A N00b's Journey through the Linux Terminal

Uriel K Alistair

Nov 2022

Contents

Preface

I started this document just as I downloaded Ubuntu without the first clue as to how to type a command. I was actually quite disappointed to find out that everything wasn't done on a GUI. "Why would anyone condemn themselves to this?" I asked.

Now I know. It is the power of the sun in the palm of your hands. This is the story of how I found out. Well, more or less. I do not, by any means, intend for this to be a comprehensive guide.

If you're entirely new to Linux, I'd recommend you read the first 3 chapters, do the first ten levels of Bandit (Over the Wire) and then continue.

If you're already accustomed, it's not unlikely that there are still quite a few things you can pick up from here.

Chapter V: Process and Shell Management and Redirections in Chapter III should have a few things you may find helpful.

The section on ls in Chapter I is worth a quick read.

The Bash Scripting chapter is worth a read if you aren't already very familiar.

Knowing Regular expressions definitely cannot hurt, so if you aren't on good terms with them, a short summary is in Chapter IV, but you must use these commands often to learn them properly. If possible, I'll attach a problem sheet covering this approximately chapter by chapter as a separate file somewhere near where you found this doc.

If you're interested, check out Section 2: Oldschool content. Awk and sed are both very powerful languages that can be learnt each under an hour.

You can probably do without the bonus content. Read if you're interested.

I hope you find this useful. If you can stride forth on the terminal a little more confidently, wielding commands with a little less uncertainty, and getting things done with a little less pain, then at the end of the day, I've shared with you my journey and my destination. There's nothing more I can ask for.

Section 1 : Basic Content

Chapter I: Knowing About Commands

Ctrl L is clear screen (command : clear)

Ctrl D is exit shell. (command : exit)

Manuals

man, whatis

man commandname

This gives the manual page for the command. Most of these are easy to read.

To get just a brief description of a command, use

whatis commandname

This just gives you the first line of the manpage, and is often clear and concise.

(I suppose that's uhh… kinda redundant when you can just use the man command but fine.)

info

This gives a very large manual about a given command.

Although man is typically the first thing you loop up, info is far more comprehensive, and typically houses information you cannot find in man.

It is much less cryptic in its language than man in my opinion, and has a really cool headings and sections navigation system.

which (+whereis)

which commandname

which gives the path of the file which has the instructions for the command being executed.

whereis commandname

This will locate the binary, source instructions as well as the manpages.

which cd gives a null output. This is because there is no place where it is stored. It is inbuilt in the shell. But that doesn't mean all shell builtins return a null output. Some like "test" actually do have a location where their code is stored. Cd is too close to the shell's heart, I suppose.

type

Type tells you what type of a command a given command is.

Examples:

1) type cd

This tells you that it's inbuilt.

2) For aliases, it will tell what the command is aliased to.

3) For other commands, it will say that it is hashed to <source file location>

"hashed" means something like attached to and memorised to be here.

type type says type is builtin :)

Notice: man type will let you know that there is no manpage for type.

manpages do not exist for builtins like type and cd.

help

This is the manpage equivalent for shell builtins. It doesn't look nearly as pretty though. It just dumps out the info on the terminal but it's typically quite short.

Since it dumps on the prompt, I'd recommend piping to less. This should be a general practice, really, so I'll keep saying this once in a while to drive it in when it's read.

apropos

apropos keyword

apropos is a somewhat helpful tool to discover new commands. It searches the manpages for the keyword you ask, and gets you some potentially relevant functions.

This is equivalent to man -k keyword

Chapter II: Viewing and Navigating Files and Folders

ls (permissions,inodes and links)

List stuff. -l will give you a bunch of details.

Example:

```
drwxr-xr-x 2 uka uka 4096 May  7 19:59 Desktop
```

The very first character is the type of file. Here, we have ourselves a folder, so it starts with a d, which stands for directory.

Regular files start with a hyphen. Do not confuse it with file permissions.

There are other things this first character could be, but I'm not all too familiar with them, and won't bother with them for now. (Symbolic Link gets l, Character file c, Block file b, Socket file s, Named pipe p)

Next up there are owner permissions, group permissions and others permissions.

r is read, w write, x execute. Here, It looks like I have rwx permissions, my group as well as other people can read the files on Desktop,and execute them, but cannot write anything to them.

Typically the permissions are given as 3 numbers rather than 9 alphabets. Pretend we have a 3 digit binary number. r w and x correspond to each bit. 110 is rw but no x. 011 is w and x. The decimal value for 110 is 6. Doing this to each of the 3 in my Desktop details, we get a permission code of 755.

Now if I want to change the permissions listed here, I can use the chmod command.

```
chmod g-x Desktop
```

Right after chmod, we have to say whose permissions we are modifying. g is group, u user. Minus is remove permission, plus is give permission. So here, I am removing execution permissions from my group.

Compactly, using the code I mentioned, we can do chmod 700 Desktop

If you don't have execute permission for a folder, you cannot even enter that folder.

Okay, now that I have understood the permissions, let's look at the other things in the ls -l output.

The last one is clearly the file name. The one before that is the last modified timestamp. Before that, we have the file size. This is in bytes by default, so to make it human readable, use the -h option when calling ls -l. For files, this shows an accurate file size. For directories however, it's typically 4096. In our imagination, a directory is a giant box inside which things exist. On physical storage, there are no boxes. There's just ones and zeroes.

In fact, one single file may exist as multiple fragments at various memory locations. The only thing that a directory will contain is a table that says I have files a, b, c… they start at memory address x and go on for so many *blocks*. That is, a directory is more like a map that guides the system to your files rather than a box that contains them.

In linux, every file is assigned a number and a name. The number is called the inode number, and is unique for every file and folder. The inode contains all the "metadata". (data about data)

By default, the inode file of a directory is allocated 4096 bytes. When you put enough files in your folders, this size will grow. To view the inode number of a file, use the -i option with ls.

The -a option will show all files, including hidden files that start with a dot. The convenient thing about ls options is that they can be listed anywhere and in whatever order you like. ls -lia is the same as ls -ail etc. you can even do ls -i foldername -l , But just don't. Keep the options neat and together. Most commands allow only this neater format.

Back to the long listing.

The two uka s are the owner name followed by the group name.

That 2 in the middle is the number of hard links that point to this file. It's uhh… kinda like a shortcut. See, there are 2 kinds of links in Linux, soft or symbolic links and hard links.

A soft link is a shortcut that contains only a *path* to the target file. When the target file is deleted or *even moved*, the link just stays dangling and is now useless. This is exactly how windows shortcuts work.

On a tangent, how do you think files are deleted? If you've ever deleted a 2 Gigabyte movie, you'd know it happens in the blink of an eye. The thing is, hunting down every place where a piece of a file is stored and overwriting it with zeroes is a waste of effort. What could you possibly accomplish by that? Instead, we just *label* the places on the hard drive as free for use, and the next time you need to store a new file, we just write it on top of the old file, replacing the 1s and 0s.

That is, the delete button doesn't really *delete* a file. It just asks people not to look that way.

Now, back to links. A hard link has the same inode value as the target. The inode contains a pointer to a location in a hard drive; It's got nothing to do with the file address path.

Since the hard link and the original file both point to the same inode file, and therefore the same data, you can delete the original file, and nothing will be lost. The hard link is still there. You can move the file around as much as you like, the change of folder addresses and accounting business inside directory tables doesn't affect the linking, because we ensure that all hard links point to the same inode.

Essentially, both the target and the hard link are on the same level. They are like 2 names of the same person.

So how does linux know when a file's space has been deleted and is free?

It maintains a count of the number of hard links, remember? We could see it. Once we delete all of them, the file is gone.

(Another Side Note: Directory tables are just your folder organisation system. You moving files here and there is just changing the entries of the directory tables. You don't *need* to wipe and move files to move them from here and there.

Moving files takes time the more files you have, because you have to update a *lot* of metadata. You need to specify which blocks each file is in. This will be much worse if your file is spread out due to defragmentation.

If you move a single 2 gig movie file, it's probably gonna move instantly. Of course, if you want to move a file between partitions, or between distinct storage devices it will take a lot of time.)

Example:

```
ls $(which apropos) -li

1835056 lrwxrwxrwx 1 root root 6 May  5 18:17 /usr/bin/apropos -> whatis

ls $(which whatis) -li

1836309 -rwxr-xr-x 1 root root 48416 Mar 18 00:33 /usr/bin/whatis
```

You can see that apropos points to whatis, and is a symbolic link, as indicated by the leading l in the long listing. The inode numbers are distinct. If this were a hard link, the leading character would've been a hyphen, and the inode numbers would be the same.

Although apropos and whatis are different commands, they point at the same file. The file looks at what name it was called with, and gives an appropriate output.

We can make these links ourselves, using the ln command.

If you look around a while, you'll find that the hard link counts for folders seem a bit… weird.

This is because Linux uses a very convenient system of hard links in every directory: All directories have a hard link "." that points at itself, and a hard link ".." which points at the parent directory, inside them. You can see these when you display hidden files in any directory.

So, when I make a folder test1, it will have 2 hard links, one from the straightforward file name, which every file gets, and one from the dot inside it. Now, if I create a subfolder in test1, that subfolder will have a ".." file that points to its parent, test1. Now the Number of hard links will be 3. At the time I first wrote that ls -l line, I did not have any folders on my Desktop, and the number was just 2.

An interesting note: The root folder of linux is its own parent folder. You'll find that . and .. in root have the same inode, no 2. That aside, I found something weird: two of the folders in root had the same inode no despite being distinct folders. Apparently, inode *doesn't* uniquely identify a file: device name and inode *together* does. That seems a bit silly but fine. I don't think we typically run into such a case. If we do, use stat filename to get the device's name.

