

# UniSat 01 --Introduction to Bash (Command Line Interface)on Unix (Linux)

---

Written @ 14 March 2020 10:09:30 PM +06 with ❤️ by Azat @ AzatAI

## UniSat 01 --Introduction to Bash (Command Line Interface)on Unix (Linux)

Some general rules

Quick Start

Basic navigation

`pwd` -- It tells you what your current or present working directory is.

`ls` -- Next we'll want to know what is there

path

`cd` Change Directories - ie. move to another directory.

Activities

Files

Everything is a File

Linux is an Extensionless System

`file [path]` -obtain information about what type of file a file or directory is.

Linux is Case Sensitive

Spaces in names

Hidden files and directories.

`ls -a` List the contents of a directory, including hidden files.

Summary

Activities

Manual Pages

Introduction

What are they?

Searching

`Long` and `Short` options

Summary

Activities

File manipulation

`mkdir` Making a directory

`mkdir [-p]` make parent directories if needed

`mkdir [-v]` print a message for each created directory

`rmdir` Removing a Directory

`touch` Creating a Blank File

`cp` Copying a File or Directory

`mv` Moving a File or Directory

`mv` Renaming Files and Directories

`rm` Removing a File (and non empty Directories)

`rm -r` Removing non empty Directories

Summary

## Activities

### Vi Text Editor

#### Introduction

#### A Command Line Editor

`vi` open file

#### Insert mode

#### Saving and Exiting

##### Other ways to view files

`cat` command

`less` command

#### Navigating a file in Vi

`h j k l` - left, down, up, right

`^ $` - the beginning and ending of the line

`nG` -- move to the nth line

`G` - move to the last line

`W` - move to the beginning of next word

`nw` - move forward n word (words)

`b` - move to the beginning of the previous word

`nb` - move back n word(s)

`{ }` - move forward/back one paragraph

#### Deleting content

`x` - delete a single character

`nX` - delete n characters

`dd` - delete current line

`dnw` - delete n word forward

`دنب` - delete n word backward

#### Undoing

##### Taking it further

#### Summary

#### Important concepts

## Activities

### Linux/Unix wildcard

#### Introduction

#### How they look?

`*` wildcard

`?` Wildcard

`[ ]` wildcard

`^` - not

#### Real life examples

## Activities

### Permission

#### Introduction

#### Linux permissions

`r` - view(read) permission

#### Change permission

#### Set permission shorthand

#### Permissions for Directories

The `root` user

## Activities

## Filters

[Introduction](#)

[Head](#)

[tail](#)

[sort](#)

[nl](#)

[wc](#)

[cut](#)

[sed](#)

[uniq](#)

[tac](#)

[Others](#)

[Summary](#)

[Activities](#)

## Grep and Regular Expressions!

[Regular expression](#)

[egrep](#)

[Regular expression overview](#)

[Some examples](#)

## Piping and redirection

[Introduction](#)

[Redirecting to a file](#)

[Saving to an existing file](#)

[Redirecting from a file](#)

[Redirecting the STDERR](#)

[Piping](#)

[Summary](#)

[Activities](#)

## Process Management

[Introduction](#)

[top](#) [what is currently running ?](#)

[ps](#) [Process](#)

[kill](#) [Killing a crashed process](#)

[Foreground and background jobs](#)

[Summary](#)

[Activities](#)

# Some general rules

---

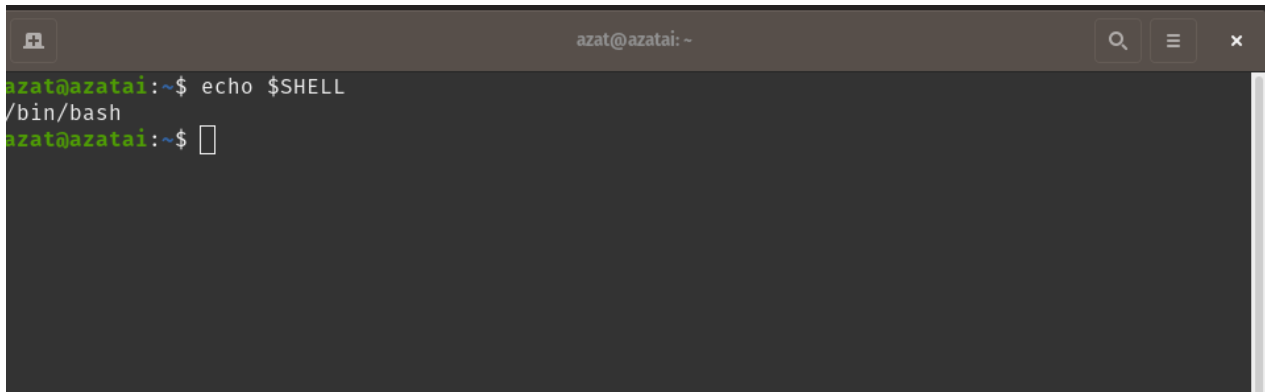
- I'll refer to Linux in the following pages, whenever I do, assume I'm actually saying Unix/ Linux. Linux is an offshoot of Unix and behaves pretty much exactly the same.
- Whenever you see `,` what this means is that you are to replace this with something useful. Replace the whole thing (including the `<` and `>`). If you see something such as `then` it usually means replace this with a number.
- Whenever you see **[something]** this usually means that this something is optional. When you run the command you may put in the something or leave it out.

# Quick Start

---

`echo` to display a system variable

```
echo $SHELL
```

A terminal window titled 'azat@azatai: ~' with search, menu, and close icons in the top right. The prompt is 'azat@azatai:~\$'. The command 'echo \$SHELL' is entered, and the output '/bin/bash' is displayed on the next line. The prompt 'azat@azatai:~\$' is shown again with a cursor.

```
azat@azatai:~$ echo $SHELL
/bin/bash
azat@azatai:~$
```

This shows that I'm currently using `bash` as my default shell.

## Basic navigation

---

`pwd` -- It tells you what your current or present working directory is.

```
azat@azatai:~$ pwd
/home/azat
```

`ls` -- Next we'll want to know what is there

```
azat@azatai:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
azat@azatai:~$
```

`ls` It's short for **list**.

```
**ls [options] [location]**
```

(`[]`) mean that those items are optional.

we may run the command with or without them.

```
azat@azatai:~$ ls -l
total 32
drwxr-xr-x 2 azat azat 4096 Jan 4 14:49 Desktop
drwxr-xr-x 3 azat azat 4096 Jan 4 19:22 Documents
drwxr-xr-x 2 azat azat 4096 Jan 4 19:55 Downloads
drwxr-xr-x 2 azat azat 4096 Jan 4 14:49 Music
drwxr-xr-x 2 azat azat 4096 Jan 4 20:07 Pictures
drwxr-xr-x 2 azat azat 4096 Jan 4 14:49 Public
drwxr-xr-x 2 azat azat 4096 Jan 4 14:49 Templates
drwxr-xr-x 2 azat azat 4096 Jan 4 14:49 Videos
azat@azatai:~$
```

We ran ls with a single command line option ( -l ) which indicates we are going to do a long listing. A long listing has the following:

- First character indicates whether it is a normal file ( - ) or directory ( d )
- Next 9 characters are permissions for the file or directory (we'll learn more about them in section 6).
- The next field is the number of blocks (don't worry too much about this).
- The next field is the owner of the file or directory
- The next field is the group the file or directory belongs to (users in this case).
- Following this is the file size.
- Next up is the file modification time.
- Finally we have the actual name of the file or directory.

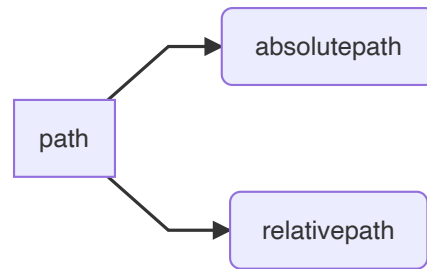
```
azat@azatai:~$ ls /home
azat  lost+found
azat@azatai:~$
```

We ran ls with a command line argument ( /home ). When we do this it tells ls not to list our current directory but instead to list that directory's contents.

```
azat@azatai:~$ ls -l /home
total 20
drwxr-xr-x 19 azat azat 4096 Jan 4 19:12 azat
drwx----- 2 root root 16384 Jan 4 14:29 lost+found
azat@azatai:~$
```

## path

In the previous commands we started touching on something called a path. I would like to go into more detail on them now as they are important in being proficient with Linux. Whenever we refer to either a file or directory on the command line, we are in fact referring to a path. ie. A path is a means to get to a particular file or directory on the system.



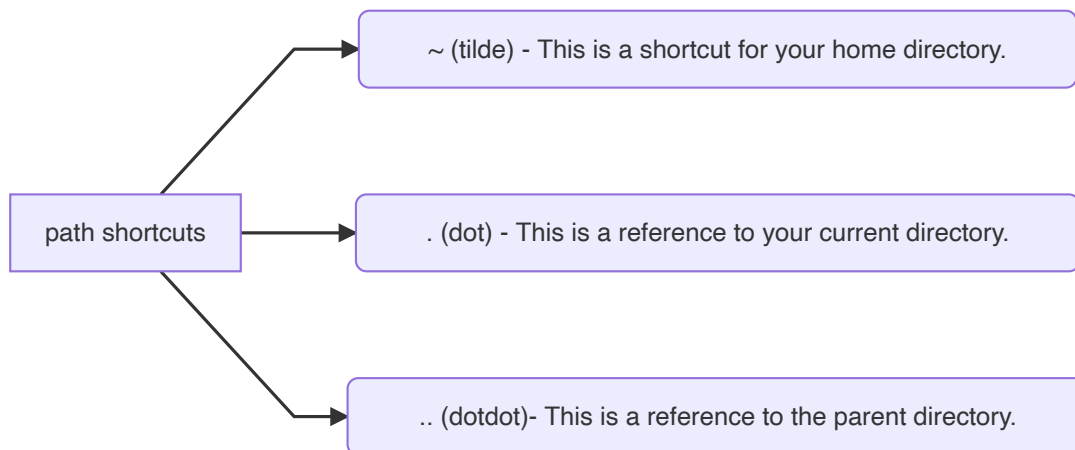
There are 2 types of paths we can use, **absolute** and **relative**. Whenever we refer to a file or directory we are using one of these paths.

The file system under Linux is a hierarchical structure. At the very top of the structure is what's called the **root** directory. It is denoted by a single slash ( / ). It has subdirectories, they have subdirectories and so on.

```
azat@azatai:~$ ls /
bin  dev  home  lib32  libx32  media  opt  root  sbin  sys  usr
boot  etc  lib  lib64  lost+found  mnt  proc  run  srv  tmp  var
azat@azatai:~$
```

**Absolute paths** specify a location (file or directory) in relation to the root directory. You can identify them easily as they always begin with a forward slash ( / )

**Relative paths** specify a location (file or directory) in relation to where we currently are in the system. They will not begin with a slash. (In many times, we like to use a `.` in relative paths)



```
azat@azatai:~$ pwd
/home/azat
azat@azatai:~$ ls ~
Desktop  Downloads  Music  Public  Videos
Documents  Edraw  Pictures  Templates
azat@azatai:~$ ls /home/azat
Desktop  Downloads  Music  Public  Videos
Documents  Edraw  Pictures  Templates
```

```

azat@azatai:~$ ls ~/Documents/
Note
azat@azatai:~$ ls /home/azat/Documents/
Note
azat@azatai:~$ ls ../../
bin  dev  home  lib32  libx32  media  opt  root  sbin  sys  usr
boot  etc  lib  lib64  lost+found  mnt  proc  run  srv  tmp  var
azat@azatai:~$ ls /
bin  dev  home  lib32  libx32  media  opt  root  sbin  sys  usr
boot  etc  lib  lib64  lost+found  mnt  proc  run  srv  tmp  var

```

## cd Change Directories - ie. move to another directory.

syntax: **cd [location]**

If you run the command **cd** without any arguments then it will always take you back to your **home** directory.

```

azat@azatai:~$ pwd
/home/azat
azat@azatai:~$ cd Documents/
azat@azatai:~/Documents$ ls
Note
azat@azatai:~/Documents$ cd /
azat@azatai:/$ pwd
/
azat@azatai:/$ cd ~/Documents/
azat@azatai:~/Documents$ ls
Note
azat@azatai:~/Documents$ cd ../../
azat@azatai:/home$ ls
azat  lost+found
azat@azatai:/home$ cd .
azat@azatai:/home$ ls
azat  lost+found
azat@azatai:/home$

```

### Tab Completion

The command line has a nice little mechanism to help us in this respect. It's called **Tab Completion**.

