The Unix terminal - Notes by Jan Marek

TABLE OF CONTENTS

A Brief Tour of the Terminal

- ~ is the terminal way of saying “your home directory”.
- $ divides the prompt from your own typing. Everything after this dollar sign is your
- text.
- On Macs, for example,
    - programs live in /Applications,
    - configuration data live in /Library,
    - and user data lives in /Users, for example /Users/jan.
    - Inside your home directory will be your desktop (e.g. /Users/jan/Desktop),
    - your documents folder (/Users/jan/Documents)
    - and so on.
- Because the home directory is so commonly used, it gets an alias in the form of ~ so rather than writing /Users/jan/Desktop I can just write ~/Desktop to mean the same thing.
- ls is short for list – it lists files. By default, ls lists the files in the current directory, but you can also list files in other directories.
- You can move between directories by using the cd command, which is short for “change directory”.
- you can use the pwd command (“print working directory”) to see the path.
- cd
    - 1. No matter where you are, you can always move to the parent directory by writing cd ..
    - 2. when you want to go back to the home directory you can just type cd ~.
    - 3. cd automatically keeps track of your previous directory. If you want to return to it, use cd - – that’s a dash.
    - 4. Moving to your home directory is in fact so common that just typing cd by itself with no directory name goes back to your home directory.
- The special .. usage is called a pseudo-directory – it looks like a directory name, but it isn’t one really, and instead is understood to mean “the parent directory.” There’s one other pseudo- directory, ., which means “the current directory.” This is important for security reasons: if you downloaded a program called “ls”, the system wouldn’t run it when you typed “ls” – it would run the system ls command. If you really wanted to run the one you downloaded, you need to use the . pseudo-directory, like this: ./ls. Without this precaution, you could accidentally run malware that happened to have the same name as a system command.
- any files that start with “.” are considered to be hidden – you need to instruct ls to show hidden files in order to show . and .. and I’ll talk more on that later.
- Pro tip: If you want to impress your Unix friends, try out the pushd and popd commands. Use pushd like you use cd, e.g. pushd Desktop. It changes into that directory, but also remembers all the previous directories you have used. You can then unwind the directory stack by running popd by itself, which returns you to the previous directory you were in. Use popd again and again to return to successive previous directories.

Easier navigation

1. rather than typing cd Desktop you can in fact type cd De then press Tab to have the terminal write the rest of the word for you. If there are multiple possible matches, the terminal will complete as much as it can.
2. if you want to re-run a command you typed previously, you can use the up and down cursor keys to browse through previous commands. When you find one you want, just hit return to run it again. If you have run lots of commands, you can search instead: press Ctrl+r (hold down Ctrl then press the “r” key), to enter search mode, then start typing to match previous commands. Again, when you find the one you want, just hit return to run it.
3. although the manual pages for terminal commands are often long and even incoherent, there is one shorter command that is useful: whatis tells you the purpose of a single command, effectively summarizing the manual page in one line of text. For example, there’s a command called “mkdir” that makes directories, but if you weren’t sure what it did you could run whatis mkdir and the system would report “whatis(1) – make directories.” (If you were wondering, mkdir somedir makes a new directory named “somedir”.)
4. if you want to execute a long-running command but don’t want to wait for it to finish, you can add an ampersand to the end of the command to make it run in the background. The command will appear to finish immediately and you’ll be able to type in new commands, but in reality it’s still running in the background.
5. Commands that affect parts of the system outside of your user account require administrator access. For example, if you try to create a directory outside of your home directory, the command will fail because you don’t have permission. If you are a system administrator – which, if it’s your own computer, you almost certainly are – you can run the command using “super user” privileges, which will ask for your password and run the command as an administrator. To do this, first ensure the command is safe: if you move or delete critical files, your system will break. Once you’re sure it’s safe, type the same command again, but prefix it with sudo, which is short for “super user do”. For example:
    - sudo ls
    - That will print the contents of the current directory, but will do it as an administrator. The result will be identical, but it’s a safe way to test out running commands as an administrator.

Reading file contents

- cat, which concatenates files. In practice this means it prints out the contents of one or more files
- Without the line break, the command prompt starts on the same line as the file's contents, which make for hard reading – so please put the line break back.
- We can print two at a time, like this:
  - cat filea.txt fileb.txt
- The contents of each file just gets sent to output, one after the other. This is helpful because the terminal allows you to redirect your output so that it's written to a file instead of printed for the user to read. To do this, use > followed by a filename. For example, this command creates filec.txt by merging the contents of filea.txt and fileb.txt:
  - cat filea.txt fileb.txt > filec.txt
- An alternative is to redirect using >> which appends rather than overwrites the file
- The cat command has two options that are frequently useful:
  - -s removes blank lines,
  - and -n numbers the output.
  - The line numbers are counted individually for each file, so printing two one-line files will give them both the line number 1.
- Warning: Redirecting a file back to itself is a bad idea unless you're very careful. For example, if you ran cat filea.txt > filea.txt you would end up with a completely blank file. This is because your terminal prepares the redirect before anything else happens, which means the first thing it does is clear filea.txt. So, by the time the cat filea.txt part executes, the file is already empty.