That's about it. That was quite a lot to unpack. Phew.

Update: Apparently, even after all this, I missed 2 important details:

1) The ls command can take a *path* as an argument, and give you the things in that path. In fact it can take multiple path arguments separated by spaces.

2) There exists an option -R to recursively list the contents of everything ls detects in subfolders and their subfolders etc. This is bound to flood your terminal, so I'd recommend redirecting the output to a file. (or piping to less)

cd

Change Directory. use cd address

. points to the current directory.

.. points to the parent directory

~ denotes home directory

Note that these 3 are really useful in any command that uses filenames, not just cd.

cd - means go to the directory we were previously in.

This is particularly useful if you want to define a sequence of actions (an alias perhaps) that involves changing into some other directory to do something. We can use this to come back to where the user was when they invoked the command (assuming you cd'd only once).

When you want to open an adjoining file or folder whose name contains hyphens, you need to use a ./ address instead of just the name.

To denote a space, "\ " is used. This ensures that a space is not interpreted as a field separator. In fact, in general, when you wish to type characters into the shell that will get interpreted you use \ to *escape* that interpretation.

Typically you can Tab fill and avoid the pain of putting in these \s. If that's not an option for whatever reason, please just write the name in quotation marks. The name will be treated as one block, and not many separated by spaces.

cat

Typically, it’s used to print the contents of a file to stdout.

Using simple redirecting, this lets us copy files (>) or append one file’s data to another file (>>).

As a tool for display, cat is messy with big files and clouds the terminal. Even after the terminal is filled, it keeps writing, and you cannot see the start of the file, or any of your previous commands.

A better choice for displaying text files, would be less.

Trivia: tac file.txt prints the file in reverse. That is, lines go from bottom to top, but contents of lines are unchanged.

less

less accesses a file page by page, and lets you scroll through it. You can then quit from the file using q and nothing will be present on your terminal. Since it accesses the file page by page, it is fast to initiate, and does not have to load the entire file in one go.

Note that when you wish to chain commands that involve the contents of the file, you want to use cat, which dumps contents to stdout which can be piped elsewhere. Less is meant for viewing, and viewing alone, without flooding your terminal.

Side Note:

whatis less gives us : “The opposite of more”

man less makes you wonder if someone was giggling as they started typing the manual page.

DESCRIPTION

 Less is a program similar to more(1), but it has many more features.

“Less is More.” - Ancient Linux Wisdom

wc

Word count.

It will give you the line, word and byte counts for a file.

To get just no of lines use the -l option, -w for words etc. Just check the manpage for wc, it’s really short.

Personally, I find that wc -l, especially with command redirection is the most common usecase for this.

file

Gives details about what type of file the argument is.

Just try it out, you’ll get what I mean.

file * will list the file types of all the files in the pwd.

Misc (pwd,uname,du,stat)

pwd : print working directory.

uname : gives the name of the operating system.

du : Gives you disk usage statistics of files and folders. du -sh dirname lets you see a directory’s file size, unlike ls which will give you only the uninformative link size.

stat will also give you file size and number of blocks on the hard drive, and all timestamps, apart from some details I can’t understand.

Chapter III: Creating and Modifying Files and Folders

touch

This is actually meant to change the last modified and last accessed timestamps of a file or folder. For instance, ls -l to see your Desktop's last timestamp. You can touch Desktop to update it.

In the absence of a file touch just creates one, so it's a convenient way to make empty files.

mkdir

Make Directory.

To specify the permissions, use -m=700. The -m option stands for mode. (chmod stands for change mode.) (700 is just an example mode, use whatever you want)

Chmod

Heading is included for completeness. Check in the permissions paragraph here.

Ln, readlink

Make symbolic and hard links.

```
ln existingfile newfile
```

Makes a hard link to the existing file.

Using the -s option lets you make symbolic links.

On the topic of links, the readlink command tells you the ultimate target of a soft link.

For instance, if you call the pico editor from the command line,

```
which pico
/usr/bin/pico
ls -l /usr/bin/pico
```

```
lrwxrwxrwx 1 root root 22 May 18  2022 /usr/bin/pico -> /etc/alternatives/pico
ls -l /etc/alternatives/pico
lrwxrwxrwx 1 root root 9 May 18  2022 /etc/alternatives/pico -> /bin/nano
ls -l /bin/nano
-rwxr-xr-x 1 root root 320136 Apr 10  2020 /bin/nano
```

That is, the soft links bounce it around, eventually, to the nano editor.

readlink -f pico will let you know the final endpoint directly.

cp

Copy. You can make a copy of a file (or a directory).

```
cp file1 file2
```

File 2 is created and has all the stuff 1 does.

If file 2 already exists, it will be overwritten. Having the -i option asks the user for a prompt confirming if you want to overwrite. That seems like a useful thing, but if you *want* to overwrite a *lot* of stuff it becomes a pain. If you're a scaredy cat, feel free to alias cp to cp-i and rm to rm –i.

```
cp file1 dirpath
```

Dirname can be passed instead of path if the target directory is in the same directory as the file being copied. In any case, the copy will be put inside the target directory and will have the same name as the original file. To give it a custom name, just add a / after the dirpath and give your filename.

The first thought that popped into my head was: What if there is a folder with that name there and you want to make a file with the same name? Won't it go inside that folder by default?

It's just not allowed in Linux. You can never have two objects with the same name in a directory, regardless of the file type. Apparently "Everything is a file" in Linux. I can't quite appreciate the nuance of that statement, and reading about it on stackexchange gives me a close approximation to a migraine, free of cost.

To copy a directory we need to specify the -r option, which stands for recursion. This recursively copies every file inside a directory to the destination. Recursion is assumed in some other commands, such as mv.

mv

Move files and Directories.

mv file 2 ..

That will move file 2 to the parent directory. You need to give an address as the second argument, but yeah, use shortcuts if possible. ../../blah/blah is a valid address. The relative paths are OP.

In general mv source destination works. Source can be file or folder, no issues. (unlike cp)

mv can move multiple files at once. * is used as a wildcard character as a placeholder for any number of characters. So, for example, mv ./*.txt dir_1 will move all the text files in the current directory to dir_1.

mv can also be used to rename files (and folders).

mv file2 file3

Will change the name of file2 to file 3. When you think about it, that's really cool but also obvious :)

rm

Remove. Not Rap monster.

It will delete a file, or a folder on demand. Simple. To delete a directory, you need to specify the -r option for recursion. If you want prompting, keep the -i option.

Sometimes, files will ask you for confirmation anyway because they are protected or whatever.

This gets annoying if you wanna delete a lot of them, so use the -f option for forced delete.

Linux assumes the user knows what they are doing. No nonsense. If you hit enter, you can kiss the folder goodbye. The enter button is a bit scary, isn't it ? xD

"When I found you... I saw raw, untamed power."

Redirecting into Files + Piping

Commands normally take input from "stdin", and output to "stdout", which any error messages sent to "stderr".

Normally, stdin, is fed by your keyboard, stdout and stderr, both point to the terminal screen, so that you can see both.

We can move any of these 3 pointers to wherever we like.

1) To move stdout to a file, use >

Eg: ls -hails > file1.txt

Note that this will destroy any preexisting file of the same name unless it doesn't have permissions to do so. Once destroyed, it will write whatever output it receives, even if it is just null output.

The preexisting file *will be gone*.

2) If you just want to append output to a file, use >>

Eg: ls -hails >> file1.txt

3) To move the stderr to a file, use 2> (Naturally, to append, it's 2>>)

Eg: ls -hails ./blah >> file1.txt 2> file2.txt

(stdout goes to file1.txt, that is, any valid output, and any errors, go to file2.txt)

(./blah is a path argument to ls. I have used a directory named blah which doesn't exist to throw an error.)

I suppose it is sort of obvious, but I'll state it anyway– if you only redirect stdout, errors will still come only to the screen, and if you redirect only stderr, stdout still comes to the screen.

If you want to write both errors and output to the same file, you'll can do something a bit roundabout - redirect stderr to stdout, using 2> &1, (&1 is the first pointer in the whole system, to stdout) and then redirect stdout to a file.

This is because if you use separate 2> and > options, they will each *destroy* the pre-existing file when they show up. The straightforward way of doing it is simply using

3.1) &>

Which will redirect both stdout and stderr to the target.

Note: Sometimes, you know your code works very well, but it throws a ton of warnings which flood your terminal. You'd love to get rid of all them and keep things neat and tidy. Instead of redirecting to a file and accumulating worthless data, you can redirect these errors into a black-hole-garbage-can called /dev/null. Typically, you *do not* enter the /dev folder, because it has a lot of sensitive files, but this time it's okay. /dev/null is periodically incinerated, and is used as a rubbish bin by all of linux. So, just use 2> /dev/null if you are confident those errors are useless.

(Note: some people like to use >&- 2>&- to "close" stdout and std err respectively. However, this is very different from redirection, and may cause your program to respond in strange ways, since it is prevented from outputting, rather than dumping the output in the trash can. Kind of like how you'll get a W grade if you don't come to class, but you can totally get away with typing this document when the prof goes on rambling about band reject filters.)

Practical example: alias py="pathtopycharmhere &>/dev/null &! disown"

The &! will send the process to the background and disown will detach the process from the terminal, meaning it will not shut down even if you close the terminal.

(As a side note, I don't use that alias for pycharm, I made it into a function, so that it can optionally take a file as an argument and open that file in py. Aliases don't work with arguments.)

4) To redirect stdin into a command , use <

Eg: wc < file1.txt

This may seem a bit weird, but it could totally be useful. You may want to store the outputs of multiple commands onto a file and then send it as an input to another command, and this can help you do exactly that. For a simpler way of chaining commands:

5) Use |

This sends the output from one program as input to another program. This is accomplished by making the second program expect input from stdout of program 1, where the output is dumped. This thing is called piping.