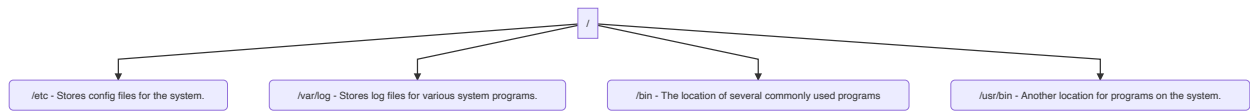
When you start typing a path (anywhere on the command line, you're not just limited to certain commands) you may hit the Tab key on your keyboard at any time which will invoke an auto complete action.

try this:

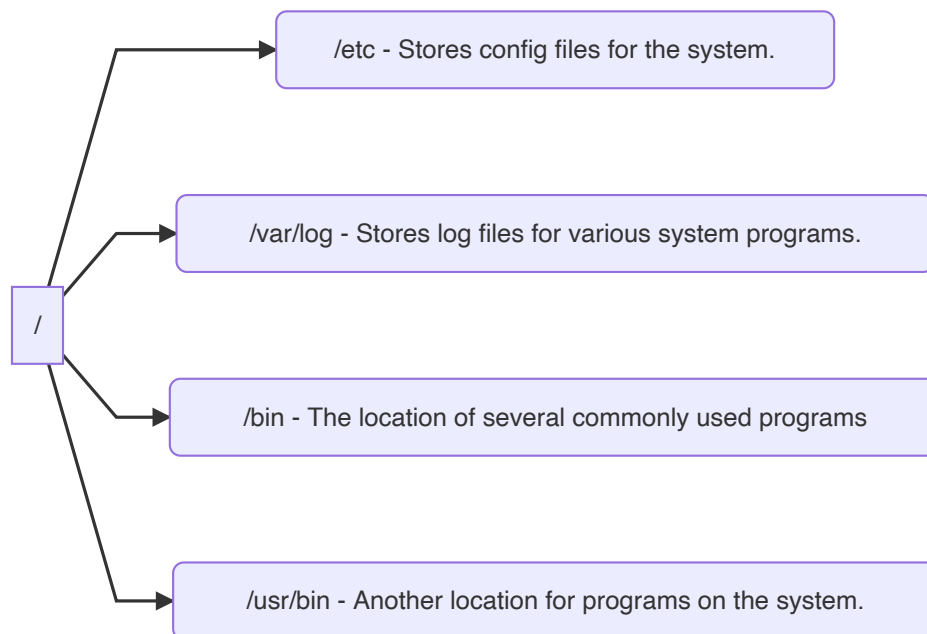
```
cd /hTab/<beginning of your username>Tab
```

## Activities

- Write a directory tree (path tree) of your operating system (wishing that you are using a Unix/Linux like OS, or a Mac OS)
- What those these locations on your OS stores? (Take at least 4 examples)
- Now go to your home directory using 4 different places and methods.
- Make sure you are using Tab Completion when typing out your paths too.



just in case you can't see the chart clearly:



## Files

### Everything is a File

Okay, the first thing we need to appreciate with Linux is that under the hood, everything is actually a file. A text file is a file, a directory is a file, your keyboard is a file (one that the system reads from only), your monitor is a file (one that the system writes to only) etc. To begin with, this won't affect what we do too much but keep it in mind as it helps with understanding the behavior of Linux as we manage files and directories.

### Linux is an Extensionless System

This one can sometimes be hard to get your head around but as you work through the sections it will start to make more sense. A file extension is normally a set of 2 - 4 characters after a full stop at the end of a file, which denotes what type of file it is. The following are common extensions:

- file.exe - an executable file, or program.(Windows)



- file.txt - a plain text file.
- file.png, file.gif, file.jpg - an image.
- file.azt - an azt program source code file.(Can be readed and writed via azt command).
- file.aztx - an azt program executable file.
- file.pyzt - an azt module python wrapper, can be imported into python and used directly.
- file.jszt - an azt module javascript wrapper, can be imported into javascript and used directly.
- file.czt - an azt module c/c++ wrapper, can be imported into javascript and used directly.
- file.azd - azt programming language serialized data file, you can compress any object or any kind of data to .azd format if you like.

In other systems such as **Windows** the extension is important and the **system uses it to determine** what **type** of file it is. Under **Linux** the system actually **ignores** the extension and looks inside the file to determine what type of file it is.

So for instance I could have a file myself.png which is a picture of me. I could rename the file to myself.txt or just myself and Linux would still happily treat the file as an image file. As such it can sometimes be hard to know for certain what type of file a particular file is. Luckily there is a command called **file** which we can use to find this out.

**file [path] -obtain information about what type of file a file or directory is.**

```
azat@azatai:~$ ls .
azat.txt  Desktop    Downloads  Music      Public     Videos
azt       Documents  Edraw      Pictures   Templates
azat@azatai:~$ file azat.txt
azat.txt: ASCII text
azat@azatai:~$ file azt/
azt/: directory
```

Whenever we specify a file or directory on the command line it is actually a path. Also because directories (as mentioned above) are actually just a special type of file, it would be more accurate to say that a path is a means to get to a particular location in the system and that location is a file.

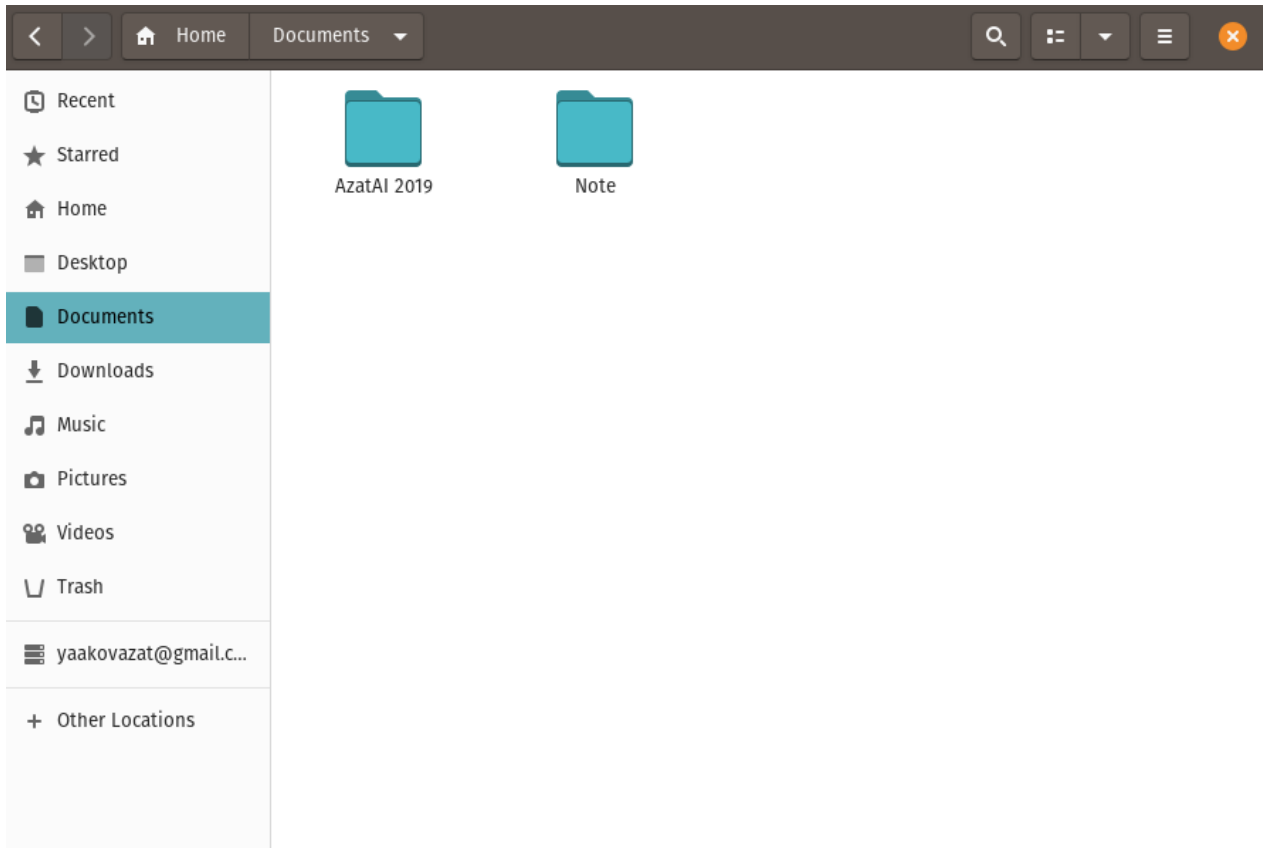
## Linux is Case Sensitive

```
azat@azatai:~$ cd documents
bash: cd: documents: No such file or directory
azat@azatai:~$ cd Documents
azat@azatai:~/Documents$ pwd
/home/azat/Documents
```

Very important to know that, **Linux** is **case sensitive**, while other operating systems like **Windows and Mac OS** are **case insensitive**.

## Spaces in names

ok, Now we have a small problem to solve now, I have just created a folder(or directory by the way) named `AzatAI 2019` and when I want to enter the directory with `cd` something happens:



```
azat@azatai:~/Documents$ pwd
/home/azat/Documents
azat@azatai:~/Documents$ cd AzatAI 2019
bash: cd: too many arguments
azat@azatai:~/Documents$
```

What just happened?

As you would remember, a space on the command line is how we separate items. So what does the computer think is:

```
cd AzatAI(the folder to move) 2019(another command)
```

and, we know that `cd` only receives a path as a single argument, and that's why that happened.

There are two ways to go about this, either way is just as valid.

- S1: using the `quotes` as `' '` or `" "`

```
azat@azatai:~/Documents$ cd 'AzatAI 2019'
azat@azatai:~/Documents/AzatAI 2019$ pwd
/home/azat/Documents/AzatAI 2019
azat@azatai:~/Documents/AzatAI 2019$ cd ..
azat@azatai:~/Documents$ cd "AzatAI 2019"
azat@azatai:~/Documents/AzatAI 2019$ pwd
/home/azat/Documents/AzatAI 2019
```

- S2: using escape characters(`\`).

```
azat@azatai:~/Documents$ cd AzatAI\ 2019
azat@azatai:~/Documents/AzatAI 2019$ pwd
/home/azat/Documents/AzatAI 2019
azat@azatai:~/Documents/AzatAI 2019$
```

Another method is to use what is called an **escape character**, which is a backslash (`\`). What the backslash does is escape (or nullify) the special meaning of the next character.

In the previous section we learnt about something called **Tab Completion**. If you use that before encountering the space in the directory name then the terminal will **automatically escape any spaces** in the name for you.

```
azat@azatai:~/Documents$
```

I

1

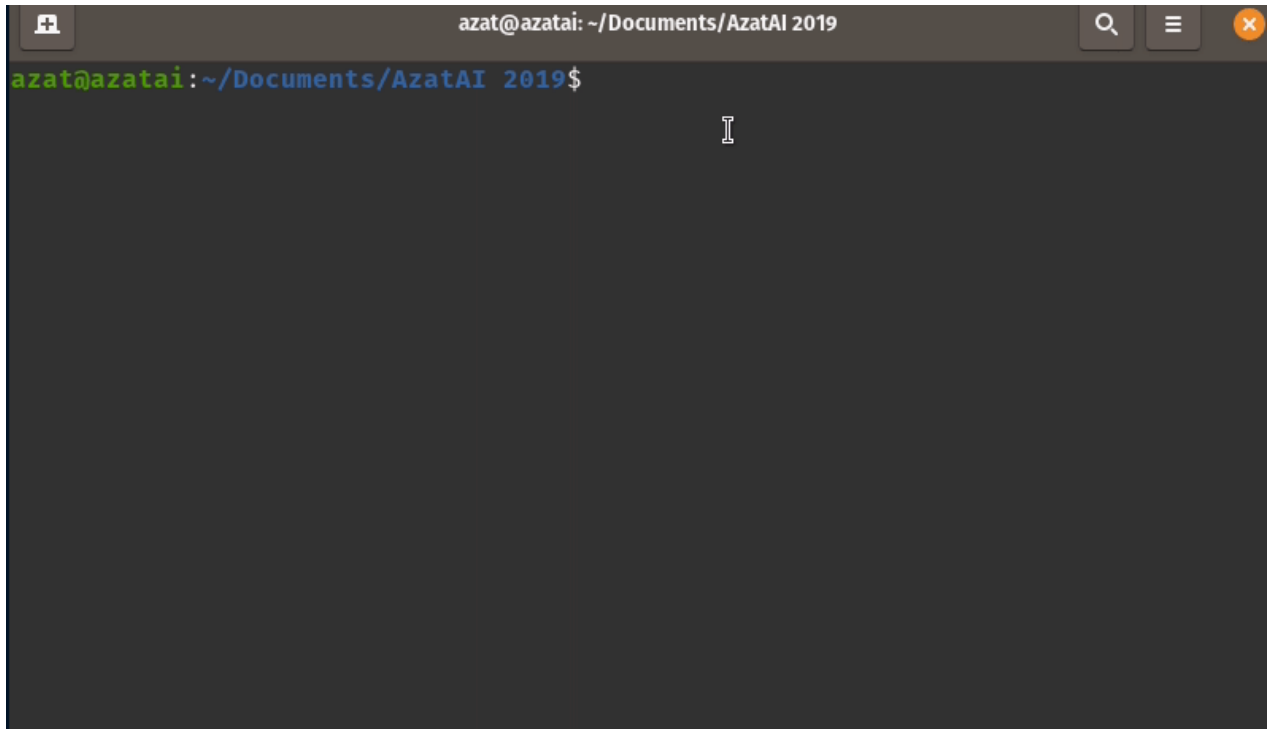
## Hidden files and directories.