Paging Through Output

- However, the file ought to be quite long, so you'll see it scroll off your screen quickly. Thanks to modern user interfaces, your terminal probably has a scroll bar so you can move back and forth over the output. Not like before.
- This is where less comes in: it reads a file in, but allows you to scroll up and down to read its full content more comfortably. This is helpful even with graphical scrollbars, because it means your fingers can stay on the keyboard rather than moving to and from your mouse or trackpad.
    - Using less is just like using cat:
        - less filec.txt
    - Once it's running, use the up and down cursor keys to scroll around, or press "q" to exit.
- less is actually extraordinary powerful: it has options for every letter of the alphabet, which means pressing keys from "a" to "z" all do different things, and in fact many letters even do different things depending on whether they are used in uppercase or lowercase.
- 52 options - They can be split into two categories:
    - parameters you pass to less when you run it,
    - and commands you run when less is running.
- parameters you pass before you run less, only three worth learning.
    - First, use -N (not -n!) to enable line numbering inside less, for example:
        - less -N filec.txt
    - less -M filec.txt
        - see the filename you're viewing, the range of lines currently visible, the total number of lines in the file, and a percentage of how far you are through it.
    - You can use the two parameters together either by specifying them individually or together. So, both of these are identical:
        - less -N -M filec.txt
        - less -NM filec.txt
    - The last pre-run less parameter you should know is +, which lets you send commands that less should run after it starts.
        - + lets you specify post-run commands to run on the command prompt
- / enters search mode, and pressing / then return repeats your previous search.
- if you want to do a backwards search, you need to use ? instead.
- The less command has a few different ways of jumping to a particular point in a file.
    - you can type a number then "g" to go to a particular line, e.g. "50g" will jump to line 50.
    - You can also use "p" to specify a percent, so "50p" will jump to the half-way point in the file.
- If you intend to work with the same file for a little while, a more powerful way of navigating is by using markers.

    - These are invisible bookmarks placed inside a file by less so you can jump around faster, but they don't get saved so they won't affect the file.
    - To place a marker, press "m" then one of the 52 letters from A-Z and a-z (it's case sensitive!).
    - To jump back to a marker, type ' (an apostrophe) then the letter you used to place your marker.
- There are two more useful ways to use less before we're done. First you can have it read multiple files at the same time just by listing more on the command prompt. For example:
    - less -M filea.txt fileb.txt filec.txt
    - When you have several files open:
        - press :n to go to the next file
        - :p to go to the previous one.
        - You can add more open files by using :e somefile.txt, tab completion works here too.
        - Once you're finished with a file, you can remove it from the list of open files with :d.
- You can also search across files – i.e., search for a word in any of the files you have open – by using /*,
    - for example, /*the will search for text in any open file.
    - Annoyingly, repeating a cross-file search uses silly keystrokes:
        - you need to press Escape then either "n" (for searching forwards) or "N" (for backwards).
- Finally, one last thing:
    - you can launch a terminal from inside less by typing ! then pressing return.
    - You can go ahead and run as many commands as you want, then press Ctrl+D to exit.
    - If you want to run one specific command, e.g. ls, use this: !ls.
        - That will run the command and return immediately.
        - If you want to refer to the file you're currently viewing in less, use % – that will automatically be replaced with the filename.

Printing parts of files: head and tail

- You've seen how cat prints a whole file, and less prints a whole file but gives you the ability to scroll around and search. Now let's look at head and tail, which are commands that print only the start or end of a file respectively.
- By default, both commands print the first or last ten lines of a file.
    - head filec.txt
    - tail filec.txt
- If you want to read more or fewer lines, use the -n parameter followed by a number. For example, this will print the final five lines of our file:
    - tail -n 5 filec.txt
- Where tail becomes really useful is when you use its -f parameter, because that enables "follow" mode:
    - the program continues to run until you press Ctrl+D, and any new additions to the file automatically get printed out.
    - If you specify two or more filenames for follow mode, tail will watch them both and tell you when either of them changes.

Counting Lines and Words

- If you want to count the number of lines, words, and characters a file, you should use the wc command, like this:
    - wc filec.txt
- You'll see output like this:
    - 26   182   1001 filec.txt
- The first column shows the number of lines, the second the number of words, and the third the number of characters. The filename is printed at the end because you can count multiple things – and when you do, you'll automatically get a total line for each of the columns.
- If you want to count only lines, only words, or only characters, use either -l, -w, or -c. For example, this counts the words in the file:
    - wc -w filec.txt