Example: ls | head -3

The output of ls is passed into head, which selects out only the first 3 entries.

Side Note: head prints the top x, tail prints the last x. Without an option giving the no of entries, it gives 10 values. These two commands are pretty useful when you want to look at the top few entries in a gigantic list.

Side Note 2: if you expect a flood of output, please don't ruin your terminal. Pipe the output to less. It is *so* much cleaner. Yes, I won't shut up about it.

Side Note 3: If you want to show stdout on the screen but you also want a copy in a file, you can pipe (command | tee file1.txt) into the tee command. (Think T joint in water pipelines) The tee command will split the output into two and send one to the files it has as arguments, and the other to stdout. If you give more than one file as argument to tee, it will fill all of them with the same data from stdin. This can be useful if you want multiple copies to be destructively utilised for your purposes.

Wanting to see the output isn't the only use case: If you want to chain the output of the command too, you need a copy sent to stdout. Using tee you can do further piping as well as store to files.

Chapter IV: Some Text operations

find

Very powerful command that lets you find files that satisfy specific conditions.

-size option lets you specify the file size.

-size 1033c means exactly 1033 bytes. Without the c it is taken as kilobytes

-size +1000c means more than 1000 bytes. -1000c means less than.

-executable gets only executable files.

Prefixing an option's hyphen with !, (like ! –option) allows us to use a not true filter.

-user option allows us to specify the user who owns the file.

-group: same as -user but for group.

-ctime +5 means file was changed more than 5 days ago. -5 means less than 5 days.

-atime ±n => file accessed n more/less than n days ago

-regex Lets you use Regular expressions to search filenames

-name is a crude version of searching through file names, not as sophisticated as regex. * is any number of characters, and ? is one wildcard character.

-print prints the full path name of the matching files. Maybe useful for piping.

-exec lets you execute a command on the files that are selected, with {} as the placeholder for the filename.

For instance, find . -size +100M -exec ls -lsh {} \;

will give a long listing of files beyond 100 MB. I needed to escape the semicolon because I want to be a part of the argument of exec rather than a termination of the find command as interpreted by the shell.

grep

grep is a very powerful pattern detection command.

grep pattern filename

Is the syntax.

Alternatively, use can just not give a file argument, thereby making it expect input from stdin. This way you can pipe things into grep.

The patterns are written in something called "regular expressions" which are of quite some relevance to the string manipulations offered in most programming languages. (grep stands for global regular expression print, but who cares)

grep 'Anu' names.txt

Searches for the word Anu, (case sensitive) through names.txt, it may be in the start or the middle of a line; it does not matter.

In order to do a case insensitive search, use the -i option.

It is always a good practice to enclose the pattern in single quotes in order to escape any alternate interpretation of the symbols we will use (like $) by the shell.

An important notice: the -v option lets you get all the entries that *don't* match the pattern.

1) Dot represents a wild card, for exactly one character. ‘s.n’ matches with sun,sin,son etc.

2) ‘*’ means a character or pattern occurs zero or more times. Thus, ‘.*’ is a wildcard for any number of characters. ‘f.*k’ matches ‘flock’ , ‘flask’, ‘freak’, ‘fickle’ etc.

3) A dollar nails down the end of the pattern to be at the end of the line. ‘.am’ matches with raman and manickam but ‘.am$’ matches only manickam.

4) Similarly, a carat nails down the start of the pattern. Using both a carat and a dollar dictates the entire pattern.

5) Notice: This is purely at EOL. We may want our matches to consider the end of *words*, for instance, ‘an$’ won't match Raman Singh, but Matches Vel Shankaran, since the ending an of Raman is in the middle of the line. ‘.an/b’ implies it detects a “word boundary”

6) Square brackets denote a kind of wild card - matching only to a small set of specified characters. [aeiou] will match with any vowel, [1-9] will match with the digits 1 to 9 (both included) [A-M] with capital letters between those, and so on. ‘EP21B0[0-4][0-9]’ Will match EP21B000 to EP21B049. Note how I had two use two brackets, not one with [00-49]; That would be nonsense; This is *one* character.

Further, there are categories of characters you can refer to using double square brackets.

[[:aplha:]], [[:alnum:]], [[:upper:]] are some examples. Using a carat inside the first square bracket negates it.

7) [^5-7] A carat denotes a *negation* when inside square brackets. That is, the character should *not* be 5 or 6 or 7.

8) The number of times a character, or an entire pattern, must occur, can be controlled by use of this very lateky syntax : \{number\}

Eg: ‘M\{1,2\}’ means one or two capital Ms is okay.

9) Parts of patterns can be grouped by \(pattern\) This is essential in more complex queries

These can be referred to again using \1, \2 etc for 1st grouping, 2nd grouping etc.

However, there are some nuances to worry about:

'\(.a\).*\1' matches Umair Ahmad because ma, a pattern matched to the start, is followed by a bunch of random stuff, then finally ends with the *same* pattern we saw at the first, ma, *not* a new wildcard.

However, '\(.a\)\{3\}' will match with Sagayam, each dot being a new wildcard.

Extended grep allows us more tools for pattern matching. This can be activated using the -E option of grep, or by using the command egrep in place of grep.

1) '+' means a character or pattern occurs once or more.

2) '|' can be used as an OR

3) Those annoying backslashes can be skipped when making groups or telling no of repetitions

Knowing regex is important. I am familiar with the basic stuff, but the lateky stuff trips me up real bad. Regardless, knowing more regex is *only* a good thing. Just to remind you, I'll keep hyperlinking to this. Just you wait.

cut

cut is a very useful command, especially when chained with grep

It cuts out a part of every line in a document.

cut -c -8 text.txt gives the first 8 characters, 5- the characters from 5 onwards, 7-10 gives 7th to 10th, both included and so on. Numbering starts at 1. Note the existence of the -c option for characters.

cut can grant you the fields present in delimited text, when you use the -d option.

cut -d "," -f 2

Splits every line with commas separating the fields, then gives only the 2nd field as output.

uniq

uniq data.txt

uniq detects adjacent lines that are identical.

-c option shows the number of occurrences of these duplicates.

-d prints only duplicated lines.

If for some reason, you want to know how many unique lines exist in a file, we will first need to sort the strings so that all the duplicates coalesce together, since 2 identical lines are far apart cannot be detected by uniq.

sort data.txt will do this job. Piping the output to uniq completes our task.

tr

It supposedly stands for transform. Though transliteration is a more useful way of thinking about it. It will map a chosen set of characters to another set of characters.

tr "[a-z]" "[A-Z]" for instance, converts lowercase letters to uppercase ones.

the brackets can be dropped, and multiple series can be added like

tr "a-zA-Z" "n-za-mN-ZA-M"

You can use the -d option and give only one set to delete the specified set of characters from the input.

Chapter V: Process and Shell Management

ps

This is the ls of processes. It's fairly straightforward.

Every process has an ID attached to it, simply called its "PID". You can find this listed in the output.

The -f option will tell you the PID of the process that *spawned* this process. To get a somewhat nice* visual feel for this, we can use the --forest option.

The -e option lists *legit* all processes. Including random OS crap that typically nobody wants.

(* if by nice you mean moderately revolting.)

(Ok, fine, it is kinda nice)

Chaining Commands

We can add dependence on the success or failure of a particular process for the execution of another process.

A && B - If A succeeds (that is, it gives an exit code of 0), then run B

A || B - If A fails (non zero exit code), then run B

Making aliases

alias shortName="your command here" (With the quotes.)

The shortened command can literally be even one letter. This really speeds you up if you find yourself copy pasting a line of code over and over.

Just make sure you don't keep a name that already exists.

Running the above command however, is a temporary measure. When you close the shell, these aliases are forgotten.

To define the alias permanently, we can just run the command as a part of a file that runs with the shell when it boots up: the .bashrc file in home. If you go through the bashrc, they'll recommend that you make a file called .bash_aliases instead, because storing it in this giant file makes it kinda hard to access repeatedly and it's not as pretty to look at.

unalias shortname will remove the alias. Using the -a option instead of any particular name (unalias -a) will remove all aliases.

If you alias a command to a name that already exists, (say, you aliased cp to cp-i) and now you want to use the original command (cp), you needn't unalias your command; Simply use / to escape the alias. (like /cp)

In case you want to see what a command is aliased *to,* as mentioned earlier, you can use the type command. Alternatively, alias name will have the same effect.

Command substitution:

This is like a function call. Okay, it’s closer to python’s formatted strings. (f’blah {expression} blah”)

In any command you want to type, you can run some other command in the middle of it by using this. To achieve this, we just put a $ sign and follow it with the expression we want in brackets.

```
$(...)
```

For example,

```
echo $(date)
```

(btw, echo is essentially a print statement. It takes stdin and gives it to stdout. Mentioning coz I just realised I never did that.)

This allows us to achieve a more complex command whose arguments themselves are fed by other commands. To do this a little more neatly, we can use a variable to store the command we want to substitute. In general, any variable can be declared as x= whatever. (NO SPACES)

The variable can then be called later using $x. To store the evaluated output of a command in a variable, we use the same $(…) technique.

Alternatively, backquotes, these ones: ` ` can be used. (The backquote is apparently above the tab button.)

This is a legacy feature superseded by $(…) So, avoid it, but I’ll demonstrate with it just to show how things work.

```
x=date

$x
```

Gives today’s date.

This works?