If the file or directory's name begins with a `.` (full stop) then it is considered to be hidden in Unix/Linux. To make a file or directory hidden all you need to do is create the file or directory with its name beginning with a `.` or rename it to be as such. The command **ls** which we have seen in the previous section will not list hidden files and directories by default. We may modify it by including the command line option **-a** so that it does show hidden files and directories.

**ls -a** List the contents of a directory, including hidden files.

```
azat@azatai:~/Documents/AzatAI 2019$ ls
azat.ai
azat@azatai:~/Documents/AzatAI 2019$ ls -a
.  ..  .azatai  azat.ai
azat@azatai:~/Documents/AzatAI 2019$
```

or view the short video to got understood:



## Summary

- **Everything is a file under Linux**

Even directories.

- **Linux is an extensionless system**

Files can have any extension they like or none at all.

- **Linux is case sensitive**

Beware of silly typos.

## Activities

Right, now let's put this stuff into practice. Have a go at the following:

- Try running the command **file** giving it a few different entries. Make sure you use a variety of absolute and relative paths when doing this.
- Now issue a command that will list the contents of your home directory including hidden files and directories.

## Manual Pages

---

# Introduction

You know what, remembering all the things in your mind seems to be cool but very hard, at least for me. Linux offers too much commands so that we are almost ready to forget them a lot. However, computer can remember and note a lot, so fortunately there is a way to help us though, we call it `Manual pages` or `man` for short.

## What are they?

Ok, for short they are just documentations of the commands. Cool ya ? What they do then ? Generally they do the things below for each command available on your system:

1. what they do.
2. how you run them.
3. what command line arguments they receive.

You invoke the manual pages with the following command:

```
man <command to lookup>
```

for example:

```
azat@azatai:~$ man ls
```

```
LS(1)                                User Commands                                LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILES (the current directory by default).
    Sort entries alphabetically if none of -cftuvSUX nor --sort is speci-
    fied.

    Mandatory arguments to long options are mandatory for short options
    too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        Manual page ls(1) line 1 (press h for help or q to quit)
```

NAME - what this command is and what that command do. (command short description)

SYNOPSIS - how this command should be run. `[]` means that is optional.

DESCRIPTION - more detailed description about the command. Below the description will always be a list of all the command line options that are available for the command.

**To exit the man pages press 'q' for quit.**

## Searching

You know what you want but don't know which command to use? Search it! There is two way to search on manuals.

1. Search from all the manuals.

```
man -k <search term>
```

for example:

```
azat@azatai:~/Documents/AzatAI 2019$ man -k list
IO::Async::Listener (3pm) - listen on network sockets for incoming
connections
acl (5) - Access Control Lists
acpi_listen (8) - ACPI event listener
add-apt-repository (1) - Adds a repository into the /etc/apt/sources.list
or...
add-shell (8) - add shells to the list of valid login shells
Algorithm::Diff (3pm) - Compute `intelligent' differences between two
files ...
Algorithm::DiffOld (3pm) - Compute `intelligent' differences between two
fil...
american-english (5) - a list of English words
appres (1) - list X application resource database
apt-add-repository (1) - Adds a repository into the /etc/apt/sources.list
or...
argz (3) - functions to handle an argz list
argz_add (3) - functions to handle an argz list
argz_add_sep (3) - functions to handle an argz list
argz_append (3) - functions to handle an argz list
argz_count (3) - functions to handle an argz list
argz_create (3) - functions to handle an argz list
argz_create_sep (3) - functions to handle an argz list
argz_delete (3) - functions to handle an argz list
argz_extract (3) - functions to handle an argz list
```

This will search the keyword you entered (or query question) from all the manuals available, and print out the results in alphabetical order.

2. Search within a manual.

press forward slash `/` within a manual page and enter the keyword(query) you want to search:

```
ls - list directory contents

SYNOPSIS
ls [OPTION]... [FILE]...

DESCRIPTION
List information about the FILES (the current directory by
default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is
speci-
fied.

Mandatory arguments to long options are mandatory for short
options
too.

-a, --all
do not ignore entries starting with .

-A, --almost-all
do not list implied . and ..

--author

/list
```

## Long and short options

There are two different options for us as you can find above, one starts with `-` is called short hand options and another one called long hand options that starts with `--`. Short hand option `-a` do the same thing as the long hand option `--all`. The long hand options are much more human readable and easy to remember while the short ones easy to type.

## Summary

### Staff we learned

```
man <command>
```

Look up the manual page for a particular command.

```
man -k <search term>
```

Do a keyword search for all manual pages containing the given search term.

```
/<term>
```

Within a manual page, perform a search for 'term'

n

After performing a search within a manual page, select the next found item.

## Activities

1. look up the man page of `cd` command.
2. search for 'change' with different two ways from man page(s).

## File manipulation

---

### `mkdir` Making a directory

Linux organizes its file system in a hierarchical way. And as you remembered, everything is a file in Linux. We create folders, especially to structure and organize our datas stored. Develop the habit of organising your stuff into an elegant file structure now and you will thank yourself for years to come.

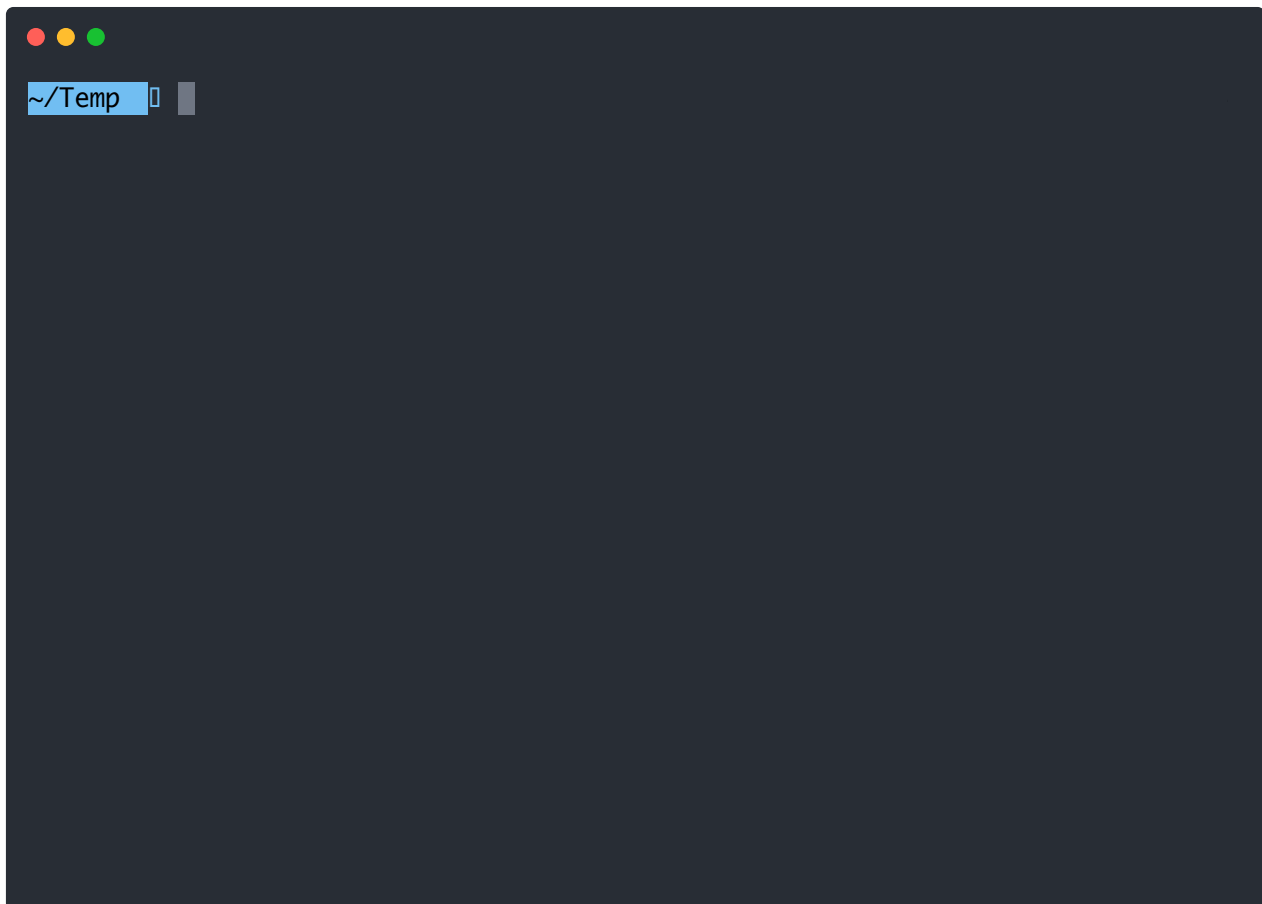
Creating a directory is quite easy, we `mkdir` command for this.

```
mkdir [options] <Directory>
```

```
~/Temp  pwd
/Users/azat/Temp
~/Temp  ls
~/Temp  mkdir temp
~/Temp  ls
temp
```

☐ Update to dynamic gif, rather than a static image.





However, to better understanding, I think now you can try to do these commands and see what is happened?

```
mkdir /home/azat/foo
mkdir ./blah
mkdir ../dir1
mkdir ~/linuxtutorialwork/dir2
```

There are a few useful options available for **mkdir**. We can easily find them by viewing `mkdir`'s manual.

```
MKDIR(1)                                BSD General Commands Manual                                MKDIR(1)

NAME
    mkdir -- make directories

SYNOPSIS
    mkdir [-pv] [-m mode] directory_name ...

DESCRIPTION
    The mkdir utility creates the directories named as operands, in the order
    specified, using mode
    rwxrwxrwx (0777) as modified by the current umask(2).
```

The options are as follows:

**-m mode**  
Set the file permission bits of the final created directory to the specified mode. The mode argument can be in any of the formats specified to the `chmod(1)` command. If a symbolic mode is specified, the operation characters `+` and `-` are interpreted relative to an initial mode of `a=rwx`.

:

Among them, two of them are popular are used often:

**`mkdir [-p]` make parent directories if needed**

Observe the code blow:

```
azat@pop-os:~/Temp$
```

I

1

```
~/Temp mkdir temp/foo  
mkdir: temp: No such file or directory
```

No such file or directory

Why?

If we create a sub directory with `mkdir`, `mkdir` expects that we already have that parent directory and the error appears if not.

The way to solve this problem is by using a short hand option `-p`, or `--parents` that will create each necessary parent folder for us.

-p, --p stands for parents.

## **mkdir [-v] print a message for each created directory**

**-v** or **--verbose** mode prints a message for each created directory.

```
azat@pop-os: ~/Temp$
```

I

1

Another thing you should notice here is that, we can write different short hand options together after a single hyphen **-**.

## **rmdir Removing a Directory**

The command to remove a directory is **rmdir**, short for remove directory.

You should be careful to do this because you cannot undo this!!!

```
rmdir [options] <Directory>
```

Two things to note. Firstly, **rmdir** supports the -v and -p options similar to **mkdir**. Secondly, a directory must be empty before it may be removed (later on we'll see a way to get around this).

```
azat@pop-os:~/Temp$  
  
1
```

## **touch** Creating a Blank File

If we **touch** a file and it does not exist, the command will do us a favor and automatically create it for us. Actually **touch** is not for creating a new file, but we **touch** a nonexistent file, **touch** will creates one.