Listing files intelligently

- You've already seen the ls command, but in order to be a productive terminal user you need to learn two new things:
    - wildcards for filenames,
    - and ls parameters.
- Wildcards. There are two that matter:
    - * means "any characters",
    - and ? means "any single character."
    - You can use this to work on a group of files at the same time.
- You've already seen that ls lists all the files in the current directory. But if you wanted to show only files that start with "D" (Desktop, Documents, etc), you would use this:
    - ls D*
- You can place that * anywhere in your filename, and have it filter appropriately. Some more examples:
    - ls *.md → will list only Markdown files
    - ls *.txt *.xml → only files that end with .txt or .xml
    - ls f*e* → files that begin with the text "f" then any letters, then "e" then any more letters
- If you find you need to specify lots of alternatives regularly, you can also use brace expansion like this:
    - ls *.{txt,xml,md} → That gets converted into this: → ls *.txt *.xml *.md
- The other wildcard you can use is ?, which matches any single character. For example, we have three files called filea.txt, fileb.txt and filec.txt, so we could list them all like this:
    - ls file?.txt
- these wildcards become much more valuable when you realize they are a feature of the terminal, not of ls. This means you can use them with cat, wc, head, and any other commands. For example, because we have three files called filea.txt, fileb.txt, and filec.txt, these commands do the same thing:
    - cat filea.txt fileb.txt filec.txt
    - cat file?.txt
    - cat file*.txt
    - cat file*

Options for listing

- The ls command has lots of options to modify the way file lists are shown
- You already learned that filenames starting with a period are considered to be hidden files. If you want to show those hidden files, use ls with the -a parameter, like this:
    - ls -a
- Two parameters you'll often see used together are -l and -h:
    - the former means "show a long listing" so that you see file size, permissions, ownership, and last modified date;
    - the latter means "show file sizes in a way humans can understand."
- Here's what the output from ls -l looks like by itself:
    - -rw-r--r--@ 1 twostraws staff 39 23 May 18:43 filea.txt
    - -rw-r--r--@ 1 twostraws staff 38 23 May 18:37 fileb.txt
    - -rw-r--r-- 1 twostraws staff 1001 24 May 22:59 filec.txt
    - here's what they all mean:
        - • -rw-r--r--@ are the permissions for this file.
        - • 1 means the number of hard links pointing to this file.
        - • twostraws is the user owner of the file.
        - • staff is the group owner of the file.
        - • 39 is the size of the file.
        - • 23 May 18:43 is the last modified time of the file.
        - • filec.txt is the file name being listed.
- Permissions in Unix are described as three sets of values: what can I do with a file, what can people like me (my group) do with a file, and what can everyone else do with a file? Groups matter a lot when you're working on a big system, e.g. where you might have a "student" group and a "teacher" group, but don't matter at all when you're working on a home computer.
- Let's break down one of the permission lines: -rw-r--r--@. The first - means it's a file rather than a directory (directories have d there), rw- means I can read and write the file but not execute it, r-- means people in my user group can read it but not write or execute it, the second r-- means people outside my user group can read it but not write or execute it, and @ means it has macOS extended attributes. macOS lets programs save attributes about files separately from the file itself, so a text editor might save the position you were at last time you opened a file, or Finder might attach labels.
- The number of hard links to a file is also a curious thing. Unix systems allow multiple filenames to point to the same file on disk. So, /Users/twostraws/hello.txt and /etc/hello.txt might be exactly the same file – one isn't an alias for another, they are both real files independently, so if you delete one the other doesn't break. At the same time, they are pointing to the same file, which means if you edit one, the other changes. For directories, the number of hard links will be 2 plus the number of links inside it.
- Now let's look at file sizes. My filec.txt has the size 1001, which means it uses 1001 bytes on disk. That's pretty easy to understand if you're looking at small files, but when

you see a size like 24721979 you need to read it carefully to understand that it means a 24MB file. This is where the -h parameter comes in...

- Using -h tells ls to use letters like B (bytes) and K (kilobytes) to describe file sizes, which makes for easier reading. You will also see M for megabytes and G for gigabytes.
- When you use ls with a directory, it will print the contents of that directory automatically. If you want it to work recursively – print the contents of the directory, print the contents of all subdirectories, print the contents of all sub-subdirectories, and so on – use the -R parameter.
- The most useful options to ls are those that sort the output. By default it's sorted alphabetically by name, but there are two other useful options:
    - -S (capital S) sorts files by size with the largest shown first,
    - -t (lowercase T) sorts files by when they were last modified with the newest shown first.
    - You can reverse the sorting by using the -r parameter. For example, this first command sorts output largest first, and the second sorts output smallest first, both with long listing so you can see the sorting has worked:You can reverse the sorting by using the -r parameter. For example, this first command sorts output largest first, and the second sorts output smallest first, both with long listing so you can see the sorting has worked:
        - ls -lS
        - ls -lrS