Not really the way you may think. Here, x is just the *string* date. Because $x simply dumps the contents of x onto the command line, date, a valid command, got executed, and gave an output on the *second* line. If x had been some other string, these lines would yield the error “no such command exists.”

```
echo $x
```

Prints “date”. (The variable's contents is merely date, it was never executed. The variable which was just a string remained that way.)

```
echo $($x)
```

Gives today’s date. (Now, the variable's content is passed as an expression into $ as a command substitution line.

This is not the behaviour I first wanted to implement. I want to store the output of the date command into the variable x.

```
x=`date`
```

This does exactly that. The backquotes executed the command and stored the output in x.

(Use, x=$(date) instead.)

(Important notice: Variable names cannot start with a number and must comprise only of alphanumeric characters + underscore. Once again, note that the declaration statement does *not* have spaces on either side of the equals.)

```
$x
```

Throws an error, because x contains the evaluated output of the command date, and no longer constitutes a valid shell command.

```
Echo $x
```

Gives today’s date. It’s the output that was stored, after all.

This leads us to realise that actually, In the first case with $($x), we get the time of the instant we run the command, whereas in the second case, the time printed is old time which was created when the x = `date` expression was evaluated.

Shell and Environment Variables

I’ve already introduced declaring and calling variables in the command substitution section.

Escaping variables

Sometimes, we may want to use the names of variables with $ signs, but without them being interpreted as variables:

echo “user is $USERNAME”

vs

echo “user is /$USERNAME”

In the first case, the value of the variable is put into the string.

In the second, the backslash prevents the variable from taking up its value.

The same effect can be achieved if you use single quotes instead of double quotes. Anything inside single quotes is pure text. Please, always use single quotes when you have proper text. Shell interpreting and mangling your strings is the last thing you want.

To delete a variable, use the unset command (unset x)

Testing the existence of variables

To test if a variable exists, we can use a command called test.

To be fair, this isn’t the sole purpose of the command “test”; it’s much more general.

A square bracket, is another way of evoking the test command.

(which [and which test do return distinct addresses, and both of them are distinct *files,* no soft links whatsoever. However, man [takes you to the manpage of *test* only.)

The command is very straightforward: inside the square brackets, you put a boolean expression. The return code of the test function will be 0 if it's true, and 1 if it’s false. (Again, 0 is *success*)

[-v myvar] ; echo $?

Thus tells you if myvar exists. The -v option checks for the existence of myvar. (Check this $? if needed.)

Double Square Brackets are apparently an upgraded version of [

The double square brackets allow usage of Regular expressions.

Annoying note: You ought to have a space after the opening and before the closing brackets.

There exists an option -z that checks if the length of a string is 0. (True if len=0)

[[-z ${myvar+x}]]; echo $?

Thus indirectly tests the existence of myvar.

I just don't understand how or why this works.

Side note: it's always a good practice to use the curly braces to enclose any variable you want to use the dollar sign for. The braces do not mean anything special; they just make things clear to both us and the computer and avoid wrong interpretation of code.

Conditional usage of variables

echo ${myvar:-x} try using myvar, if it doesn't exist, use x.

echo ${myvar:=x} if myvar doesn't exist, declare it to be equal to x; use that value.

echo ${myvar:+x} try using myvar, if it exists, run/use x. If it doesn't, just stop. Here, the argument of echo becomes x if myvar exists.

String Slicing and Matching with Variables

echo ${myvar:3:6} (From 3 , Take 6 characters, if 6 is not available, it's ok.)

echo ${myvar: -3:6} (Negative indexing is perfectly fine) (Notice the space before the negative. This is because :- means that you should use the value succeeding the - if myvar does not exist.)

echo ${#myvar} (Returns the length of string myvar)

```
echo ${myvar%pattern}
echo ${myvar%%pattern}
```

```
echo ${myvar#pattern}
echo ${myvar##pattern}
```

Percent matches the pattern from the end and removes it. One is minimal, Two is maximal.

Hash matches the pattern from the front and removes it. One is minimal, Two is maximal.

Example:

```
x="myfirst.hello.jpg"
echo ${x%.*}
myfirst.hello
echo ${x%%.*}
myfirst
```

```
echo ${x#*.}
```

```
hello.jpg
echo ${x##*.}
jpg
```

It kinda annoys the hell out of me, and I don't wanna use it, but I gotta admit, ##*. Seems like a very efficient way to get the extension of a file.

```
echo ${myvar/pattern/string}
echo ${myvar//pattern/string}
```

// Replaces every occurrence of pattern with string. (/ first occurrence only)

```
echo ${myvar/#pattern/string}
echo ${myvar/%pattern/string}
```

/# replaces only if pattern is seen at the start.

/% only if its at the end

`echo ${myvar,}` Change first char to lowercase

`echo ${myvar,,}` Change all

`echo ${myvar^}` Change first char to uppercase

`echo ${myvar^^}` Change all

Type defining variables

declare -i x; (x can only be an integer)

-l (only lowercase chars)

-u (uppercase)

-r (read only)

declare -a myarray;

myarray[indexno]=setvalue

echo {#myarray[@]}

Returns the length of the array

echo {!myarray[@]}

Returns the indices that contain a value.

The @ in the brackets is a way to refer to all possible entries in the array.

The weird thing about these is that you can use any number you like as the index, it needn't be the first n natural numbers.

declare -A myarray;

Capital A makes it an associative array, a dictionary. You can use any key, not just integers as indices.

declare -x will export the variable to child shells and processes.

Some common default variables

The variables we were defining were stored in the shell only, making them shell variables. Apart from the user defined ones, there are tons of default ones.

Default shell variables are always in ALL CAPS, and will be accessed with a $ like any other variable.

Some frequently used variables are: (Attaching dollars coz it just looks *weird* without it)

$USERNAME

$HOME

$HOSTNAME

$PWD : present working directory

$PATH : the directories which Bash will scan for valid commands.

$0 : Name of the Shell

$$: Process ID of the shell

$- : Flags set in the bash shell

$? : Return code of the previously run program

$?

is already something we are sort of familiar with: It gives the exit code of the last run program. An exit code of 0 denotes success, 1 failure, 2, misuse, 127, command not found, 130, ctrl C termination and 137, process killed by someone. (Eg: kill -9 <process id>)

The exit code uses 8 bits, so takes only values 0 to 255. If you design your processes to give an exit code not in this range, it will simply become modulo 255.

To see what the variables hold, you can just do

```
echo $USERNAME
```

or

```
printenv USERNAME
```

In the absence of an argument, printenv will just print all the shell variables it can find. (+ a shitton of other things)

env does a similar thing.

Since both of these are overkill (and ugly), we can use echo for a more specific case:

echo {!P*} will print all the variables starting with capital P.

Side note: echo * actually prints all the files in the pwd, just like ls.

Spawning Shells

When you run the command "bash" a new subshell is spawned.

Run ps -f to see for yourself.

You can do all the subshellception you like.

One important thing to note is that shell variables that you define do not transfer to subshells. So if you want a shell to send out a variable to all its children, you must use

```
export myvar
```

However, any changes you make inside the child shell will not reflect on the original variable. The subshell is merely sent a copy, not the original. To see this for yourself, change myvar inside the spawned bash shell and then type "exit" to return to the original shell.

```
echo $-
```

Tells us "himBHs" which is just a list of all the options of our bash shell.

Man bash tells you what all these options mean.

h : locate and hash commands

B : brace expansion enabled

i : interactive mode

m : job control enabled

H : ! style history substitution enabled

s : commands are read from stdin

c : commands are read from arguments

When we spawn a shell with bash, we can give limited options to suit our needs.

bash -c "echo \$-; exit 300"

The -c option tells the command to automatically exit the newly created bash shell after performing the given task. Notice how I used a backslash for $- ; this is because if I don't, the variable will get evaluated when the command is first interpreted, returning himBHs once again.

This spawned subshell returns only hBc, the bare minimum required to do the given job.

I've used the exit command with a custom exit code, (which is, again, taken mod 256) just to show that you can do this.

The H option is really useful. Since the default shell has it, you can browse through the history of all the commands written into the shell, and call them once again.

To call an older command, you start the line with an ! (called a "bang") and follow it up with a line number.

The command "history" will show you the history for the past few eons, and you can call line no n with !n.

!! calls the last command, and is synonymous with !-1, which in my opinion is kind of redundant when you have an up arrow.

To look for a specific thing you typed in the past, you can pipe history's output into grep, or just use ctrl R to start a recursive search of the history, which is a tad bit more convenient. Hit ctrl R again to browse through the matches.

The -B option, Brace expansion, is something to do with echo, which is why it got triggered in the above example. It's easier to just show Brace expansion than to describe it:

echo {a..z} gives a b c d e f g h i j k l m n o p q r s t u v w x y z

echo {A..D}{X..Z} gives AX AY AZ BX BY BZ CX CY CZ DX DY DZ

Demonstrative problem: Accomplish the following in *one* line of code:

Create a file documents.txt containing all the possible file names in the format file_XYZ.txt where X is a lower case alphabet, Y is also a lower case alphabet and Z is a number between 0 and 4.

Few examples of file names in this format are 'file_dh4.txt', 'file_sd1.txt', 'file_ja0.txt', 'file_at2.txt'.

The file names in documents.txt should be separated by a single space.

SPOILERS AHEAD

Answer :

```
echo "file_"{a..z}{a..z}{0..4}".txt" > documents.txt
```

Process Management

Spawning processes:

coproc command

(CO–PROCess)

Sends the process to the background, and reports back to the shell once it is done. The control of the shell isn't lost just because that process takes time to run.

Suffixing & to the end of the process has the same effect, except it never reports back.

When a process is running, you can ctrl Z to STOP the process, but keep it in a suspended state.

Killing Processes:

kill -9 <pid>

As in reality, there are various levels of brutality to the act of killing.

kill -L

Will list all these ways of murder, and you can pick your poison.