What dose touch do actually is : if we touch a file and it does not exist, the command will do us a favor and automatically create it for us.

```
azat@pop-os:~/Temp$
```

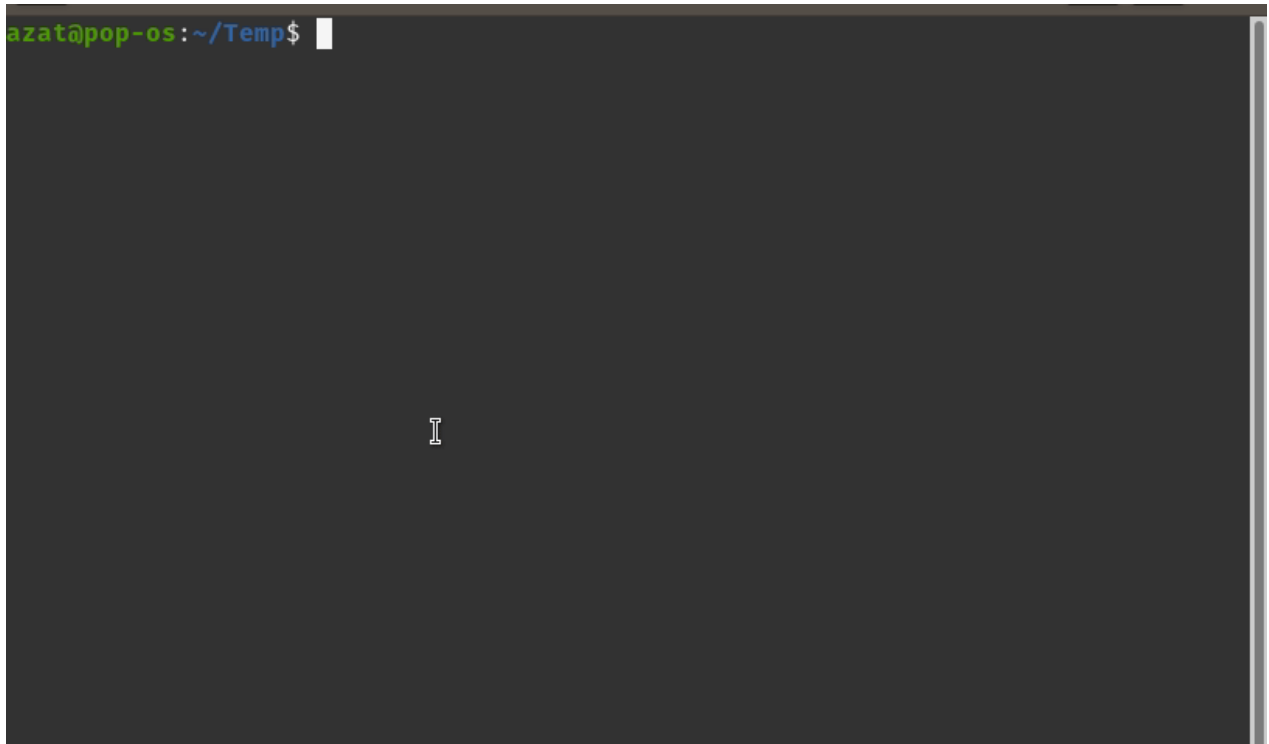
## **cp** Copying a File or Directory

Often before changing something, we may wish to create a duplicate so that if something goes wrong we can easily revert back to the original. The command we use for this is `cp` which stands for copy.

```
cp [options] <source> <destination>
```

Note that both the source and destination are paths. This means we may refer to them using both absolute and relative paths.

Using the `-r` option, which stands for recursive, we may copy directories. Recursive means that we want to look at a directory and all files and directories within it, and for subdirectories, go into them and do the same thing and keep doing this.



## `mv` Moving a File or Directory

To move a file we use the command `mv` which is short for move. It operates in a similar way to `cp`. One slight advantage is that we can move directories without having to provide the `-r` option.

```
mv [options] <source> <destination>
```

```
azat@pop-os:~/Temp$
```

I

1

## **mv** Renaming Files and Directories

Normally mv will be used to move a file or directory into a new directory, note that, we may provide a new name for the file or directory and as part of the move it will also rename it.

In programming, actually you gonna see a lot of such magic then...

```
azat@pop-os:~/Temp$
```

I

## **rm** Removing a File (and non empty Directories)

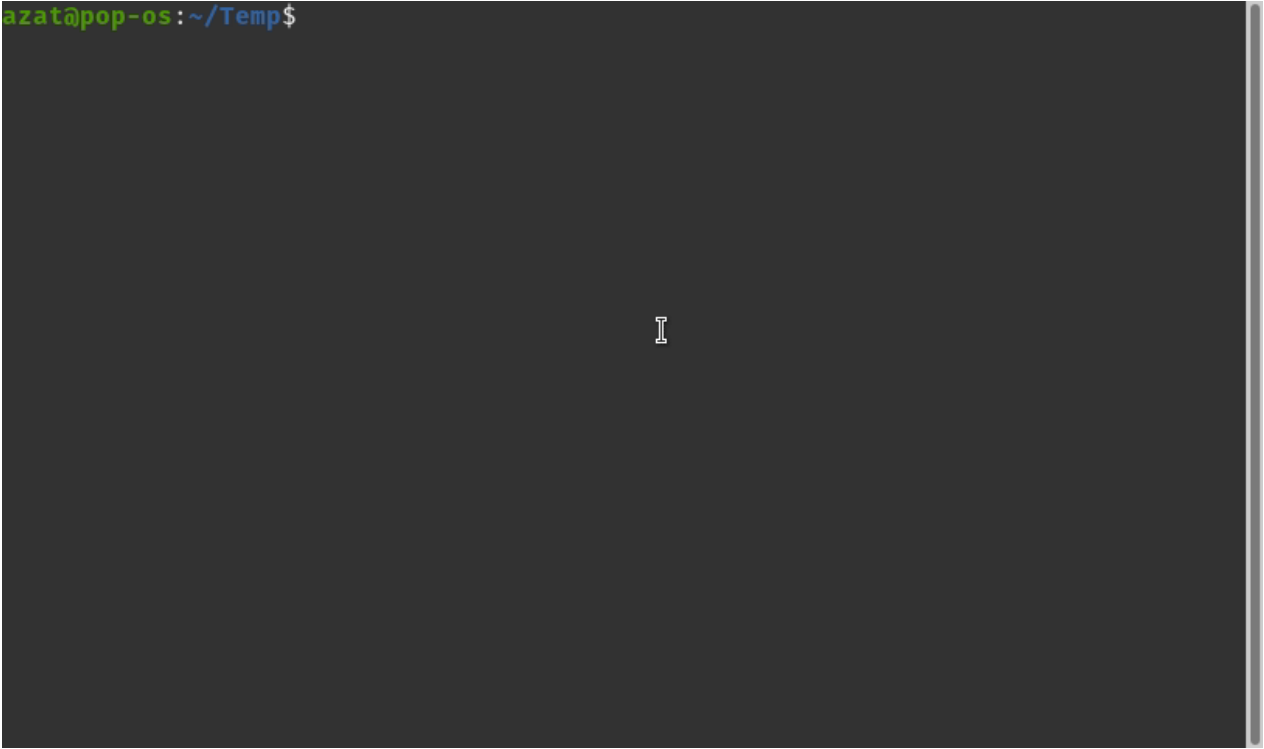
The command to remove or delete a file is **rm** which stands for remove.

```
rm [options] <file>
```

Try:

```
mkdir foofoo  
ls  
rm foofoo
```

```
azat@pop-os:~/Temp$
```



Not work, right? Then Why and How?

WHY: Remember, the `rm` command is used to remove a file and non empty directories.

## `rm -r` Removing non empty Directories

Like several other commands introduced in this section, `rm` has several options that alter it's behaviour. We can see `rm` options through `man` always:

```
man rm
```

When `rm` is run with the `-r` option it allows us to remove directories and all files and directories contained within.

#### OPTIONS

Remove (unlink) the FILE(s).

**-f, --force**

ignore nonexistent files and arguments, never prompt

**-i** prompt before every removal

**-I** prompt once before removing more than three files, or when removing recursively; less intrusive than **-i**, while still giving protection against most mistakes

```
azat@pop-os:~/Temp$
```

There are something to explain:

1. `rm -r` removes directories and all of the files inside the directory.
2. `rm -rf` by adding `-f` option, we can remove anything by force, (never prompt)  
!!! You should be careful to do this if you are a new comer to the Linux !!!
3. `rm -ri` is very helpful when you are not quite sure what you are deleting.

You should use `rm -ri` a lot before you are quite experienced in Linux!

## Summary



**mkdir**

Make Directory - ie. Create a directory.

**rmdir**

Remove Directory - ie. Delete a directory.

**touch**

Create a blank file.

**cp**

Copy - ie. Copy a file or directory.

**mv**

Move - ie. Move a file or directory (can also be used to rename).

**rm**

Remove - ie. Delete a file.

## Activities

1. Change directory to your home directory.
2. Create a directory named `AzatAi`.
3. The directory name was not correct, now rename it to `AzatAI`.
4. Now create two different directory inside the `AzatAI` directory, name them `foo` and `foofoo`
5. Inside the `foo` create a new file `<your name>.txt`, write your `TODOs` inside the file.
6. Copy the file to `AzatAI/backups`.
7. Create a file `done.txt` inside the `foofoo` folder, and write down the tasks that you finished today.
8. Backup all the `foo` and `foofoo` folders to `Desktop/Backup0`.
9. Delete all the files and directories within the `AzatAI` directory(Including AzatAI).

Note: You should use interactive mode while removing.

## Vi Text Editor

### Introduction

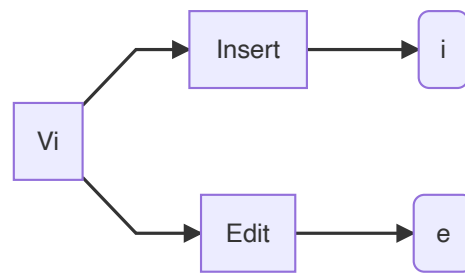
I believe that you have used Microsoft Word, Notepad, or maybe some other text editing tools like Atom, SublimeText, maybe then visual studio code. All of them are nice text editors to use in **GUI**. However, Vi is a text editor that is most likely very different to any editor you have used before.

Vi is a very powerful tool. Vi is preinstalled almost every Linux/Unix distributions. It's the powerful tool you use to create, edit, update and search text files in terminal. Mostly linux servers do not have a graphical user interface (because they don't need one) and you can only use terminal based text editors, you will then find, how strong and powerful Vi is!

### A Command Line Editor

Vi is a command line editor and everything in Vi is done via the keyboard (you do not need the GUI and you do not need the mouse).

There are two modes in Vi,



In `insert` mode, you may input or enter content into the file. In edit mode, you can move around the file, perform actions such as deleting, copying, search and replace, saving etc.

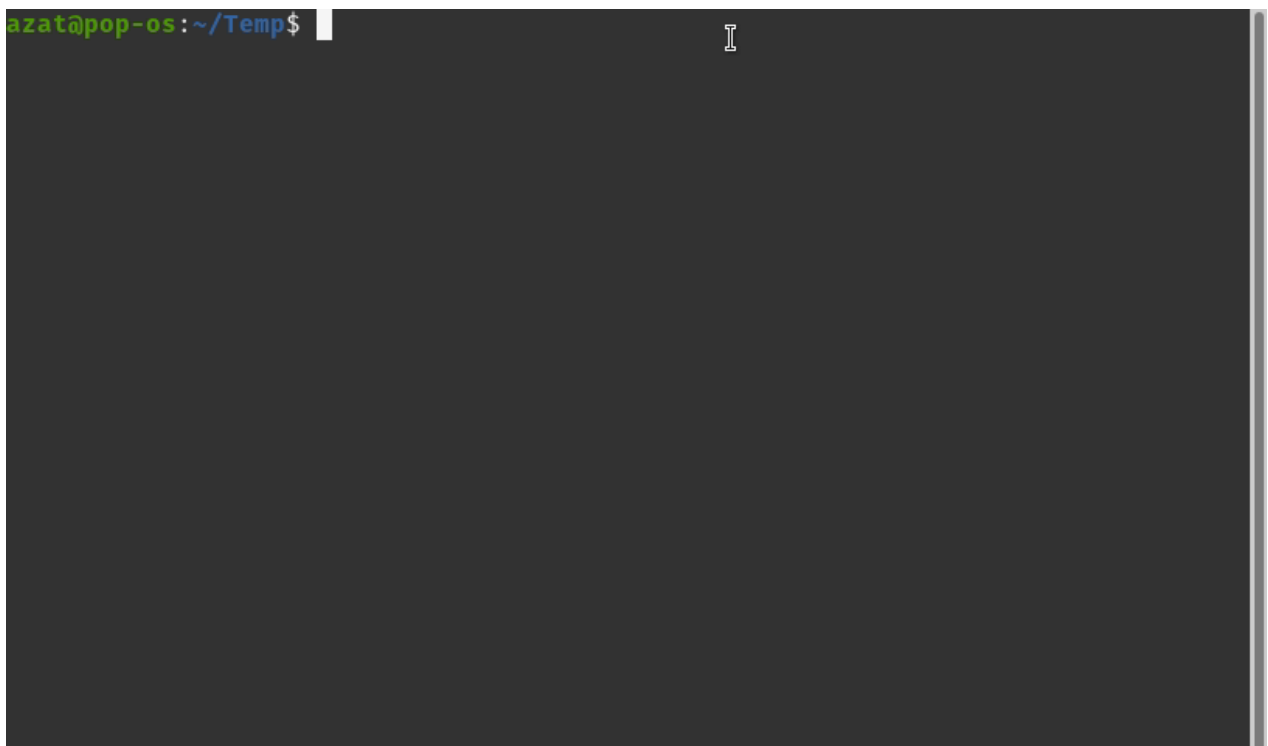
## `vi` open file

When we run `vi` we normally issue it with a single command line argument which is the file you would like to edit. Also remember that when we specify the file it can be with either an absolute or relative path.

```
vi <file>
```

Now we'll edit our first file.