Piping one command into another

- The definition of “file” in Unix is broad, and in fact you’ll often hear long-term Unix users say that everything is a file. This means things like network sockets are considered files, devices are considered files, the input and output on your screen is a file, and more. Strictly speaking they are called “file descriptors”, but the reality is the same: whenever we have been working with file so far, we could be working with almost anything else.
- You have already seen how > and >> can write to and append to files. There’s another way of redirecting output, and it’s the pipe symbol, |. This might be accessed by pressing Shift+\ on your keyboard. This takes the output from one program and redirects it to another.
- ls -lS | head
    - Let’s break it down:
        - • ls:listfiles
        - • -l: with long listing format
        - • S: sorted largest first
        - • |: send output to...
        - • head: show first 10 items
            - Combined, that command will show the 10 largest files in the current directory.
- ls | wc -l
    - Let’s break that down:
        - • ls:listfiles
        - • |: send output to...
        - • wc:countwords
        - • -l: report only number of lines
            - Combined, that command will count the number of lines returned by ls, which is the number of files in the current directory.
- You can pipe together as many commands as you need, and even though I’ve only taught you a few commands you can already use them in interesting ways. For example:
    - ls -S | head -n 50 | less -N
    - list files sorted largest first, then use head to use only part of the output. This time I specified -n 50 to get the 50 largest files, but that’s quite hard to read in the terminal so I piped it again. This time it’s to less, and I used the -N parameter to number the lines of output.

Finding files based on search criteria

- Now you've seen how many options ls and less have, it should come as no surprise to you that the command for finding files is a real Swiss army knife of functionality. Again, I'm going to filter down the many options to the handful you'll find most important.
- To find files that match search criteria, you use the find command. This takes three parameters in its most common usage:
    - find somewhere -iname somefile.txt
        - The first parameter, somewhere, is where file should look for matches. You can specify . to mean the current directory if you want. Either way, find always operates recursively, so it will search the directory you specify, as well as all its children, grandchildren, and so on.
        - The second parameter, -iname, means "look for a file named..." and should be followed by the name to look for. The "i" in "iname" means "case-insensitive", which means it will find somefile.txt, Somefile.txt, SOMEFILE.txt, and so on. If you want case-sensitive searching for some reason you can use -name instead.
        - The third parameter, somefile.txt, is actually attached to the second parameter, because that's the name of the file we want find to look for. The find command can be used to search for things other than a file's name, so you normally specify things in pairs: "look for a name: somename" or "look for a size: somesize"
- Important: You can use wildcards with find, but you need to be careful. I already said that wildcards are a feature of the shell, not of the commands you run, which means these two commands do different things:
    - find . -iname *.txt
    - find . -iname "*.txt"
    - In the first command, wildcard expansion is performed by your terminal. In the second, wildcard expansion is performed by find.
        - We had the files filea.txt, fileb.txt, and filec.txt earlier, and when you run the first command they would be matched by the terminal. This means the first command is effectively this:
            - find . -iname filea.txt fileb.txt filec.txt
        - That means "find any files with the name filea.txt, fileb.txt, or filec.txt, in the current directory or any subdirectory." That might be what you want, but chance are you mean "find any file that ends in .txt" instead – in which case the second command is the one you want.
- You can search using other criteria, with a popular alternative being size. To do this, specify another pair of parameters: -size followed by a size option. This can be specified in a variety of ways, so here are some examples:
    - find . -size 10k
    - find . -size +10k
    - find . -size -10k

    - find . -size +1G
    - The first line looks for files that are exactly 10 kilobytes, the second for files that are greater than 10 kilobytes, the third for files smaller than 10 kilobytes, and the fourth for files that are greater than 1 gigabyte. Note that “k” is small, but “G” (and “M” for megabytes) are both capital – don’t worry, find will tell you if you screw that up.
- You can combine criteria together, but make sure you specify them in pairs. For example, this is correct:
    - find . -name "*.zip" -size +1G
    - Whereas this will not work:
    - find . -name -size "*.dmg" +1G
- You can of course pipe the output from find into other commands. For example, this will
- count the number of files over 100MB:
    - find . -size +100M | wc -l
    - As you can see, wc -l is a real workhorse and you’ll find yourself using it a lot!

Executing Output

- Where find starts to get interesting is its ability to execute a command on any files that match your criteria. This uses pretty unpleasant syntax, but it’s not hard to learn:
    - -exec
        - starts the command to execute on each file.
    - {}
        - (an open brace then a close brace) is replaced by the name of the matching file.
    - \;
        - (a backslash them a semi-colon) ends the command to execute.
- For example, this command finds all text files in the current directory or any subdirectory, and prints their contents out
    - `find . -iname "*.txt" -exec cat {} \;`
- You could use that create a combined file that contains all the other text files combined, like this:
    - `find . -iname "*.txt" -exec cat {} \; > output`
- Important: If you use this technique, make sure you don’t redirect to a file that matches the search criteria, otherwise you’ll get into a recursive loop until you run out of disk space!
- As an alternative, you can use -ok rather than -exec to have find ask you whether the command should be executed for each matching file:
    - `find . -iname "*.txt" -ok cat {} \; > output`