SIGTERM is a *graceful* way of killing things. It kind of politely asks you to die repeatedly.

-9, SIGKILL, is the red-eyes-Chara way of ending things. It won't just kill the process, it will go and find every last one of its children and stab them repeatedly as they beg for mercy, till their throats run dry from screaming, and they slowly bleed to death.

SIGTERM will not go after the children; it isn't so heinous.

In the absence of any options, kill will send a SIGnal to TERMinate only.

Note: Ctrl C sends a SIGTERM.

Looking at processes:

top will show you a list of the background processes in order of CPU consumption, in a really neat table that refreshes every couple of seconds.

jobs will show you the jobs you have sent to the background.

fg will bring the last one you sent behind back to the ForeGround.

Chapter VI: Bash Scripting

This is in essence, writing your own commands.

When you wish to do so, keep in mind these guidelines:

Don't be chatty - if all is well, do not output to the terminal.

Generate output in the same format as input

Let someone else do the hard part.

Do one thing well.

Default to stdin/out

Shebangs

Any piece of code you write needs to be identified as a particular language and then interpreted. To accomplish this, we declare the path to the interpreter in the first line. Always.

It's called a shebang. Without it, the code is meaningless. For bash scripts, the shebang is

#!/bin/bash

Running a script

There are two ways to run a script, first, sourcing and the second, executing.

Given a shell script, say abc.sh, you can do

source abc.sh or equivalently, . abc.sh

Dot isn't exactly an alias to source. Dot is the original. It's the POSIX standard. Source tries to do the same job.

./abc.sh executes the script.

The second method requires that you have executable permissions, and actually runs in a child shell. This can be seen clearly if you run ps --forest inside the script. When sourced, it will remain with the PID of the original shell.

The key difference between these two methods is that sourcing is for setting up an environment, while executing is typically for executing some commands on files and stuff and changing them to enable some functionality.

For instance, I need to *source* the ros noetic setup.bash file. This is because I want the ros commands to be recognised by my parent shell. I want all the env variables and things the script summons to be available in my parent shell.

When you execute a shell script, it does its job inside a child shell so all the variables generated in the process, environmental changes etc, will be lost. When you source it, you never leave the parent shell, so any changes will be retained.

Check this answer on superuser for some nice examples.

Calling variables

Normal variable definition statements can be put into your script. Calling them will have to be done only with a dollar sign, *everywhere*.

Most shell commands we know, depend heavily on the options passed to it. In our scripts, we can call the arguments passed using $1, $2 and so on. $0 is the name with which we invoke the shell script. This may be used to change functionality, like in the case where we saw soft links pointing to the same code. $# contains the number of shell arguments given.

The if statement

An if statement is written by using if followed by a space followed by a *command* whose exit code is checked to be true or false. (Again, trippily enough, 0 is success, but we will personally have to *never* think about this in that way.) At any rate, this is the ideal use case for the test command.That is , [[. (To be fair, just [, but I'll use [[since it's there. [[isn't there in the posix standard though.)

There's still one more thing we need to discuss before we can write if statements without a hitch: doing arithmetic. It's not a trivial issue, and it's something you'll use in a lot of if statements.

Doing math in bash

Firstly, you cannot just do something like [[$x>5]] because the > sign does *string* comparisons.

Always use -gt -lt -eq -ge -le for comparison operators.(>, <,==,≥,≤ respectively)

Secondly, [[$x+5 -gt 20]] won't work either.

x+=5 will not as well.

You'll need to wrap any arithmetic operations in ((...)) for it to take effect, like [[(($x+5)) -gt 20]] and ((x+=5))

(Side note, I use x and not $x here because this is a redefinition statement)

Actually, there are 4 ways you need to note,

1. The let command.

x=$1+5 does not work.

let x=$1+5 will do the job.

Since you are forced to never space separate things here by bash, you can bypass it to make it look clean by using double quotes like:

let "x = $1 + 5"

2. $[expression]

x=$[$a+20]

3. The expr command

This command executes an expression and prints the output.

expr $a + 20

expr "$a + 20"

b=$(expr $a + 20)

I've got to do a command substitution to store the output in a variable.

4. ((...))

This is the most convenient one in my opinion.

b=$((expr $a + 20))

Mind the $.

((b++))

That's about it.

Sample Problem:

Write a bash script that reads a value from the standard input stream and prints PNUM if the value is a positive number or 0; prints NNUM if it is a negative number; else print STRING.

My Answer:

```
read num
re="^[+-]?[0-9]*([.][0-9]+)?$"

if ! [[ $num =~ $re ]]; then
echo STRING
elif [[ $(bc <<<"$num>0") -eq 1 || $num -eq 0 ]]; then
echo PNUM
elif [[ $(bc <<<"$num<0") -eq 1 ]];then
echo NNUM
fi
```

The issue in this problem, is that bash *doesn't* do float arithmetic, and I failed all the test cases with decimals when I used (( )). Same thing holds with the checking inside [[$num -gt 0]]

That's why I had to outsource the work and pipe the evaluation into bench calculator (Yes xD, that's what bc stands for)

In general, you might want to feed a multi line query to bc to do a floating point operation, to assign to a variable or something. To do this, we use something called the "heredoc" feature.

(*Here is a document*)

```
d=$(bc -l << END
scale = 5
($a+$b)^$c
END
)
```

The double inward says we are piping in commands, and the END specifies the string that will specify the end of the doc. This should be exactly the only contents of a full line to denote the end.

Note how I didn't use any tabs to indent the lines inside. Those tabs aren't ignored by default. To ignore them, use <<- in place of <<

```
d=$(bc -l <<- ABC
scale = 5
($a+$b)^$c
ABC
)
```

Also note the use of bc -l instead of just bc. This is *essential.* Do *not* forget it. That imports the math library and allows floating point operations.

Important notice: the =~ matches Regular expressions. The regex *needs* to be stored in a variable like I did before I tested with it. Otherwise bash screws up very badly in understanding what the string means. I checked if it was a number, and if it was, used bc, because bc apparently gives success exit codes for string inputs.(super weird, I know.)

The solution code is so much more elegant though:

```
read n
num="^-?[0-9]*\.?[0-9]*$"
neg="^-"
if [[ $n =~ $num ]]; then
  [[ $n =~ $neg ]] && echo NNUM || echo PNUM
else
  echo STRING
fi
```

They just checked if it had a minus sign to decide if it's a negative number, and avoided any form of arithmetic checks.

I'd like to note at this point that the exit command will quit the bash script and give its argument as the return code. For example, you may check if $1 is a number, and if it is not, you can quite the code with a return code of 1 (Error)

Any custom error code is fine, but do know the conventional codes.

The for loop

For loops are simple. For can be followed by space delimited arguments and they will be iterated over. (warning: check 1st mistake in n00b mistakes)

For can also iterate through brace expansions like:

```
for i in file_{a..z}{a..z}{0..4}

do

        echo i

done
```

As I mentioned, by default the for loop separator is space.

We can change the Internal field separator by just redefining it.

Here is a very clean way to see the contents of the path variable and the order in which it is traversed.

If you wish to add scripts to the path, you might wanna add it somewhere near the top, so that something else doesn't come before that script and cause a conflict of some sort.

You ought to change the default path variable to include your custom folder if you do things like that. I don't suppose I'll be doing that level of things anytime soon though. Full on defining custom commands ought to feel cool.

```
#!/bin/bash
IFS=:
n=1
for i in $PATH
do
        echo $n $i
        (( n++ ))
done
```

Hey, Actually, let me add this script as a command called path. It's very clean. OK. Done. :)

That Aside, for loops also work with the c++ syntax. (except you don't need to type define the variables)

```
for (( i = 0; i <=c ; i++ ))
do
      echo $i
done
```

You can use multiple initialization statements, as well as multiple increment operations. However, there has to be just one condition, of course. Separate these by commas like

```
(( i = 0, j=40; i*j >=c ; i++,j-- ))
```

You might want to store output of commands in files. Although you're always welcome to use >,>> and such, there is a very convenient way to redirect the output of just a single loop: put the > after the done statement.

If you're storing to a file in a script, there will always be a high chance that you are overwriting some file, especially if you run that script often. It's thus a good practice to first check if such a file exists, and exit the script if it does, do something appropriate. Alternatively, you can create a file named tmp.$$ meaning that the extension is the process ID of that script, which will likely never be repeated.(For all practical purposes, atleast. It's a *temporary file*. We are *not* making an archive.)

The break and continue commands work inside for and while loops as expected.

While loops are very straightforward:

```
n=10
i=0
while [ $i -lt $n ]
do
        echo $i
        (( i++ ))

        if [ $i -eq 5 ]
        then
                break
        fi
done
```

If you have nested loops, you may want to break out of several loops at once. That is, when you are, say, 3 loops down the rabbit hole, and you encounter a condition that lets you decide that you can stop going through all 3 loops and move on with the rest of the code, you want to break out of 3 *levels*.

break 3 does exactly this.

I *wish* python had this. Setting so many flags and breaking slowly is such a pain. This is just. *chef's kiss*

Defining functions

There is no keyword like def needed for it. Just go ahead and write

fnname(){

….

}

That's all.

$n can call the nth argument to a function. If it doesn't exist, it is treated as null.

Select & case

The case statement is an elegant way to handle an if else swathe that gets too large but is also rather simple. I personally don't encounter too much utility for it, but here is an example to demonstrate its usage. I'll save a bit of space and also find an example with a select loop. A select loop is a *menu based* selection. You ought to select one of the options offered and unless you select a valid option which results in you *breaking* out of the loop, it will keep going.

```
echo select a middle one
select i in {1..10}
do
case $i in
1 | 2 | 3)
echo you picked a small one;;
8 | 9 | 10)
echo you picked a big one;;
4 | 5 | 6 | 7)
echo you picked the right one
break;;
esac
done
echo selection completed with $i
```

Getopts

Although I described the variables that let you call the arguments, it is not nearly as sophisticated as the generic shell command we see; Those guys demand some compulsory options, only some options take an argument that is associated with that particular option, and they reject the function call if you give any invalid option.