When you run this command it opens up the file. If the file does not exist then it will create it for you then open it up. (no need to touch files before editing them) Once you enter `vi` it will look something like this (though depending on what system you are on it may look slightly different).



## Insert mode

You always start off in edit mode so the first thing we are going to do is switch to insert mode by pressing **i**. You can tell when you are in insert mode as the bottom left corner will tell you.

```
azat@pop-os:~/Temp$
```

I

## Saving and Exiting

Notice: Any `command` you give to the Vi, should be typed inside the command mode (edit mode) not inside the insert mode, if so, you should exit the insert mode by typing `ESC` first.

There are several ways to save the current file in Vi:

- `zz` - save and exit
- `q!` - discard all changes(this will lost your changes), exit. (Do not save and exit)
- `:w` - save and exit
- `:wq` - save and exit

Any command starting with a colon (:) requires you to hit `enter` to complete the command.

Entered the insert mode by typing i

Now let's save and exit.

```
"firstfile" [New File]
```

Think what is the difference between `:w` and `:wq` ?

## Other ways to view files

### cat command

`cat` is a useful command in Linux (of course all Linux commands by means are useful ), it allows us to get the file content without entering the file.

```
cat <file>
```

```
azat@pop-os: ~/Temp$
```

```
I
```

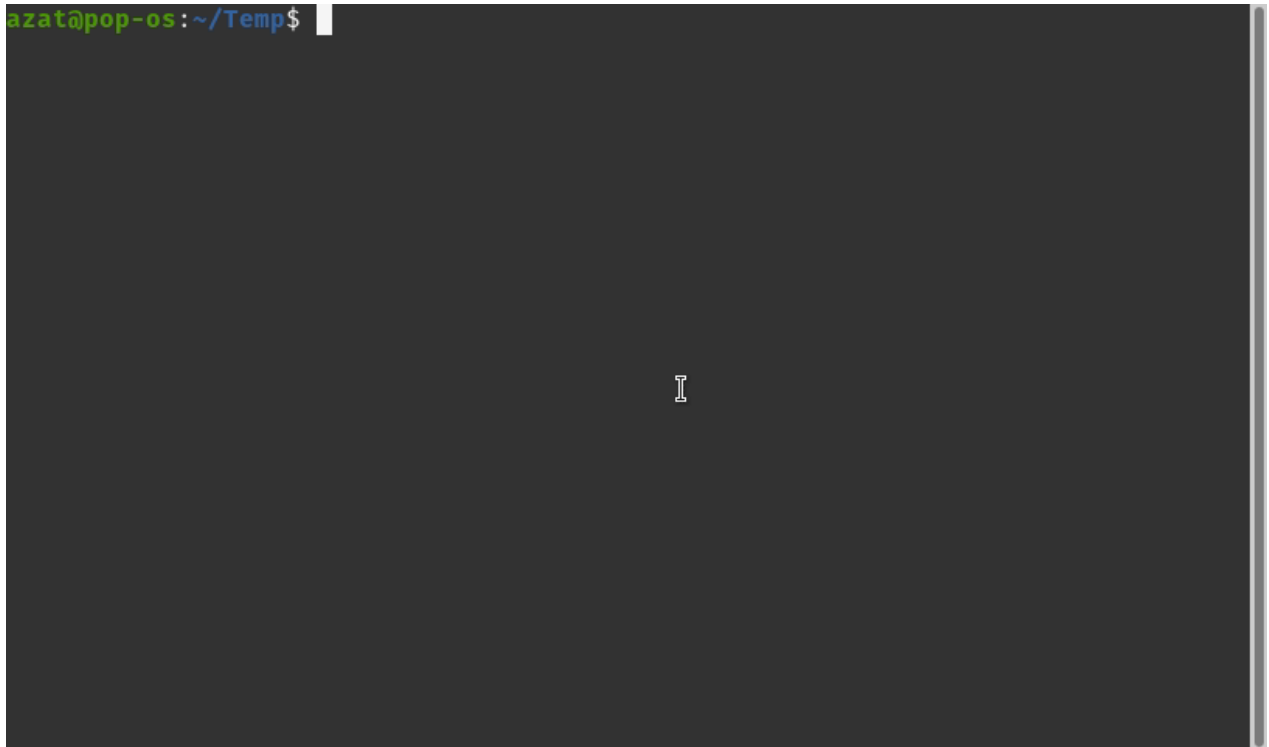
```
1
```

This command is nice when we have a small file to view but if the file is large then most of the content will fly across the screen and we'll only see the last page of content. For larger files there is a better suited command which is **less**.

### **less** command

**less** allows you to move up and down within a file using the arrow keys. You may go forward a whole page using the **SpaceBar** or back a page by pressing **b**. When you are done you can press **q** for quit.

```
less <file>
```



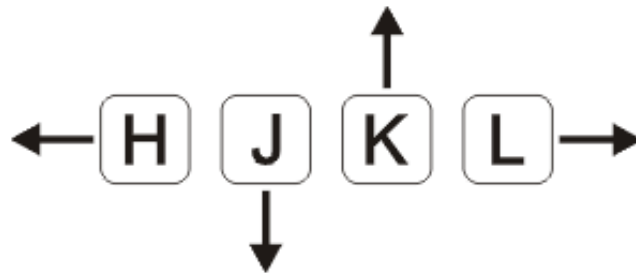
## Navigating a file in Vi

Below are some of the many commands you may enter to move around the file. Have a play with them and see how they work:

- **Arrow keys** - move the cursor around
- **j, k, h, l** - move the cursor down, up, left and right (similar to the arrow keys)
- **^ (caret)** - move cursor to beginning of current line
- **\$** - move cursor to end of the current line
- **nG** - move to the **n**th line (eg 5G moves to 5th line)
- **G** - move to the last line
- **w** - move to the beginning of the next word
- **nw** - move forward n word (eg 2w moves two words forwards)
- **b** - move to the beginning of the previous word
- **nb** - move back n word
- **{** - move backward one paragraph
- **}** - move forward one paragraph

You should practice all of them above and try to remember!!!!

If you type **:set nu** in edit mode within `vi` it will enable line numbers. I find that turning line numbers on makes working with files a lot easier.



```
azata@pop-os: ~/Temp$  
  
I  
  
1
```

**h j k l** - left, down, up, right

Try to navigate by using `h(left)j(down)k(up)l(right)`

**^ \$** - the beginning and ending of the line

Try to navigate to the beginning and end of line with `^` and `$`

```
azat@pop-os:~/Temp$ vi azt.md
azat@pop-os:~/Temp$
```

## **nG** -- move to the nth line

Try to move to the nth line by **nG**

```
Project description
Intro for ~azt Programming language
~azt programming language
About azt
~azt is an intelligent hybrid progressive programming language that makes your d
evelopment more intelligent, more faster and much more easy. It aims to become a
programming language which gives it high-level, object oriented, high performan
ced programming. ~azt programming language is easy to learn as Python, fast enou
gh as C++ and generic as Java.

Installing azt
~azt is an intelligent hybrid progressive programming language that take advanta
ges of many other programming languages like C, C++, Java, Python and Javascript
. Therefore you should have installed all the requirements first:

a C compiler(for C and C++ base of azt).
Linux: The GNU C Compiler (gcc) is usually present, or easily available through
the package system. On Ubuntu or Debian, for instance, the command sudo apt-get
install build-essential will fetch everything you need.
Mac OS: The best practice is to install Xcode command line tools. type xcode-sel
ect --install and you follow the instructions. Or simply, you can install Xcode
while it automatically installs the Xcode command line tools for you.

"azt.md" 39 lines, 3399 characters
```

## **G** - move to the last line

Move to the Last line of the document by:

Last line: **G**

```
Linux: Simply run "sudo apt install python3" and I'm sure you will get Python3 v
ery soon, just Try, Why not?
Mac OS: The best way to install python3 on mac os is defenetly through brew, ins
tall brew if you don't have(how did you able to breath without brew?!) and then
try: brew install python3.
Windows: Dear me! Just install Windows Subsystem for Linux and then follow Linux
instructions for the rest, ok?
Node.js
Both for Linux and Mac OS: Download and install the LTS versions of node.js spec
ified for your operating system.
You'll finally able to install azt on your system without a trouble. Please be
noticed that, azt supports only the Python 3 version, and that's the version yo
u should install on your OS, if necessary run azt within a 'virtualenv'.

The simplest way of installing azt is by using pip:

sudo pip install azt
NOTE: You should use sudo command to run the code as azt needs administrator pri
vilage.

Alternatively, you can also try the shell command to install azt, download the l
atest version of azt and unzip the file, enter the azt installer folder and then
simply run install.sh by:
```

## **w** - move to the beginning of next word

```
Alternatively, you can also try the shell command to install azt, download the l
atest version of azt and unzip the file, enter the azt installer folder and then
simply run install.sh by:

sudo ./install.sh
NOTE: You should use sudo command to run the code as azt needs administrator pri
vilage.

Hello World in azt
A simple Hello World! program written with azt would be like this:

# hello_world.azt
print("Hello World!")
You may confused here: Is it a Python code ? Yes, and NO. However many programmi
ng languages use the same way for Hello World. But, one important message you sh
ould know here is that, azt programming language is itself a hybrid programming
language that based on many other programming languages, that's the nature of az
t programming language to have a similar structure of code as x programming lang
uage but extremly faster than the x programming language.

You can test it simply with timing tools, have a try!
~
```

## **nw** - move forward n word (words)



Alternatively, you can also try the shell command to install azt, download the latest version of azt and unzip the file, enter the azt installer folder and then simply run install.sh by:

```
sudo ./install.sh
```

NOTE: You should use sudo command to run the code as azt needs administrator privilege.

Hello World in azt

A simple Hello World! program written with azt would be like this:

```
# hello_world.azt
```

```
print("Hello World!")
```

You may confused here: Is it a Python code ? Yes, and NO. However many programming languages use the same way for Hello World. But, one important message you should know here is that, azt programming language is itself a hybrid programming language that based on many other programming languages, that's the nature of azt programming language to have a similar structure of code as x programming language but extremely faster than the x programming language.

You can test it simply with timing tools, have a try!

~

## **b** - move to the beginning of the previous word

Alternatively, you can also try the shell command to install azt, download the latest version of azt and unzip the file, enter the azt installer folder and then simply run install.sh by:

```
sudo ./install.sh
```

NOTE: You should use sudo command to run the code as azt needs administrator privilege.

Hello World in azt

A simple Hello World! program written with azt would be like this:

```
# hello_world.azt
```

```
print("Hello World!")
```

You may confused here: Is it a Python code ? Yes, and NO. However many programming languages use the same way for Hello World. But, one important message you should know here is that, azt programming language is itself a hybrid programming language that based on many other programming languages, that's the nature of azt programming language to have a similar structure of code as x programming language but extremely faster than the x programming language.

You can test it simply with timing tools, have a try!

~

## **nb** - move back n word(s)

Alternatively, you can also try the shell command to install azt, download the latest version of azt and unzip the file, enter the azt installer folder and then simply run install.sh by:

```
sudo ./install.sh
```

NOTE: You should use sudo command to run the code as azt needs administrator privilege.

Hello World in azt

A simple Hello World! program written with azt would be like this:

```
# hello_world.azt
```

```
print("Hello World!")
```

You may confused here: Is it a Python code ? Yes, and NO. However many programming languages use the same way for Hello World. But, one important message you should know here is that, azt programming language is itself a hybrid programming language that based on many other programming languages, that's the nature of azt programming language to have a similar structure of code as x programming language but extremely faster than the x programming language.

You can test it simply with timing tools, have a try!

## **{}** - move forward/back one paragraph

Alternatively, you can also try the shell command to install azt, download the latest version of azt and unzip the file, enter the azt installer folder and then simply run install.sh by:

```
sudo ./install.sh
```

NOTE: You should use sudo command to run the code as azt needs administrator privilege.