Searching for text with grep

- Now it’s time to meet one of the most powerful commands in the Unix toolkit, but for now I’m only going to use a small part of it.
- The command is called grep, and stands for “Globally search a Regular Expression and Print the results.” I’ll only be covering part of it here because we haven’t covered regular expressions yet, so we’ll return to grep later on.
- You’ve seen how find searches for files based on their metadata, such as their name or size. Well, grep is used to search for files that contain content you specify, which means it’s more CPU-intensive because it might need to scan hundred of gigabytes of content.
- Let’s start with something simple:
    - grep "posts" *
    - That will look through all files in the current directory, and find those that match the search string “posts”. In the output, you’ll see one line per match, and each line has both the filename and content of the matching line. This makes grep super-fast at finding something important, because you can see not only the name of the matching file, but also where the search text occurred in context.
- Some terminals enable colored matching by default, but if yours doesn’t then add the --color option to have your search text highlighted in the output:
    - grep --color "posts" *
- By default, grep searches only the current directory. If you want it to search recursively through subdirectories, the -r parameter. It also searches with case sensitivity by default; if you want it to ignore letter case, use the -i parameter. We can combine it all together like this:
    - grep -ri --color "posts" *
    - That will recursively search for the case-insensitive text “posts”, then print out results in color.
- You can change the match information that grep prints by using one of three parameters: -n adds the line number where each match was found, -l prints only the names of matching files, and -c prints the name of each file that was searched along with the number of matches in each file. It’s unlikely you will ever want to use more than one of those at once.
- There’s one more grep option you will find useful, which is -v. This inverts your search,
    - which means this command will return all lines that don’t contain the text “posts”:
        - grep -v "posts" *
- Just like our other commands, grep can be piped elsewhere, and it’s actually common to pipe grep back into itself. For example, this command finds all files that contain the text “posts”, then pipes that through another grep command to show only lines that don’t contain “the”, then counts the total number of matches:
    - grep "posts" * | grep -v "the" | wc -l

Copying, moving, and deleting files

- Unless you're increasingly obsessed with the terminal, chances are you'll only use these commands in combination with others, e.g. using the -exec option for find.
- To copy a file, use the cp command.
- If you want to move it instead, use mv.
- Note that Unix doesn't have a rename command for renaming, because that's identical to moving – use mv instead.
- Example:
    - cp filec.txt filed.txt
    - mv filed.txt filez.txt
        - The first command will copy filec.txt to filed.txt, leaving filec.txt intact.
        - The second command will move filed.txt to filez.txt, which means filed.txt will no longer exist.
- If you want to copy whole directories, you should use the -R parameter to cp to enable recursive copies for directories. For example, this command will recursively copy one directory to another:
    - cp -R somedirectory nameofcopy
- Finally, you can delete files using the rm command, like this:
    - rm filez.txt
- That doesn't work well for removing directories, and in practice you will see this following –VERY DANGEROUS– command used far more frequently:
    - rm -rf somedirectory
    - I've marked it as dangerous because I don't want you to type it by accident then send me angry emails: that command deletes whole directories immediately, including all subdirectories, and without any further prompts from you.

Reading File Information

- 2 simple commands that will make your terminal life easier:
  - which and file.
- The first of these tells you where a command is on your system, for example:
  - which ls
- That will probably print out “/bin/ls”, telling you where the ls command is. This is useful because commands are frequently in various places around the filesystem depending on what they do.
  - For example, commands stored in “/bin” are those critical to the running of the system, commands stored in “/usr/bin” are non-critical, and commands stored in “/usr/local/ bin” are those installed by the user and not the system. In practice all three are searched for commands you run, so using which can tell you exactly what will run.
- The other useful command for identifying files is just called file. If you pass this a filename as a parameter, it will tell you what kind of file it is.
  - For example file image.png will print out something like this:
    - PNG image data, 740 x 209, 8-bit/color RGB, non-interlaced
- Files can be recognized regardless of their file extension because of “magic numbers” – most file types start with a special sequence of characters that identify their format uniquely.

Combining commands

- You've already seen how > and >> can redirect output to write or add to a file, and how | can connect the output from one command to the input of another, but there are two more things I'd like to demonstrate so that your knowledge is complete.
- First, there's a command called tee that effectively combines > and | into one. It's called tee because it works like a T-junction: you come in on one path, but there's a turn to the left and a turn to the right. The magic of tee is that it allows you to take both turns: it writes your output to a file (like >) and also sends it to output so it can be piped somewhere else.
- To give you an idea of how tee works, try running this command:
    - ls -S | tee step1.txt | head -n 5 | tee step2.txt | cat -n | tee step3.txt
    - So, that single command will print the five largest files in the current directory, with numbers next to each line. But it will also save the results of each step into separate files, so you'll end up with step1.txt, step2.txt, and step3.txt that show how the data gets processed as each stage completes.
- The second interesting way to work with data is with backticks. These might be placed to the left of your 1 key or your Z key depending on your keyboard, but they look like this: ` – like single quotes that go from left to right.
- Backticks allow you to run a command and inject its result into your current command. For example, you've already seen the which command that tells you the location of a command – i.e., which ls will print "/bin/ls". If you only want to know the directory, "/bin". you would use the dirname command, which accepts a filename as its parameter and returns the directory that contains the file.
    - So, this will return "/bin":
        - dirname /bin/ls
    - If you want to combine the two operations together – find the location of the ls command but print only its directory - then you would use backticks, like this:
        - dirname `which ls`
    - Your terminal will evaluate the command inside backticks first, and will get back "/bin/ls". That output then gets placed into the original command where the backticks were, like this:
        - dirname /bin/ls
    - That then returns "/bin". You can even put backticks in backticks, but it gets messy quickly. As a result, it's common to see people use $(...) syntax, where you place your sub- commands inside the parentheses. For example:
        - dirname $(which ls)
    - Using this approach, we can now run commands like this:
        - ls $(dirname $(which ls)) | wc
        - That 1) finds the location of ls, 2) pulls out only the directory information, 3) writes a list of all commands in the same directory as ls, then 4) counts each command.