All this functionality can be gained by using just one command: getopts.

This lets you develop rather professional looking shell scripts.

You run getopts in a while loop with some boilerplate syntax that looks like this:

```
while getopts "ab:c:" options;
do
        case "${options}" in
                b)
                        barg=${OPTARG}
                        echo accepted: -b $barg
                        ;;
                c)
                        carg=${OPTARG}
                        echo accepted: -c $carg
                        ;;
                a)
                        echo accepted: -a
                        ;;
                *)
                        echo Usage: -a -b barg -c carg
                        ;;
        esac
done
```

That string and the contents of the loop should be the only thing you modify. "abc" means a b and c are valid options. There is no such thing as a *required option*. That's called an *oxymoron*.

If you require arguments, they should be positional arguments to your script.

Your options may take arguments. Just adding a colon after a letter in that string does all the magic of making the user have to compulsorily enter an argument for that option. Be careful though, "a:bc" can be called with -a -b -c and -b will be taken as the argument to -a, calling only the -a and -c options. You may want to see the optarg's format to decide if you can move further or throw an error.

As you can see, we are looping through the options in a while loop, checking where exactly each one matches. Keep this in mind when you make your code. The option being processed at the moment has an argument $OPTARG. We are storing this in a separate variable for our purposes. Don't shove in your code that goes according to each option inside this while loop.

Let the while loop set some flags that decide what needs to be done, and then you can do it outside with an if or a case block.

Important Notice: It is surprisingly easy to make n00b mistakes **without ever realising it** when writing shell scripts. You may use shell check to catch these early on.

Section 2 : Oldschool Content

Chapter I: AWK Programming

Awk is a very powerful programming language, although it is admittedly rather old. Fundamentally, it looks at text based data like it is a spreadsheet. It splits up each line into pieces using some delimiter that you tell it. You can have even more than one valid delimiter.

Awk lets you code with loops and conditions similar to C, allowing us to perform sophisticated manipulations with the fields of the records awk splits up and gives us. Because awk is kind of niche, it gets to be tremendously fast in what it does. Its speed is certainly not shabby if not straight up impressive.

```
#!/usr/bin/gawk -f
BEGIN{
        FS=“,”
}
{
        print $1
}
END{
        print “done”
}
```

There are 3 blocks in an awk script. To be fair, you can have as many blocks as you like.

All the blocks labelled BEGIN will run before any records are processed. They will be executed in the order that they appear.

Similarly, any blocks labelled END run after all records are processed, in the order of their occurrence in the script.

The other blocks without a heading, are the default blocks. They are executed once for every single record that is processed. Instead of a heading, you can give some criteria that are checked before they are executed. This could be a Regular expression.

For example,

```
$1 ~ /[[:alpha:]]/ {
# Code block here
}
```

There are a bunch of default variables that exist, like the FS I had declared there. FS stands for Field Separator. FS can use Regular expressions. So FS =[.;:-] lets me single handedly use 5 different delimiters at once. (Lol. *Just sayin I can.*)

Note: think record = row, field to entry of a column

NR is the number of the record, but this keeps counting even after a file ends.

FNR is the record number in a given file that is being processed.

NF is the number of fields in the current record.

FILENAME is the name of the current file.

$1 $2 … $n gives the nth record. Although $ is the standard prefix for shell variables when they are invoked, awk variables do *not* need a $ prefix when summoned.

Awk variables do not even need type declarations, unlike C or C++. Very nifty.

Operations:

a in array	Python esque array membership check

++	increment
– –	decrement
a ~ /regex/	match to Regular expression
a !~ /regex/	doesn't match Regular expression
a b	concatenate a and b (Just a space)
expr ? a : b	conditional expression.

Unlike python, x not in arr will not work here. Use !(x in arr).

Functions:

There are an absurd number of inbuilt functions inside awk. User defined functions can be given as separate blocks.

To make things cleaner, it is recommended that you keep all udfs in a separate script, and run that script before and along with your main script.

```
Function myfunc(a){
        a=a+0
        return sin(a)
}
```

Adding zero to a automatically converts it into a numerical data type from a string type. Sin is an inbuilt function.

Awk lets you use arrays, but only as dictionaries. I mean, you need to manually give the indexing to an array, and it doesn't default to whole numbers.

Sample Problems:

You are given a file that contains the board exam scores of some students from many different schools. Each line in the file contains four comma-separated fields: school code, roll number of the student, name of the student, and marks in the below format.

School_code,Student_Roll_no,Name,Marks

All the fields could be alphanumeric values except the last field Marks which is a number less than 500, as the maximum marks of the exam is out of 500.

Write an AWK script to print the roll numbers of the toppers of each school. For example if there are details of students of 11 schools in the file, then your output should contain 11 roll numbers in any order, one for each school. Ignore the case of multiple toppers.

My Answer:

```
 BEGIN{
   FS=","
}

FNR==1{
   max[$1]=$4
   maxbois[$1]=$2
}

! ($1 in max){
   max[$1]=$4
   maxbois[$1]=$2
}
```

```
max[$1]<$4{
   max[$1]=$4
   maxbois[$1]=$2
}




END{
   for(i in maxbois){
      print maxbois[i]
   }
}
```

Solution Code:

```
BEGIN {
  FS=","
}
{
  if ($4 > max[$1]) {
    max[$1] = $4
    max_student[$1] = $2
  }
}
END {
  for (i in max_student) {
    print max_student[i]
  }
}
```

Apparently, awk has an interesting approach to undeclared variables. The only extra effort I had put in, was to a) define the variables for the first time; b) define the key value pairs for the first time. This was a python inspired move. But awk isn't python. If a variable has not been declared as anything, it is assumed to be "Null" or Zero.

When printing such a variable, nothing shows up.

This permits you to call non existent keys and undeclared variables in conditional statements without any issue. However, this worked here because 0 was less than any $4 that could show up and acted as the defining statement when it should. This will not work in general.

Also, notice how the for loop iterates through the keys and not the values, just like a dictionary.

Here's another fairly straightforward question:

EmployeeDetails.csv contains the Employee ID, Employee Name, Leaves taken this year and Gender, of all the employees working in a company XYZ.

Write an awk script that takes the file EmployeeDetails.csv as input and prints the name of the employee(s) with the lowest number of leaves taken this year. If there is more than one employee with the lowest number of leaves, print the name of each employee on a new line.

My Answer:

```
BEGIN{
   FS=","
   c=0
}
FNR==1{
   min=$3
}
{
   if($3<min){
      min=$3
      c=0
      ans[c]=$2
   }
```

```
    else if($3==min){
        c+=1
        ans[c]=$2
    }
}
END{
    for(i=0;i<=c;i++){
        print ans[i]
    }
}
```

<u>Solution Code:</u>

```
BEGIN{
  FS = ",";
}
{
  if (NR == 1)
  {
    lowc=int($3);
    count =0;
    name[count] = $2;
    next;
  }
  leave = $3;
  if (leave < lowc)
  {
    lowc = leave;
    delete name;
    count = 0;
    name[count] = $2;
  }
  else if (leave == lowc)
  {
    count++; name[count] = $2
  }
}

END{
  for (i=0; i<=count; i++)
  {
    print name[i];
  }
}
```

Fundamentally, the logic used is the same, but the way we have coded it is slightly different. Firstly, they have used “next”, a statement like “continue” in a loop. I cut down on the double coding by making common code run in the equals section of the code. In addition, I do not delete the array when I find a new minimum, I just overwrite entries and output only the final set of entries.

(To be honest, I did it because I didn’t know the delete keyword, but now that I look at it, it’s quite cool this way)

Thinking about the first question having multiple toppers now makes me realise there is no way to do it easily without having a proper array data type I can append to. The code will very quickly gain complexity. I can’t actually see a way to make array objects without having to declare a name for each array, so I cannot solve this problem.

If I can reuse the name of the school as a variable name for the array that school’s entry points to, then I can use the id of the topper as the key as well as the value to hackishly make an array without maintaining an index for each array. Then I can delete the array before making an update to further avoid any counting business and just print full arrays.

But well, I don’t think you can declare variables on the fly like that.

At this point, using C++ might just be a better idea. Parsing a csv will in itself take a little bit of effort, and it will certainly be slower than awk in racing through files. But there shouldn’t be any limitations like this. :(

Awk does have its uses though. Just not in this case.

Chapter II: SED Programming

Sed, the "Stream Editor", precedes awk. It's included in all linux systems, as part of the posix standard, and it seems to me, a powerful antique with some fairly interesting functionality. As far as I can tell, it isn't as versatile as awk, and I can't really think of too many use cases for it.

Nevertheless, it's an intriguing rapid string processing, pattern recognising command. Here's how it's used:

You'll be feeding a file to sed. You need to first tell sed which lines you want to do something to, then tell it what to do with each of those lines.

(Address pattern) (Actions) (Options for action if needed);

This constitutes one command. (No need of brackets, there's not even a separator, which admittedly makes it quite confusing if you aren't familiar with sed) The overall syntax is this:

sed -options -e 'address {a1; a2; a3;}' filename

The curly braces are being used to perform multiple actions for the same address match.

If you want to execute multiple address match commands, use multiple sets of -e followed by command string, or separate your commands by semicolons inside one large string. Use single quotes, the contents of double quotes will get interpreted by shell and mangled.

Let's explore each of the parts of the sed command I mentioned. Address selection first.