Hello World in azt

A simple Hello World! program written with azt would be like this:

```
# hello_world.azt
```

```
print("Hello World!")
```

You may confused here: Is it a Python code ? Yes, and NO. However many programming languages use the same way for Hello World. But, one important message you should know here is that, azt programming language is itself a hybrid programming language that based on many other programming languages, that's the nature of azt programming language to have a similar structure of code as x programming language but extremely faster than the x programming language.

You can test it simply with timing tools, have a try!

### Tip

If you type **:set nu** in edit mode within vi it will enable line numbers. I find that turning line numbers on makes working with files a lot easier.

```

26
27 Alternatively, you can also try the shell command to install azt, downlo
ad the latest version of azt and unzip the file, enter the azt installer folder
and then simply run install.sh by:
28
29 sudo ./install.sh
30 NOTE: You should use sudo command to run the code as azt needs administr
ator privilege.
31
32 Hello World in azt
33 A simple Hello World! program written with azt would be like this:
34
35 # hello_world.azt
36 print("Hello World!")
37 You may confused here: Is it a Python code ? Yes, and NO. However many p
rogramming languages use the same way for Hello World. But, one important messag
e you should know here is that, azt programming language is itself a hybrid prog
ramming language that based on many other programming languages, that's the natu
re of azt programming language to have a similar structure of code as x programm
ing language but extremly faster than the x programming language.
38
39 sssdddddffffggggghhhhhou can test it simply with timing tools, have a try
!
```

## Deleting content

Below are some of the many ways in which we may delete content within vi.

- **x** - delete a single character
- **nx** - delete n characters (eg 5x deletes five characters)
- **dd** - delete the current line
- **dn** - d followed by a movement command. Delete to where the movement command would have taken you. (eg d5w means delete 5 words)

### **x** - delete a single character

```

26
27 Alternatively, you can also try the shell command to install azt, downlo
ad the latest version of azt and unzip the file, enter the azt installer folder
and then simply run install.sh by:
28
29 sudo ./install.sh
30 NOTE: You should use sudo command to run the code as azt needs administr
ator privilege.
31
32 Hello World in azt
33 A simple Hello World! program written with azt would be like this:
34
35 # hello_world.azt
36 print("Hello World!")
37 You may confused here: Is it a Python code ? Yes, and NO. However many p
rogramming languages use the same way for Hello World. But, one important messag
e you should know here is that, azt programming language is itself a hybrid prog
ramming language that based on many other programming languages, that's the natu
re of azt programming language to have a similar structure of code as x programm
ing language but extremly faster than the x programming language.
38
39 sssdddddffffggggghhhhhou can test it simply with timing tools, have a try
!
```

## **nX** - delete n characters

```
26
27 Alternatively, you can also try the shell command to install azt, downlo
ad the latest version of azt and unzip the file, enter the azt installer folder
and then simply run install.sh by:
28
29 sudo ./install.sh
30 NOTE: You should use sudo command to run the code as azt needs administr
ator privilege.
31
32 o
33
34
35 # hello_world.azt
36 print("Hello World!")
37 You may confused here: Is it a Python code ? Yes, and NO. However many p
rogramming languages use the same way for Hello World. But, one important messag
e you should know here is that, azt programming language is itself a hybrid prog
ramming language that based on many other programming languages, that's the natu
re of azt programming language to have a similar structure of code as x programm
ing language but extremply faster than the x programming language.
38
39 h
~
```

## **dd** - delete current line

```
azat@pop-os:~/Temp$
```

## **dnw** - delete n word forward

```

Intro for ~azt Programming language
~azt programming language
About azt
~azt is an intelligent hybrid progressive programming language that makes your d
evelopment more intelligent, more faster and much more easy. It aims to become a
programming language which gives it high-level, object oriented, high performan
ced programming. ~azt programming language is easy to learn as Python, fast enou
gh as C++ and generic as Java.

Installing azt
~azt is an intelligent hybrid progrIessive programming language that take advanta
ges of many other programming languages like C, C++, Java, Python and Javascript
. Therefore you should have installed all the requirements first:

a C compiler(for C and C++ base of azt).
Linux: The GNU C Compiler (gcc) is usually present, or easily available through
the package system. On Ubuntu or Debian, for instance, the command sudo apt-get
install build-essential will fetch everything you need.
Mac OS: The best practice is to install Xcode command line tools. type xcode-sel
ect --install and you follow the instructions. Or simply, you can install Xcode
while it automatically installs the Xcode command line tools for you.
@
@

```

## **dnb** - delete n word backward

```

Programming language
~azt programming language
About azt
~azt is an intelligent hybrid progressive programming language that makes your d
evelopment more intelligent, more faster and much more easy. It aims to become a
programming language which gives it high-level, object oriented, high performan
ced programming. ~azt programming language is easy to learn as Python, fast enou
gh as C++ and generic as Java.

Installing azt
~azt is an intelligent hybrid progressive programming language that take advanta
ges of many other programming languages like C, C++, Java, Python and Javascript
. Therefore you should have installed all the requirements first:

a C compiler(for C and C++ base of azt).
Linux: The GNU C Compiler (gcc) is usually present, or easily available through
the package system. On Ubuntu or Debian, for instance, the command sudo apt-get
install build-essential will fetch everything you need.
Mac OS: The best practice is to install Xcode command line tools. type xcode-sel
ect --install and you follow the instructions. Or simply, you can install Xcode
while it automatically installs the Xcode command line tools for you.
@
@

```

## Undoing

Undoing changes in **vi** is fairly easy. It is the character **u**.

- **u** - Undo the last action (you may keep pressing u to keep undoing)
- **U (Note: capital)** - Undo all changes to the current line

```
Project description
Intro for ~azt Programming language
~azt programming language
About azt
~azt is an intelligent hybrid progressive programming language that makes your d
evelopment more intelligent, more faster and much more easy. It aims to become a
programming language which gives it high-level, object oriented, high performan
ced programming. ~azt programming language is easy to learn as Python, fast enou
gh as C++ and generic as Java.

Installing azt
~azt is an intelligent hybrid progressive programming language that take advanta
ges of many other programming languages like C, C++, Java, Python and Javascript
. Therefore you should have installed all the requirements first:

a C compiler(for C and C++ base of azt).
Linux: The GNU C Compiler (gcc) is usually present, or easily available through
the package system. On Ubuntu or Debian, for instance, the command sudo apt-get
install build-essential will fetch everything you need.
Mac OS: The best practice is to install Xcode command line tools. type xcode-sel
ect --install and you follow the instructions. Or simply, you can install Xcode
while it automatically installs the Xcode command line tools for you.

"azt.md" 39 lines, 3399 characters
```

## Taking it further

We can now insert content into a file, move around the file, delete content and undo it then save and exit. You've already learned the basic concepts and commands of the VI. However, never stop learning deeper and further, if so, please refer following resources to go further:

- [Basic vi commands](#)
- [vi Editor in UNIX](#)

Of course there are many vi cheat sheets out there too which list all the commands available to you.

If possible, you should learn `vi` commands below ,so that you can increase your work efficiency.

- copy and paste
- search and replace
- buffers
- markers
- ranges
- settings

Have fun and remember to keep at it. `vi` will be painful at first but with practice it will soon become your friend.

## Summary

Staff we learnt:

`vi` - Edit a file.

`cat` - View a file.

`less` - Convenient for viewing large files.

# Important concepts

## No mouse

`vi` is a text editor, that works perfectly on the command line and you should be get used to use it without the graphical interface. That would be a first skill that you will and have to learn to become a qualified programmer.

## Edit commands

There are many edit commands in `vi`, never be tired to try and learn, you should a lot of excersizes and try more resources, remember, practice makes perfect.

## Activities

Let's do little more practice:

- Create a text file, write down your cv inside.
- Save the file and view it both in `cat` and `less`.
- Practice movement in vi with your file.
- Practice deleting the content.
- Discard the changes and exit.

## Linux/Unix wildcard

---

### Introduction

A wildcard is a generic term referring to something that can be substituted for all possibilities.

In computer terms, usually a simple "wildcard" is just a `*` that can match one or more characters, and possibly a `?` that can match any single character.

If you have learnt regular expression before, you may have confused then, because wildcard seems like regular expressions, however, regular expressions are very powerful and much more complex than wildcards and we will catch up later.

### How they look?

Here is the basic set of wildcards:

- `*` - represents **zero or more** characters
- `?` - represents **a single** character
- `[]` - represents **a range of characters**

#### `*` wildcard

`*` - represents zero or more characters

```
azat@pop-os:~/Temp$
```

I

1

As you can find, wild cards are very useful.

```
azat@pop-os:~/Temp$ pwd
```

```
/home/azat/Temp
```

```
azat@pop-os:~/Temp$ ls
```

```
azt.md      blahblah.txt  clear      foofoo  ok.request
```

```
blafiii.txt bob.md        firstfile  ok
```

```
azat@pop-os:~/Temp$ ls b*
```

```
blafiii.txt  blahblah.txt  bob.md
```

```
azat@pop-os:~/Temp$
```

I

1

How about some more examples?



```
azat@pop-os:~/Temp$ pwd
/home/azat/Temp
azat@pop-os:~/Temp$ ls
azt.md      blahblah.txt  clear      foofoo      ok.request
blafiii.txt bob.md        firstfile  ok
azat@pop-os:~/Temp$ ls b*
blafiii.txt  blahblah.txt  bob.md
azat@pop-os:~/Temp$ ls bl*
blafiii.txt  blahblah.txt
azat@pop-os:~/Temp$
```

```
ls *.txt
```

This code will show us all the .txt files.

## ? Wildcard

Now let's introduce the **?** operator. **?** - represents **a single** character. In this example below, we are looking for each file whose second letter is **l**.

```
azat@pop-os:~/Temp$
```

Or how about every file with a three letter extension?

```
azat@pop-os:~/Temp$
```

## [ ] wildcard

Unlike the previous 2 wildcards which specified any character, the range operator allows you to limit to a subset of characters. [ ] - represents **a range of characters**

In this example we are looking for every file whose name either begins with a s or v.

```
azat@pop-os:~/Temp$
```

So for example if we wanted to find every file whose name includes a digit in it we could do the following:

```
azat@pop-os:~/Temp$
```

I

## - not

We may also reverse a range using the caret ( ^ ) which means look for any character which is not one of the following.

```
azat@pop-os:~/Temp$
```

I

## Real life examples

1. Find the file type of every file in a directory.

```
file ./*
```

```
azat@pop-os:~/Temp$
```

I

1

2. Move all files of type either jpg or png (image files) into another directory.

```
mv /*.??g ./images
```

```
azat@pop-os:~/Temp$
```

I

3. Find out the size and modification time of the .bash\_history file in every users home directory.

```
ls -lh /home/*.bash_history
```

`.bash_history` is a file in a typical users home directory that keeps a history of commands the user has entered on the command line. Remember how the `.` means it is a hidden file?

```
azat@pop-os:~/Temp/images$
```



## Activities

Let's play with some patterns.

- A good directory to play with is `/etc` which is a directory containing config files for the system. As a normal user you may view the files but you can't make any changes so we can't do any harm. Do a listing of that directory to see what's there. Then pick various subsets of files and see if you can create a pattern to select only those files.
- Do a listing of `/etc` with only files that contain an extension.
- What about only a 3 letter extension?
- How about files whose name contains an uppercase letter? (hint: `[:upper:]` may be useful here)
- Can you list files whose name is 4 characters long?

```
ls /etc/*.*
```

```
azat@pop-os:~/Temp$
```

I

```
ls -l /etc/*.???
```

```
-rw-r--r-- 1 root root 726 Aug 8 22:46 10-kernel-hardening.conf
-rw-r--r-- 1 root root 257 Aug 8 22:46 10-link-restrictions.conf
-rw-r--r-- 1 root root 1184 Aug 8 22:46 10-magic-sysrq.conf
-rw-r--r-- 1 root root 158 Aug 8 22:46 10-network-security.conf
-rw-r--r-- 1 root root 19 Oct 18 08:24 10-pop-default-settings.conf
-rw-r--r-- 1 root root 1292 Aug 8 22:46 10-ptrace.conf
-rw-r--r-- 1 root root 506 Aug 8 22:46 10-zero-page.conf
lrwxrwxrwx 1 root root 14 Dec 5 23:08 99-sysctl.conf -> ../sysctl.conf
-rw-r--r-- 1 root root 324 Jun 6 2018 protect-links.conf
-rw-r--r-- 1 root root 792 Jun 6 2018 README.sysctl
```

```
/etc/tmpfiles.d:
total 0
```

```
/etc/update-motd.d:
total 20
```

```
-rwxr-xr-x 1 root root 1220 Aug 28 00:31 00-header
-rwxr-xr-x 1 root root 1157 Aug 28 00:31 10-help-text
-rwxr-xr-x 1 root root 4677 Dec 5 20:39 50-motd-news
-rwxr-xr-x 1 root root 299 Dec 7 03:52 91-release-upgrade
```

```
/etc/usb_modeswitch.d:
total 0
```

I

```
azat@pop-os:~/Temp$
```

```
ls -l /etc/*[[:upper:]]*
```

```
azat@pop-os:~/Temp$
```

1

I

```
ls /etc/????
```

```
azat@pop-os:~/Temp$ ls ????
```

```
foo1
```

```
azat@pop-os:~/Temp$ c
```

I

## Permission

---

### Introduction

Permission specify what a particular person may or mayn't do with respect to a file or directory.