Sorting and de-duping

- Commands like ls have sorting built right in, but you can also sort any kind of output by using the sort command.
- If you use it with no parameters you get a general alphabetical sort, but you can also use the -n option to get a numeric sort, and/or the -r option to get a reverse sort.
- Sorting is particularly useful when used with the uniq command, which removes duplicate lines of data – but only if they are adjacent.
    - That is, you need to make sure your data is sorted before calling uniq otherwise duplicate data might remain.
    - uniq requires duplicate lines to be adjacent to one another. To fix this, we need to sort the file first
- If you don't mind learning another parameter, sort has a dedicated -u parameter that more or less mimics piping to uniq. However, I prefer to use uniq because it has its own set of options.
    - For example, uniq -i performs case-insensitive de-duping, uniq -c prints how often each item appeared in the input, and uniq -u prints only items that appear exactly once.
- Like many other commands, you can make sort draw on multiple files at a time by listing them all. When combined with uniq this means we can merge two files, sort them, then strip out duplicates, giving us the combined list of unique lines in each file:
    - sort file1 file2 | uniq
- Once you learn that sort can bring multiple files together, you should realize that if you bring the same file in more than once then its lines are guaranteed to appear in the output more than once. If you combine this with the -u parameter to uniq, which outputs lines that exist only once, then with a single command you can find all lines that appear in file2 but not in file1:
    - sort file1 file1 file2 | uniq -u
- With sort and uniq in your arsenal, you now have the ability to place them at the end of any of your existing commands to ensure there are no duplicates in there.

Terminal Tips and Tricks

<u>Record your Sessions</u>

- As good as your terminal history is, it only stores the commands you executed rather than their outputs.
- However, if you run the **script** command it will start up a terminal logger that will record everything you type as well as its output, meaning that you can create comprehensive logs for everything you do.
- To stop logging, run **exit**, then view your log using **less typescript**

<u>Write Text to a File</u>

- You can have the terminal print any text you want by using the echo command, like this:
    - echo "Hello, world"
- This might seem pointless, but remember that you can redirect the terminal’s output to files using >. With echo and redirect combined, you can write text straight to a file by providing some text and a filename:
    - echo "Write this text" > filename.txt
- When you open quote marks, you can place line breaks wherever you need them and they will form part of the text that gets echoed. When you close the quote marks, your command carries on as usual.
- When you press return after the first line, the terminal will show > for its command prompt as a signal that you can keep typing.

<u>Get to Grips with Pipes</u>

- Named pipes are a wondrous thing on Unix, but a lot of folks don't use them – and often don't even know about them. Put simply, when you run a command such as ls | head, you're piping the output from ls directly into head.
- What named pipes allow you to do is to create a file-like object that you can write data to, and when that data is read out by another process it will be cleared. These are called "FIFO pipes" because what is First In is First Out.
    - To try named pipes out, here are three examples:
    - the first creates a named pipe called "fifo_pipe",
    - the second opens the pipe for reading and will automatically print out any data that appears,
    - and the third (which you'll need to run in a different terminal window) sends some text to the pipe.
    - As you send the text, you'll see it appear immediately in the other terminal window.
        - mkfifo fifo_pipe
        - tail -f fifo_pipe
        - echo "Hello from the other side" >> fifo_pipe
- The advantage to named pipes is that any data that gets sent into them gets removed when its read, which makes it perfect for one-shot tasks or replacing temporary files.

Fix a bad terminal

- Outputting binary files to the command line can sometimes leave your whole terminal broken.
- To fix this, type the following and press return:
    - reset

Search your history for a specific command

- The history command will list all the commands you have executed recently.
- Combine that with grep and you can pull out a list of all commands that match a specific phrase.
    - history | grep "command"

Save that one for later

- Ever typed the perfect command, then realized you needed to do something else before you ran it? Rather than try to remember it all then re-type it later, go to the start of the line and put a # symbol at the front, then press return.
- The # symbol means that the whole command is treated as a comment and thus ignored, but it still gets entered into your history, so you can scroll back up through your history, remove the #, and run it normally.
- For example:
    - #grep "posts" * | grep -v "the" | wc -l

Repeat a command with root privileges

- If you run a command and get the error back that you don't have the correct privileges, don't re-type it.
- Instead, run this to have sudo repeat the command for you:
    - sudo !!