The number n - nth line only

$ - last line

% - all lines

n1,n2 All lines from n1 to n2, both included. (Note the comma as separator) n1~n2 From n1 till end, with every n2 th line chosen. For instance, 1~2 implies 1,3,5…

/regex/ - all lines that match the Regular expression

/re1/,/re2/ - from the first line that matches re1 till the first line that matches re2.

n,/re/ - from n till the first line that matches re. /re/,n also works.

/re/, +5 first line that matches re, and the next 5 lines.

/re/, ~5 first line that matches re, and the next 4 lines. (5 lines in total)

Any of these criteria can be followed immediately by an ! to negate that selection. That is, the action will be performed for all addresses that *don't* fall into this range.

Now that we can select the lines we want to work with, let's look at the actions.

Note that by default, sed will go ahead and print the entire file.

Why?

This lets you do a subtractive filter using an action of deleting from the baseline of print all.

If you want to disable this, you should pass the -n option to sed.

The basic 3 actions are these: p, print, d, delete and s, replace. If you wish to print just some select lines, you should really use the -n option to sed, for if you don't, sed will print all lines, but will print the lines you selected twice. The second copies won't be grouped together at the end, just double occurrences in the place of that line.

s, substitution, works with regex. This obviously needs options, unlike p and d. The syntax is this:

s/regex/replacement/flags

Note: if you're gonna use extended regex, you'll have to include the -E option to sed.

Commonly used flags:

g - Apply the replacement to *all* matches to the regexp, not just the first.

Number - Only replace the numberth match of the regexp.

p - If the substitution was made, then print the new pattern space.

I or i - makes sed match regexp in a case-insensitive manner.

Aside from these 3 fundamental operations, there are many more:

i - inserts a line of text in the line above the matching lines

a - "append" inserts a line of text in the line below the matching lines

i and a option syntax: just give a space and follow it by your new line of text.

= - prints just the line number of the matched line

q - quits sed. This action can be given with only one address, not multiple.

There are a lot more options, many obscure. Check this if interested.

I think we are actually pretty good now with the basic functionality of sed.

I mentioned that if you wanted to run multiple commands, you could separate them with ; and do so. If you wish to perform a sophisticated operation, this may get cumbersome. So, you can write a sed script, with the shebang #!/usr/bin/sed -f. In each line of this script, you give a proper command. Things look very tidy.

To use this sed script, use the -f option (file) : sed -f mysedscript.sed filetobeprocessed

In my opinion, we have already acquired the ability to do quite a lot of text processing with sed, but the abilities of sed *far* exceed this. Sometimes, when it comes to the more sophisticated things, using awk might be a better option, because trying to use sed will straight up give you a headache. The thing is, awk works with delimited, structured data. Sed can process any string, and come out with clean results. The use case for awk and sed is in fact, very different, so looking at awk as a replacement for sed isn't exactly correct.

There are 2 main concepts that are more sophisticated that I wish to discuss: programmatically branching to make *loops* in sed, and the *pattern* and *hold* spaces.

Firstly, the pattern space or pattern buffer, is a temporary space in which, typically, only the present line exists. When we do any operations, like substituting, print etc. it is always done on the pattern space. Every time a new line is scanned the pattern space is wiped clean and filled with just the new line.

The hold space is a longer duration storage that does not involve itself with the story unless you specifically bring it in.

You can utilise the hold space to store several lines you have read in the past, and then just fill the pattern space with it one day and do something more grand.

For instance, consider this lovely example I found on stackexchange:

```
sed -n '1!G; h; $p'
```

Command 1 : in lines except line 1, do command G. G appends the contents of the hold space to the pattern buffer, separating it by a newline.

Command 2 : in every line do command h. The h command inserts the contents of the pattern buffer into the hold space, and wipes away anything that was there in the hold space prior.

Command 3 : in the last line alone, print the pattern space.

This simple command now does:

1: No G. h=> hold now has line1. P won't be in the picture for a while.

2: G => line 1 is appended to pattern space. Pattern space is now line 2

line 1

H => this is stuffed into the hold space.

3. G. Pattern space at 3 : line 3

Hold space at this point:

line 2

line1

New Pattern space:

line3

line2

line1

H => New hold space is the first 3 lines in reverse.

For the last line, this will have the entire file in reverse order, ready to be executed by the print command in one shot. The output matches the output of the tac command.

That conveys the basic idea. I'll stop here. That link has a lot of details and is pretty well explained. Do read it if you're interested in taking sed usage to its full potential.

Now, that programmatic shite.

The thing is sed does not have conventional if then statements, or normal "for" or "while" loops.

Instead, it has something like GOTO. Sounds like a dangerous idea tbh. From my little experience, GOTO rarely ends well, but that's really how things work at the assembly level, when you think about how exactly a while or a for loop is implemented.

Anyway, sed lets you go to a place you had labelled before, called a branch. Sounds simple enough.

Here are the letters you need to know:

b

branch unconditionally (that is: always jump to a label, skipping or repeating other commands, without restarting a new cycle). Combined with an address, the branch can be conditionally executed on matched lines.

t

branch conditionally (that is: jump to a label) *only if* a s/// command has succeeded since the last input line was read or another conditional branch was taken.

T

similar but opposite to the t command: branch only if there has been *no* successful substitutions since the last input line was read.

As a real-world example of using branching, consider the case of quoted-printable files, typically used to encode email messages. In these files long lines are split and marked with a *soft line break* consisting of a single '=' character at the end of the line:

```
$ cat jaques.txt
All the wor=
ld's a stag=
e,
And all the=
 men and wo=
men merely =
players:
They have t=
heir exits =
and their e=
ntrances;
And one man=
 in his tim=
e plays man=y parts.
```

We can run the following sed script to merge that into some readable text:

```
:x
 /=$/ { N ; s/=\n//g}
 bx
```

When line 1 is read, if it ends with =. N reads line 2, cuts out the = sign and the newline. Sticks it line 1 and again takes the next line of input, till a point where there is no = sign.

Now this terminates.

Now we move on to the next line, as usual. Typically it will be line 2 after line 1, but now we can consume a lot more than that, so whatever is next comes in. This is called a new *cycle*. An execution of the sed command sequence on a *new* line from scratch, rather than some crazy looping that kept going on when it was initiated by the first line.

This entire thing is from that link. They have explained it very well, so there is no need for me to redo this job. I have conveyed the basic idea here, but if you ever look at some problem you want to solve in your life, and your spider sense tingles to say "maybe use sed?" then you can always look up that link to utilise all the features that may aid you on your way.

With that I'll wrap up this section.

sed rox.

Chapter III: Command line Editors

Personally, I'm more than happy with gedit.

I'm going through these because a) they are famous b) they are in my course and c) the keybinds used here kind of propagated out into the programming world and are fairly common.

I was introduced to ed and emacs, apart from nano, but really, knowing one among emacs and vim is enough for my tests, since they let you choose the question between the two.

I always choose vim, of course.

The Vi editor

There are 3 modes in this editor: 1) the insert mode to directly insert text into the file and

2) the command mode which lets us control things better 3) The navigation mode where you just move the cursor

To enter the insert mode, you simply hit i. (pressing a and o works too; a moves to the next character before starting, and o to a new line.)

To enter the command mode, you press esc. To type commands, you need to start with : else you'll remain in a navigation mode.

Although arrow keys do work, people prefer hjkl to move around.

H left, L right, J down, K up.

cntrl+f Scroll forward one screen

cntrl+b Scroll backward one screen

cntrl+d Scroll down half screen

cntrl+u Scroll up half screen

cntrl+l Redraw screen

cntrl+r Redraw screen removing deleted stuff

0 start of the current line

$ end of the current line

w beginning of next word

b beginning of preceding word

:0 first line in the file

1G first line in the file

:n nth line in the file

nG nth line in the file

:$ last line in the file

G last line in the file

To save your edits, use :w. To write changes and quit, use :wq. This is equivalent to :x

:q! Will quit without saving changes. :q will quit if you have written out all changes.

Here are some things you can do while in the nav mode, without hitting : to enter the command line mode.

x delete single char under cursor

Nx delete N chars from cursor

D Delete rest of the line, from the char under cursor

dw delete one word, from the char under cursor dNw delete N words, from the char under cursor

dd delete current line

Ndd delete next N lines, starting from current line

yy copy current line to buffer

Nyy copy next N lines, including current, into buffer

p Paste buffer into text after current line

u undo previous action

Section 3 : Bonus Content

Software Management

Managing software is fundamentally a rather complicated process. Most of the details are abstracted away through the use of a software manager. For Debian operating systems, this is done by the "apt" (Advanced Package Tool) manager.

apt-cache

The apt-cache command lets you query the cache maintained by apt about all the packages it knows of, and all the packages you have installed.

apt-cache search keyword

Looks for the keyword in all indexed (not just installed) package titles and descriptions.

(Keyword can be a Regular expression.)

apt-cache pkgnames | sort | less

This lists all the indexed packages. Since this list might be rather large, I'm piping it to less. (By rather large, I mean only about 80,000 lines.)

Since it won't be in any order, it's a good idea to sort it first.

apt-cache show packagenamehere

Will show you the details of a particular package.

Hashes

On a tangent, I'd like to note that there exists something called a checksum to verify a file's integrity.

Essentially, there is some algorithm that takes into account every bit of a file to generate a short string, say, of 256 bits. Even if you modify that file, by one bit, the hash is radically different. So, if a file has been maliciously manipulated, even by a little, the checksum won't match.

Since there are a potentially infinite number of documents, but only a finite number of hashes, it is true that an infinite number of files map to each hash. However, it is *not* easy to artificially induce a collision like that.

That is, it is *very* difficult to engineer the bits so that your malicious modified file gets the same hash by sheer trial and error, or by use of some clever trick.