As such, permissions are important in creating a secure environment. Luckily, permissions in a linux system are quite easy to work with.

## Linux permissions

Linux permissions dictate 3 things you may do with a file, read, write and execute. They are referred to in Linux by a single letter each.

- **r** read - you may view the content. (read)
- **w** write - you may change the content.
- **x** execute - you may execute (run) the file if it is a program or script.

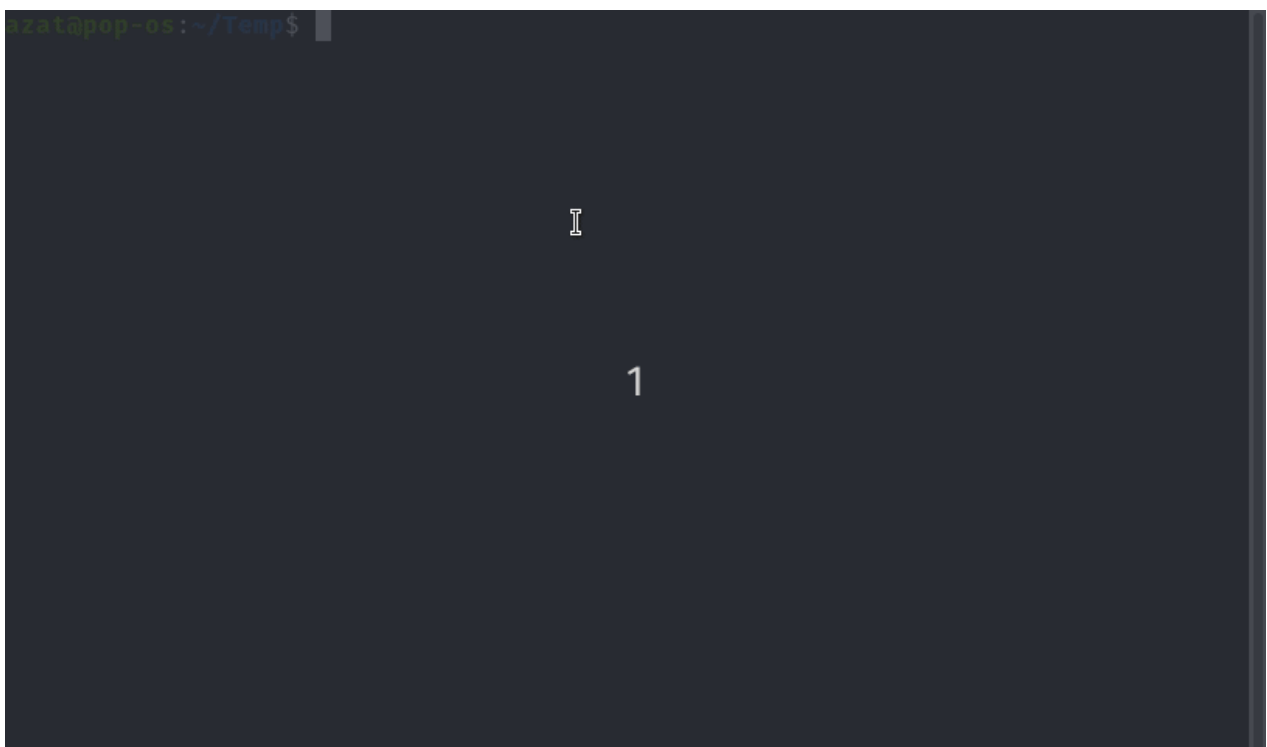
For every file we define 3 sets of people for whom we may specify permissions.

- **owner** - a single person who owns the file. (Generally the one who created the file, but in some way the ownership can be granted to others.)
- **group** - every file belongs to a single group.
- **others** - everyone else who is not in the group or the owner.

### **r** - view(read) permission

To view permissions for a file we use the long listing option for the command ls.

```
ls -l [path]
```



```
azat@pop-os:~/Temp$ ls -l ok
total 4
drwxr-xr-x 2 azat azat 4096 Jan  9 10:19 okk
```



- The first character identifies the file type. If it is a dash ( - ) then it is a normal file. If it is a d then it is a directory. ( `d` here)
- The following 3 characters represent the permissions for the owner. A letter represents the presence of a permission and a dash ( - ) represents the absence of a permission. In this example the owner has all permissions (read, write and execute). ( `rwX` here)
- The following 3 characters represent the permissions for the group. In this example the user has the ability to read and execute, but do not write.( `r-x` here)
- Finally the last 3 characters represent the permissions for others (or everyone else). ( `r-x` here)

## Change permission

To change permissions on a file or directory we use a command called **chmod** It stands for change file mode bits.

```
chmod [permissions] [path]
```

**chmod** has permission arguments that are made up of 3 components

- Who are we changing the permission for? A user? Owner? Group ?
- Are we granting (+) or revoking (-) the permission?
- Which permission are we setting? -read (r), write (w) or execute (x) ?

Lets do some practice:

1. For the file `main.azt` , grant the execute permission to the group.

```
azat@pop-os:~/Temp$ ls -l main.azt
-rw-r--r-- 1 azat azat 0 Jan  9 21:15 main.azt
azat@pop-os:~/Temp$ chmod g+x main.azt
azat@pop-os:~/Temp$ ls -l main.azt
-rw-r-xr-- 1 azat azat 0 Jan  9 21:15 main.azt
```

2. Then remove the write permission for the owner.

```
azat@pop-os:~/Temp$ chmod u-w main.azt
azat@pop-os:~/Temp$ ls -l main.azt
-r--r-xr-- 1 azat azat 0 Jan  9 21:15 main.azt
azat@pop-os:~/Temp$
```

3. Assign multiple permission at once:

```
azat@pop-os:~/Temp$ ls -l main.azt
ls: cannot access '-': No such file or directory
ls: cannot access 'l': No such file or directory
main.azt
azat@pop-os:~/Temp$ c
```

I

1

```
azat@pop-os:~/Temp$ ls -l main.azt
-r--r-xr-- 1 azat azat 0 Jan  9 21:15 main.azt
azat@pop-os:~/Temp$ chmod gu+wx main.azt
azat@pop-os:~/Temp$ ls -l main.azt
-rwxrwxr-- 1 azat azat 0 Jan  9 21:15 main.azt
azat@pop-os:~/Temp$ chmod go-wx main.azt
azat@pop-os:~/Temp$ ls -l main.azt
-rwxr--r-- 1 azat azat 0 Jan  9 21:15 main.azt
azat@pop-os:~/Temp$
```

## Set permission shorthand

There is a shorthand method works as well to specify the permissions. Our typical number system is decimal. It is a base 10 number system and as such has 10 symbols (0-9) used. Another number system is octal which is base 8 (0-7). Now it just so happens that with 3 permissions and each being on or off, we have 8 possible combinations ( $2^3$ ). Now we can also represent our numbers using binary which only has 2 symbols (0 and 1). The mapping of octal to binary is in the table below.

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Now the interesting point to note is that we may represent all 8 octal values with 3 binary bits and that every possible combination of 1 and 0 is included in it. So we have 3 bits and we also have 3 permissions.

If you think of 1 as representing on and 0 as off then a single octal number may be used to represent a set of permissions for a set of people.

Three numbers and we can specify permissions for the user, group and others. Let's see some examples. (refer to the table above to see how they match)

```
azat@pop-os:~/Temp$ touch demo
azat@pop-os:~/Temp$ ls -l demo
-rw-r--r-- 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 751 demo
azat@pop-os:~/Temp$ ls -l demo
-rwxr-x--x 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$
```

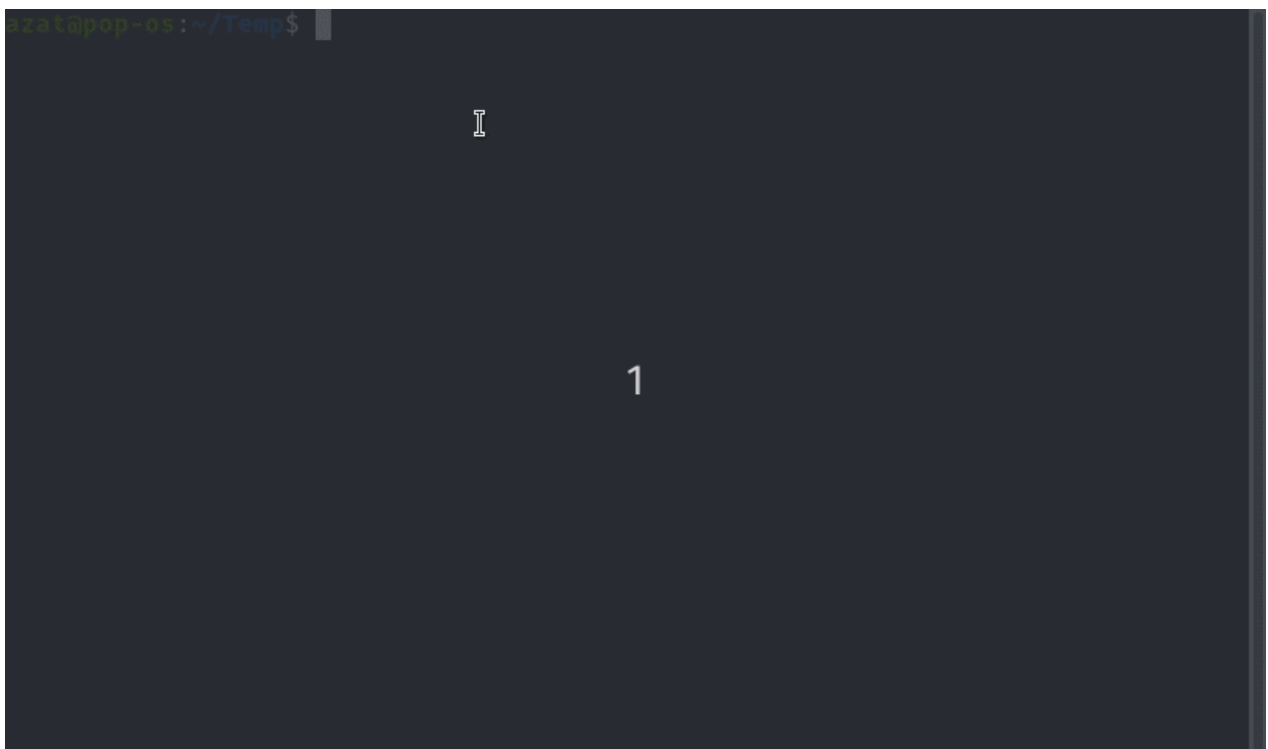
People often remember commonly used number sequences for different types of files and find this method quite convenient. For example **755** or **750** are commonly used for scripts.

If you confused then, I wanna make it a little bit clear:

We use three numbers right ? Then it means the first number is for the user, second for the group and third for others.

```
azat@pop-os:~/Temp$ ls -l demo
----- 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 1 demo
azat@pop-os:~/Temp$ ls -l demo
-----x 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 2 demo
azat@pop-os:~/Temp$ ls -l demo
```

```
-----w- 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 3 demo
azat@pop-os:~/Temp$ ls -l demo
-----wx 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 4 demo
azat@pop-os:~/Temp$ ls -l demo
-----r-- 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 5 demo
azat@pop-os:~/Temp$ ls -l demo
-----r-x 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 6 demo
azat@pop-os:~/Temp$ ls -l demo
-----rw- 1 azat azat 0 Jan  9 21:30 demo
azat@pop-os:~/Temp$ chmod 7 demo
azat@pop-os:~/Temp$ ls -l demo
-----rwx 1 azat azat 0 Jan  9 21:30 demo
```



You should also be careful about the order of permission.

		u	g	o	
		7	5	4	
		/   \			
access		r w x	r w x	r w x	
binary		4 2 1	4 2 1	4 2 1	
enabled		1 1 1	1 0 1	1 0 0	
		<hr/>	<hr/>	<hr/>	
result		4 2 1	4 0 1	4 0 0	
		<hr/>	<hr/>	<hr/>	
total		7	5	4	

Use the chmod command to set file permissions.

The chmod command uses a three-digit code as an argument.

The three digits of the chmod code set permissions for these groups in this order:

1. Owner (you)
2. Group (a group of other users that you set up)
3. World (anyone else browsing around on the file system)

Each digit of this code sets permissions for one of these groups as follows. Read is 4. Write is 2. Execute is 1.

