be overwritten
- `ls >> f` appends output to file f.

input redirection

get input from file

- `grep x < file` equivalent to `grep x file`
- `tr a b < old> new` get the input for tr from file `old` and save output to `new`
  - `this is necessary because tr does not accept a file`

pipe

redirect output from one program to input of another program

- `ls | grep hello` puts output of ls through grep

command substitution

put output of command on command line ()

- `cat $($Is -rt I tail -n 1)` The part in braces outputs the filename of the last modified file.
  - `cat will get that filename as its argument`

command chaining

- `;` put multiple commands on a single line
- `&&` chain on success
- `||` chain on error
- `touch a; ls` first run `touch`, then `ls`
- `pandoc cheatsheet.md -s -o cheatsheet.pdf && open cheatsheet.pdf` if `pandoc` ran smoothly it will open the pdf

Keys

Key	Description	Key	Description
`Ctrl+L`	`Clear Screen`	‘Ctrl+A	‘ | Jump to the beginning of line
`Ctrl+C`	`End process`	‘Ctrl+E	‘ | Jump to the end of line
`Ctrl+Z`	`Suspend process`	‘Ctrl+X	X‘| Toggle between the start of line and current cursor position
`Up` or ‘C	trl+P‘ | History back	‘Ctrl+K	‘ | Cut to the right
`Down` or	`Ctrl+N`| History forward	‘Ctrl+U	‘ | Cut to the left
`Ctrl+R`	`Search history`	‘Ctrl+W	‘ | Cut word to the left
`Ctrl+_`	`Step back, undo`	‘Ctrl+Y	‘ | Past the last cut

sort

sort input, without argument sorts alphabetically

- `sort -n`; sort numerically
- `sort -r`; reverse sort
- `sort -k2`; sort by second column
- `sort -k2 -t,`; sort by second column, and set delimiter to `,`. Usefull vor csv

uniq

only shows unique elements of a list

- `uniq -c` print count of repetitions

grep

search text

- `grep somestring file`; prints every line in file `file` containing string `somestring`
- `grep somestring *`; prints every line in all files matching `*` in the current directory containing string `somestring`
- `grep -i file` case-insensitive search
- `grep -c file` print number of matching lines
- `grep -v file` invert meaning of search: will filter out matching lines
- `grep -l file` only list files containing string $=>$ less time consuming
- `grep -n file` precede matching line with line number
- `grep 'my string' -r path` Recursively search files in `path` for string `my string`

head and tail

print either the first few or the last few lines of a file

- `head myfile.csv`; print the first 10 lines of file `myfile.csv`
- `head -n 5 myfile.csv`; print the first 5 lines of file `myfile.csv`
- `tail myfile.csv`; print the last 10 lines of file `myfile.csv`
- `tail -n 15 myfile.csv`; print the last 15 lines of file `myfile.csv`
- `tail -f myfile.csv`; print the last 10 lines of file `myfile.csv` and append new lines if lines get appended

find

find files and folder

- `find path`; lists all files in all subdirectories of `path`
- `find . -name "*.txt"`; finds all `.txt` files under the current directory
- `find path -name "*.txt" -mtime -60s -a -mtime -120s`; find all `.txt` files in the folder `path` that are older than 60 seconds but newer then 120 seconds

Example: find file that changed during an action

Shows all changed things which are newer then the created timestamp in the tmp folder.

```
touch /tmp/timestamp
*do stuff*
find /path/to/search/for/changes -newer /tmp/timestamp
```

stream editor

read a file, do changes and print it to the standard output

- sed `'s/Glacier/Lake/n' lakes.txt`; changes all occurrences of `Glacier` in file `lakes.txt` to `Lake`

Newham, Cameron, and Bill Rosenblatt. 2005. *Learning the Bash Shell: Unix Shell Programming*. " O'Reilly Media, Inc.".

Shotts Jr, William E. 2012. *The Linux Command Line: A Complete Introduction*. No Starch Press.