Older hashing algorithms like md5, are now totally broken, because computers are just so much faster than they used to be, and we can now successfully make deliberate modifications to get the same hash.

With the development of computers the standard hash size gets pushed higher and higher up, SHA1 is now less preferred, SHA2 is a reliable standard, and SHA3 is the upcoming one.

Fundamentally, a hash function satisfies 3 properties: It is quick to compute the hash of even a very large file; It is not so quick that you can manufacture a file with the same hash; and Even a small change to the source radically changes the output.

Presently, even if you had a lot of money and resources, it would take you more time than the present age of the universe to brute force find an alternative message with a given SHA 256 hash.

If it's still bugging you that a collision is inevitable, it is because you can't quite process with a monkey brain how big the possibilities of 256 bits are.

Let me try to put this in perspective.

There are a *lot* of stars in our galaxy.

There are more grains of sand on earth than there are stars in the milky way. (10 million times more)

Now that’s a lot of sand.

Now, in *one* grain of sand, there are more atoms, than there are grains of sand on earth.

That’s a lot of atoms.

Now, the number of hashes SHA256 can make, is a billion billion times the number of atoms *on earth*.

All the atoms! On Earth! 10^{18} times that!

Anyway, for all practical purposes, a hash is a one to one map that is easy to compute in one direction and basically impossible to engineer to be what you want it to be.

This is *not* encryption. You cannot retrieve information from a hash. It’s a secure *signature* that is secure only by the fact that we can handle big numbers.

Let’s say you’re developing a website. You are facebook. You have to store the passwords of your users, so that you can verify who they are every time they enter. However, if you simply store the plain text passwords in a database somewhere, you’re in a dangerous position. If someone hacks into the database, or someone misuses access privileges, you are in for a data breach.

That’s why someone had the bright idea of not storing the passwords at all. Instead, they thought, why not store the *hash* of the password?

(Although I discussed hashes as if they’re for files, a file is basically a string of 1s and 0s, no matter how long it may be. The hash algorithm only takes in some string input, so it doesn’t matter if it’s a word or file)

This way, even if the database is leaked, nobody can tell what the password was, and when you want to check if a user is entering the correct password, you just need to check if the hash matches.

Seems like a brilliant idea!

Sometime in 2013, Adobe lost millions of emails, hashed passwords, and hints in a breach. You take one look at it and see that a million people have the same hashed password. You look at the hints and can probably guess the password, and when you do, you can verify for a fact that you are right, by computing the hash for yourself.

That is to say, it’s still a bad idea. Some passwords are common, the 1234567s and password123s are all going out in one fell swoop, because if not the hints, the commonality of the hash will give them away.

To use such a breach to your advantage, you can manually compute the hash of a ton of common passwords till you get a matching hash. This is computationally expensive, but doable, assuming we are targeting weak, predictable, common passwords.

Of course, someone already precomputed these hashes and put them in a table. Now, anyone can use such a table to look up the password the hash corresponds to. These "rainbow tables" are *big*, and they'll *probably* break your password, *fast*. (If your password is common and insecure)

So, how *does* a company store a database of passwords?

What they do is generate a randomised string and suffix it to your password before the hash is computed. So, if your password is 1234567, and you are given a random string eLK5Nt9N1o, the hash of 1234567eLK5Nt9N1o is computed rather than 1234567 itself. Now, this random string, called a "salt", is stored in the database. You need to look up the salt, suffix it to the user's input every time they log in and compute the hash to verify.

Since every user is allocated a unique salt, firstly, you cannot just pick out all the password123s by just looking. You can no longer get away with quickly looking up passwords in a rainbow table either, because we are now storing hashes of garbage words, not common ones.

If you want to know if a person's password is 1234567, you will need to suffix *their* user's hash (which will also have been leaked with the database), to 1234567 and then verify it. That is, even that previously trivial task is now a bit of a pain. If a user has an actually secure password, then you're probably not gonna be able to crack it at all. Even if it's a fairly common password, you cannot just look it up in a rainbow table, you *need* to compute *every single* user's hash and check even to catch insecure passwords.

Let me put it this way, you need to build a rainbow table, for *every salt*, and every user has their own salt, and this table will break the password only if it's a terribly insecure one.

This is again, in the event of a breach. Ideally, you have a system that won't face a breach, but this is a pretty good fallback mechanism. We use

a) A dynamic randomly generated salt for every user

b) A particularly slow hashing function, not a superfast commonly used one like SHA1, so that it is by no means an easy task to crack the hashes

c) An iterated hash (apply a number of different hashes a multitude of times) so that it's not even easy to accurately recreate the hashing process by someone who doesn't know what you're doing.

Having discussed hashes to some detail, let me just conclude this detour with a few commands you can run:

```
sha256sum file1.txt
```

```
sha1sum file1.txt
```

```
md5sum file1.txt
```

Are all commands that generate the respective checksum for you to verify a file's integrity.

Really, this only checks *integrity* not *authenticity* because if a malicious person tampered with the place from which you downloaded the file, they were probably capable of changing the checksum too. This only tells you the file promised and the file received are the same. You'll have to use other means to verify if the person doing the promising has not been compromised.

Sudo

This stands for superuser do, and is thus pronounced sudu, not like pseudo. Nevertheless, I will continue saying sudo regardless of the proclaimed propriety.

A list of the superusers is stored in /etc/sudoers. You'll need sudo privileges to even cat the contents.

If you do *not* have the permissions, but you still try to run a sudo command, then the computer will inform you that you are not in the sudoers file, and this incident of you trying to gain sudo access will be reported. By reported, it means that it will log it somewhere a sudoer can look.

This log is in /var/log/auth.log

It'll tell you about every time a person gained or was denied access to sudo privileges.

apt-get

```
sudo apt-get update
```

Will fetch from the internet, (It looks up the official ubuntu websites mentioned in /etc/apt/sources.list and the directory, sources.d in the same folder.) all the details about all the indexed packages.

This will update your apt-cache with a newer list of packages, in addition to the primary function of finding out if the software installed in your system has seen any updates that you have not installed yet.

```
sudo apt-get upgrade
```

Will actually go ahead and download the updates to your packages.

You may notice that the blob of text this command ends with a suggestion to run

```
sudo apt-get autoremove
```

This command will delete any packages that are strictly worthless because they have been superseded by some other package, and are now totally outdated.

To delete a particular package, use

```
sudo apt-get remove packagename
```

Do not do this to system packages unless you know what you’re doing. Every package has a priority level: required>>important>standard>optional>extra

You can know about all the details of a package using apt-cache show packagename including the priority level.

To install a particular package, just use

```
Sudo apt-get install packagename
```

Using sudo apt-get reinstall packagename will check the files and fix anything that you screwed up in that package.

An alternative to apt is snap, which offers much the same functionality. Don’t accidentally install the same package via both. I’d recommend using only apt, but ymmv.

Hardware Things

I'm going to make this a rather brief section. There are a lot of details when it comes to hardware specifications. Building a PC has driven me sufficiently insane to understand that much. Even so, knowledge only leaves you with a craving for these obscure details, when they may become remotely relevant so here are some commands that will tell you those details.

Battery: "unpower -e" will give you a few entries. Use "upower -i path" where path is one of the outputs of the previous command. Do this in particular to the output that looks like a battery. You should be able to see among other things, the peak capacity to which the battery can now charge, and the capacity it was manufactured to be charged to.

RAM, Processor and others: dmidecode (This needs superuser privileges since it goes quite deep into the system to retrieve the data)

Dmidecode gives you a ton of info, use --type memory to know just about the RAM modules, or --type processor to know about that and so on.

```
sudo hdparm -Tt /dev/sda3
```

This will test your storage device for response speeds and data transfer rates. That /dev/sda3 is my hard drive, my nvme has a much longer name but is some path starting with /dev/nvme…

You can look up the names with du -h or through your file manager.

Terminal Prompts

This is just a short section on making some purely cosmetic changes to the way the terminal prompts you. There are 4 distinct prompts we see:

1) The normal prompt before every command
   uka@UK-HP:~$

   Default is username @ hostname : Working Directory $

2) A prompt to complete a command or string if it is left incomplete
   uka@UK-HP:~$ echo 'hello

   > every

   > body'

   Default is just >

3) The prompt in select loops
4) "Prompts" in execution trace outputs (I don't consider these prompts but ok)

Each of these prompt strings are stored in a shell variable. PS1 PS2 PS3 and PS4 corresponding to the ones I have listed above. (Yes, PS is no longer just playstation or ponniyin selvan)

PS1 is probably what you want to customise. I couldn't give less of a shit about the others.

There are a bunch of escape sequences that are used to write this prompt string.

\u is username

\h is hostname till the first period

\H is full hostname

\w is pwd

So the default prompt is "\u@\h:\w$ "

However, if you look at the PS1 declared in your .bashrc, it will look a lot longer: This is just to colour code the various parts of the prompt. Changing it to the above mentioned one will show you the *same prompt* but without any colour.

You may like to keep the time as your prompt:

\A Time in 24 hour hh:mm

\t Time in 24 hour hh:mm:ss

\T Time in 12 hour hh:mm:ss (But this does *not* show if it is AM/PM)

\@ Time in 12 hour hh:mm AM/PM

You may want to keep a line number, counting the no of the command since the start of the shell.

\# does this.

Just replace the \u things in the older PS1 with the things you want if you'd like to retain the colours. It might be a good idea to keep a copy of the original PS1 commented out nearby in case you wish to revert back to it someday.

ssh

Secure shell. Use ssh username@server to connect to that server.

This can be used to remotely access other systems.

The -p option lets you choose which port to connect to.

I won't bother talking more about this because I am lazy.