Time a program's execution

- Measuring how long a program takes to run is a cinch with the time command.
- Place it before any command you would otherwise run, like this:
    - time mycommand

Clear a file quickly

- This command is obvious when you think about it, but precious few people are aware of it.
- Put simply, if you want to clear the contents of a file without actually deleting it, use a right angle bracket before its name, like this:
    - > file.txt
- That works because it’s writing empty text to the file.

Create delays in pipes

- Unix command piping is actually quite complicated behind the scenes, and it's easy to get into race conditions because your commands aren't executed linearly from left to right. For example, a command like this one has a race condition that many people are blissfully unaware of:
    - cat somefile.txt | tee somefile.txt
- On paper, that prints the contents of somefile.txt to tee command, which prints it to the terminal and saves it back to somefile.txt. You would never use this command in practice – it's there for demonstration purposes only.
- In practice, that command will work just fine maybe 90% of the time. The other 10% of the time, it will wipe the contents of somefile.txt, and output nothing. This is called a race condition: both cat and tee are racing to execute as fast as they can, and if tee begins execution first (which can happen) it will delete the contents of somefile.txt ready for its new input. Then, when cat runs, it will find no input to read, so nothing will happen.
- I've already said that redirecting output to the same file you're reading from is tricky unless you're careful, but fortunately there's a simple solution that is perfect for this use case: tell the system to insert a short "sleep" pause between the cat and the tee.
- To make this work, we're going to use a new command called sleep and ask it to wait for 0.1 seconds, but we also need to combine the sleep and tee calls into a single operation. This is is done using parentheses around the whole command you want to run, with semicolons between individual statements. So, the full command is this:
    - cat somefile.txt | (sleep 0.1; tee somefile.txt)
- By adding that tiny 0.1 sleep after the pipe, that command will run correctly.

Clear the sudo cache

- After running sudo and entering your password, it will be cached for a period of time so that you don't have to keep on entering it each time you want to run another command.
- This is helpful for most people, but it's also a security risk: anyone with access to your keyboard can use sudo to cause untold destruction to your system.
- The fix? Clear the sudo password cache like this:
    - sudo -K

Share files the no-hassle way

- Python comes with a built-in HTTP server designed to serve up all the files in whatever directory you were in when you started Python. This means you can share everything in a directory at the address http://127.0.0.1:8000 (replace 127.0.0.1 with your external IP address) simply by running this command:
    - python -m SimpleHTTPServer
- macOS comes with Python built in, as do most Linux distributions.

<u>Find files modified today</u>

- That file you saved a few minutes ago – how did you manage to lose it? Oh well – fortunately
- you can find it with one simple command that shows any file modified in the last 24 hours:
    - find . -type f -mtime 0
- The -type f parameter pair mean “look for files only” and `-mtime 0” means “modified right now.”

<u>Made a typo? Fix it quickly</u>

- If you make a typo that causes a command to fail, you can quickly re-run it with the typo correct by using the caret symbol – that’s ^ if you weren’t aware.
- For example, let’s say I want to list all files that have a name starting with “file”, so this happens:
    - virgil:Desktop twostraws$ ls filf*
- Oops! As you can see, I wrote “filf” rather than “file”, so nothing was found. To fix this, type a caret symbol, then your typo, then another caret simple, and the text you want to replace it with. For example, to fix the mistake above I would type ^filf^file. When you do this, your terminal will replace the text in the command, show you how it looks now, then run it, like this:
    - virgil:Desktop twostraws$ ^filf^file

Remote terminals

- Now that you’re comfortable using the terminal, you’ll be pleased to know that everything you’ve learned can be applied on servers. This means if you run a website, for example, you’re now able to work on the server just as you are on your local computer.
- Remote connections are made using a program called SSH, which is short for “Secure Shell.” It connects to a remote computer – which could be a Linux server or a macOS desktop – and gives you a terminal where you can run commands. All your commands are encrypted, so it’s completely safe to control important servers this way.
- Most Linux servers already have SSH installed and activated, largely because that’s how cloud systems are administered. If you’re running desktop Linux it might already be installed and activated; if not, you’ll need to install the OpenSSH server – something like this ought to do it:
  - sudo apt-get install openssh-server
- Please consult your distribution for specific installation and configuration instructions. If you’re on macOS, go to System Preferences > Sharing, then enable Remote Login.
- Once SSH is installed, you can connect to it from any other computer. To do that, run the ssh command like this:
  - ssh remoteusername@yoursite.com
- You need to replace “remoteusername” with whatever account username you have on the remote machine. If the remote username is the same as your local username you can just write this:
  - ssh yoursite.com
- By default, SSH connects on port 22, but it’s not uncommon for a different port to be used – it’s a bit like security through obscurity. For example, if you want to connect to port 2020 instead, use this:
  - ssh yoursite.com -p 2020
- Regardless of how you connect, you’ll be asked whether you want to trust the remote server. This happens the first time you connect because the remote machine is unknown, but when you choose to trust it you shouldn’t see this again. In fact, if you do see the message again you should be careful because someone might have compromised either the server or your connection to it.
- Once you trust the remote machine, you will be asked to enter the password for your remote username. Finally, you’ll be logged in, and it works just like the terminal you’ve been using so far.
- To close your SSH connection, press Ctrl+D.

Other uses for SSH

- As well as giving you an interactive terminal to another machine, SSH can also be used to copy files using its scp command. This line below, for example, connects to another server and uploads the file data.zip to the remote directory /home/twostraws/Documents
    - scp data.zip you@yoursite.com:/home/twostraws/Documents
- So, you write your remote username, then the host name, then a colon, then the path where you want to copy the file.
- As good as scp is, it's just not efficient when you have lots files to copy. For that purpose, you need sftp to provide you with an interactive FTP-like transfer pipe, through which you can upload and download as much as you want. Fire it up like this:
    - sftp you@yoursite.com
- Once that's done, you can upload and download files using commands like put data.zip and get data.zip.
- SSH is also capable of acting like a tunnel. If, for example, you were entirely hypothetically thinking of connecting to a cloud server so that you could, gosh, I don't know, maybe get around Netflix's geolocation that stops you from watching the TV you want... well then SSH is exactly what you're looking for.
- For example, I have a server in Austin, Texas, and if I wanted to watch US Netflix – entirely hypothetically, you understand – I could set up a tunnel from my computer to the US-based server using a command like this:
    - ssh -vND 8887 me@mysite.com
- The -v parameter enables verbose mode so I can see what's going on, and the ND parameters set up port forwarding. What it does is take any data to come through on the local port 8887, then transparently forward it on to the server in the US. This means I could (IN THEORY AND TOTALLY NOT IN PRACTICE) tell my computer to use a SOCKS proxy at the address localhost on port 8887, and it would send me packets to the US server so I could (but absolutely don't, no sir) watch US Netflix. For no other reason than I'm sure you're curious, that setting is under System Preferences > Network > Advanced > Proxies on macOS.

Terminal multiplexing

- let’s talk about the program screen. This lets you run multiple virtual terminals inside one terminal. Now, if you’re working locally and have a graphical interface, this is really a non-issue: you can probably open more tabs, or at the very least open new windows, so there’s no need for screen. But if you’re working remotely, then you don’t want to have four different SSH connections open to the same server. Instead, you can connect once, launch screen, then have four virtual screen terminals all running over the same connection.
- To get started, run the screen command by itself. You’ll see a short welcome message, and a prompt to press space to continue. When you do that, you’ll be back at your shell again – but this time you’re actually inside screen. Type ls to see the files in the current directory; it doesn’t matter what it says, this is just to load up the terminal with some example output.
- Now, the keys to control screen are special, because they are designed to avoid conflicting with any programs that might be running inside it. All commands start by you pressing Ctrl+a, which enters command mode. The next key you press is the actual screen command you want to run. For example, pressing “c” will create a new virtual terminal. So, you press Ctrl+a together then release and press “c”. This is usually written as “Ctrl+a c”.
- Try it now, and you’ll see your terminal clears – or at least appears to. The original terminal is still there, but we created a second one. Run ls -l in this second terminal to fill it with some example output so we can identify it. To go back to the first terminal, press Ctrl+a 0 – i.e., press Ctrl and a together, then release and press 0. You’ll see the original ls output there, untouched. Now try Ctrl+a 1 to go to the ls -l terminal again.
- You can have ten terminals numbered this way if you want to, but alternative you can press Ctrl+a " (a double quote mark) to choose between the open shells graphically. When the program inside a virtual terminal finishes – i.e., if you quit the terminal by pressing Ctrl+D – that virtual terminal automatically closes and you’ll be moved to an adjacent one. If there are no more virtual terminals, screen quits.

What makes screen clever

- So far you're probably thinking that screen is neat, but more bother than it's worth. But wait: there's more! First, screen has the ability to lock itself so that the terminal windows are hidden from others. This is separate to locking your actual computer, so it's possible to have your computer unlocked but your screen sessions locked. To do this, press Ctrl+a x. You'll be asked to enter a password two times, but after that the screen will lock and you'll need that password to unlock. In the future, using Ctrl+a x will lock screen immediately.
- Where screen gets really clever is that it allows you to disconnect from it then reconnect later – and pick up where you left off. So if you SSH into a server and have a busy screen session with lots of things running, so you can leave that running, quit SSH, then reconnect later on and continue all the previous operations.
- To disconnect from your current session (while leaving everything running) use Ctrl+a d. You'll be dropped back to the terminal where you launched screen from, and you'll see the message "[detached]" printed so you know it's worked. You can now close your terminal window or disconnect from SSH – it doesn't matter. When you want to reconnect, run screen -r to pick up where you left off – easy!