The sums of these numbers give combinations of these permissions:

- 0 = no permissions whatsoever; this person cannot read, write, or execute the file
- 1 = execute only
- 2 = write only
- 3 = write and execute (1+2)
- 4 = read only
- 5 = read and execute (4+1)
- 6 = read and write (4+2)
- 7 = read and write and execute (4+2+1)

## Permissions for Directories

The same series of permissions may be used for directories but they have a slightly different behaviour.

- **r** - you have the ability to read the contents of the directory (ie do an ls)
- **w** - you have the ability to write into the directory (ie create files and directories)
- **x** - you have the ability to enter that directory (ie cd)

```
azat@pop-os:~/Temp/test$ cd ..  
azat@pop-os:~/Temp$
```

I

1

```
azat@pop-os:~/Temp$ ls test  
file1 file2 file3  
azat@pop-os:~/Temp$ chmod 400 test  
azat@pop-os:~/Temp$ ls -ld test  
dr----- 2 azat azat 4096 Jan  9 21:52 test  
azat@pop-os:~/Temp$ cd test/  
bash: cd: test/: Permission denied  
azat@pop-os:~/Temp$ ls test  
ls: cannot access 'test/file1': Permission denied  
ls: cannot access 'test/file2': Permission denied  
ls: cannot access 'test/file3': Permission denied  
file1 file2 file3  
azat@pop-os:~/Temp$ chmod 100 test  
azat@pop-os:~/Temp$ ls test  
ls: cannot open directory 'test': Permission denied  
azat@pop-os:~/Temp$ cd test  
azat@pop-os:~/Temp/test$ pwd  
/home/azat/Temp/test
```

Note, on lines 4 above when we ran `ls` I included the `-d` option which stands for directory. Normally if we give `ls` an argument which is a directory it will list the contents of that directory. In this case however we are interested in the permissions of the directory directly and the `-d` option allows us to obtain that.

These permissions can seem a little confusing at first. What we need to remember is that these permissions are for the directory itself, not the files within. So, for example, you may have a directory which you don't have the read permission for. It may have files within it which you do have the read permission for. As long as you know the file exists and it's name you can still read the file.

# The `root` user

On a Linux system there are only 2 people usually who may change the permissions of a file or directory. The owner of the file or directory and the root user.

The root user is a superuser who is allowed to do anything and everything on the system. Typically the administrators of a system would be the only ones who have access to the root account and would use it to maintain the system. Typically normal users would mostly only have access to files and directories in their home directory and maybe a few others for the purposes of sharing and collaborating on work and this helps to maintain the security and stability of the system.

## Activities

Play around with file and directory permissions.

- Create a folder with inside your personal articles and images.
- Set permissions to all of the images using wide cards so that only you can read the file.
- Set permission to your doc or docx files so that only you and the group can write.
- Set permission to your folder so that only you can enter the folder.

## Filters

---

With the help of filters, we may easily turn data into information or can easily extract informations that we want. One of the underlying principles of Linux is that every item should do one thing and one thing only and that we can easily join these items together.

## Introduction

A filter, in the context of the Linux command line, is a program that accepts textual data and then transforms it in a particular way. Filters are a way to take raw data, either produced by another program, or stored in a file, and manipulate it to be displayed in a way more suited to what we are after.

## Head

Head is a program that prints the first so many lines of it's input. By default it prints 10 lines.

```
head [-number of lines to print] [path]
```

```
azat@pop-os:~/Temp$
```



```
azat@pop-os:~/Temp$ head mysampleddata.txt
```

```
Fred apples 20
```

```
Susy oranges 5
```

```
Mark watermellons 12
```

```
Robert pears 4
```

```
Terry oranges 9
```

```
Lisa peaches 7
```

```
Susy oranges 12
```

```
Mark grapes 39
```

```
Anne mangoes 7
```

```
Greg pineapples 3
```

Of course, we can show as many lines as we want:



```
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
azat@pop-os:~/Temp$ head mysampleddata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
azat@pop-os:~/Temp$ clear
```

```
azat@pop-os:~/Temp$ head 3 mysampleddata.txt
head: cannot open '3' for reading: No such file or directory
==> mysampleddata.txt <==
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
```

Take attention that, there are a hyphen there before number!

## tail

Tail is the opposite of head. Tail is a program that prints the last so many lines of it's input.

```
tail [-number of lines to print] [path]
```

```
azat@pop-os:~/Temp$
```

1

I

```
azat@pop-os:~/Temp$ tail -3 mysampleddata.txt
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

## sort

Sort will sort it's input, nice and simple. By default it will sort alphabetically but there are many options available to modify the sorting mechanism.

```
sort [-options] [path]
```

```
azat@pop-os:~/Temp$
```

I

1

## nl

`nl` stands for number lines and it dose just that.

```
azat@pop-os:~/Temp$ sort mysampleddata.txt
Anne mangoes 7
Betty limes 14
Fred apples 20
Greg pineapples 3
Lisa peaches 7
Mark grapes 39
Mark watermellons 12
Oliver rockmellons 2
Robert pears 4
Susy oranges 12
Susy oranges 5
Terry oranges 9
azat@pop-os:~/Temp$
```

I

```
azat@pop-os:~/Temp$ nl mysampleddata.txt
 1 Fred apples 20
 2 Susy oranges 5
 3 Mark watermellons 12
 4 Robert pears 4
 5 Terry oranges 9
 6 Lisa peaches 7
 7 Susy oranges 12
 8 Mark grapes 39
 9 Anne mangoes 7
10 Greg pineapples 3
11 Oliver rockmellons 2
12 Betty limes 14
```

The basic formatting is ok but sometimes you are after something a little different. With a few command line options, nl is happy to oblige.

```
nl -s '. ' -w 10 mysampleddata.txt
```

In the above example we have used 2 command line options. The first one **-s** specifies what should be printed after the number while the second one **-w** specifies how much padding to put before the numbers.

```
-s, --number-separator=STRING
    add STRING after (possible) line number

-w, --number-width=NUMBER
    use NUMBER columns for line numbers
```

```
azat@pop-os:~/Temp$
```

`wc` stands for word count.

```
wc [-options] [path]
```

```
azat@pop-os:~/Temp$ wc mysampleddata.txt
 12   36 197 mysampleddata.txt
```

Sometimes you just want one of these values. `-l` will give us lines only, `-w` will give us words and `-m` will give us characters.

```
azat@pop-os:~/Temp$ wc -l mysampleddata.txt
12 mysampleddata.txt
azat@pop-os:~/Temp$ wc -m mysampleddata.txt
197 mysampleddata.txt
azat@pop-os:~/Temp$ wc -w mysampleddata.txt
36 mysampleddata.txt
```

You may combine the command line arguments too. This example gives us both lines and words.

```
azat@pop-os:~/Temp$ wc -lw mysampleddata.txt
 12   36 mysampleddata.txt
```

## cut

`cut` is a nice little program to use if your content is separated into fields (columns) and you only want certain fields.

```
cut [-options][path]
```

In our sample file we have our data in 3 columns, the first is a name, the second is a fruit and the third an amount. Let's say we only wanted the first column.

```
-d, --delimiter=DELIM
        use DELIM instead of TAB for field delimiter
-f, --fields=LIST
        select only these fields; also print any line that contains no
        delimiter character, unless the -s option is specified
```

`cut` defaults to using the TAB character as a separator (delimiter, `-d`) to identify fields (`-f`). In our file we have used a single space instead so we need to tell `cut` to use that instead.

```
azat@pop-os:~/Temp$ man cut
azat@pop-os:~/Temp$ clea
```

```
azat@pop-os:~/Temp$ cut -f 1 -d ' ' mysampleddata.txt
Fred
Susy
Mark
Robert
Terry
Lisa
Susy
Mark
Anne
Greg
Oliver
Betty
```

If we wanted 2 or more fields then we separate them with a comma as below.

```
azat@pop-os:~/Temp$ cut -f 1 -d ' ' mysampleddata.txt
Fred
Susy
Mark
Robert
Terry
Lisa
Susy
Mark
Anne
Greg
Oliver
Betty
1
azat@pop-os:~/Temp$ cleaar
Command 'cleaar' not found, did you mean: I
  command 'clear' from deb ncurses-bin (6.1+20190803-1ubuntu1)
Try: sudo apt install <deb name>
azat@pop-os:~/Temp$
```

```
azat@pop-os:~/Temp$ cut -f 1,2 -d ' ' mysampleddata.txt
Fred apples
Susy oranges
Mark watermellons
Robert pears
Terry oranges
Lisa peaches
Susy oranges
Mark grapes
Anne mangoes
Greg pineapples
Oliver rockmellons
Betty limes
```

## sed

sed stands for **Stream Editor** and it effectively allows us to do a search and replace on our data. It is quite a powerful command but we will use it here in it's basic format.

```
sed <expression> [path]
```

A basic expression is of the following format:

```
s/search/replace/g
```

The initial **s** stands for substitute and specifies the action to perform (there are others but for now we'll keep it simple). Then between the first and second slashes ( / ) we place what it is we are **searching** for. Then between the second and third slashes, what it is we wish to **replace** it with. The **g** at the end stands for global and is optional.

```
azat@pop-os:~/Temp$
```

I

```
azat@pop-os:~/Temp$ sed 's/oranges/bananas/g' mysampleddata.txt
Fred apples 20
Susy bananas 5
Mark watermellons 12
Robert pears 4
Terry bananas 9
Lisa peaches 7
Susy bananas 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

It's important to note that sed does not identify words but strings of characters.

## uniq

uniq stands for **unique** and it's job is to remove duplicate lines from the data. One limitation however is that those lines must be adjacent (ie, one after the other).

```
uniq [options] [path]
```



```
azat@pop-os:~/Temp$
```

I

```
azat@pop-os:~/Temp$ cat mysampleddata.txt
```

```
Fred apples 20
```

```
Susy oranges 5
```

```
Susy oranges 5
```

```
Susy oranges 5
```

```
Mark watermellons 12
```

```
Robert pears 4
```

```
Terry oranges 9
```

```
Lisa peaches 7
```

```
Susy oranges 12
```

```
Mark grapes 39
```

```
Mark grapes 39
```

```
Anne mangoes 7
```

```
Greg pineapples 3
```

```
Oliver rockmellons 2
```

```
Betty limes 14
```

```
azat@pop-os:~/Temp$ uniq mysampleddata.txt
```

```
Fred apples 20
```

```
Susy oranges 5
```

```
Mark watermellons 12
```

```
Robert pears 4
```

```
Terry oranges 9
```

```
Lisa peaches 7
```

```
Susy oranges 12
```

```
Mark grapes 39
```

```
Anne mangoes 7
```

```
Greg pineapples 3
```

```
Oliver rockmellons 2
```

```
Betty limes 14
```

# tac

Linux guys are known for having a funny sense of humor. The program tac is actually cat in reverse. It was named this as it does the opposite of cat. Given data it will print the last line first, through to the first line.

```
tac [path]
```

```
azat@pop-os:~/Temp$ tac mysampleddata.txt
Betty limes 14
Oliver rockmellons 2
Greg pineapples 3
Anne mangoes 7
Mark grapes 39
Mark grapes 39
Susy oranges 12
Lisa peaches 7
Terry oranges 9
Robert pears 4
Mark watermellons 12
Susy oranges 5
Susy oranges 5
Susy oranges 5
Fred apples 20
azat@pop-os:~/Temp$
```

## Others

There are a lot and a lot more linux commands to explore, you can search for them and learn, it would be really cool ~

- awk
- diff
- .....

## Summary

command	action
<code>head</code>	View the first n lines of the data.
<code>tail</code>	View the last n lines of the data.
<code>sort</code>	Organises the data into order.
<code>nl</code>	Print line numbers before the data.
<code>wc</code>	Print a count of lines, words, and characters.
<code>cut</code>	Cut the data into fields and only display the specified fields.
<code>sed</code>	Do a search and replace on the data.
<code>uniq</code>	Remove duplicated lines.
<code>tac</code>	Print the data in reverse order.

## Activities

Let's mangle some data.

- First off, you may want to make a file with data similar to our sample file.
- Now play with each of the programs we looked at above. Make sure you use both relative and absolute paths.
- Have a look at the man page for each of the programs and try at least 2 of the command line options for them.

## Grep and Regular Expressions!

### Regular expression

In this part of tutorial we will look at another filter which is quite powerful when combined with a concept that called regular expressions or re's for short. Regular expressions are similar to the wildcards that we learned at previous section. They allow us to create a pattern. They are a bit more powerful however. Regular expressions are typically used to identify and manipulate specific pieces of data. Eg. We may wish to identify every line which contains an email address or a url in a set of data.

### egrep

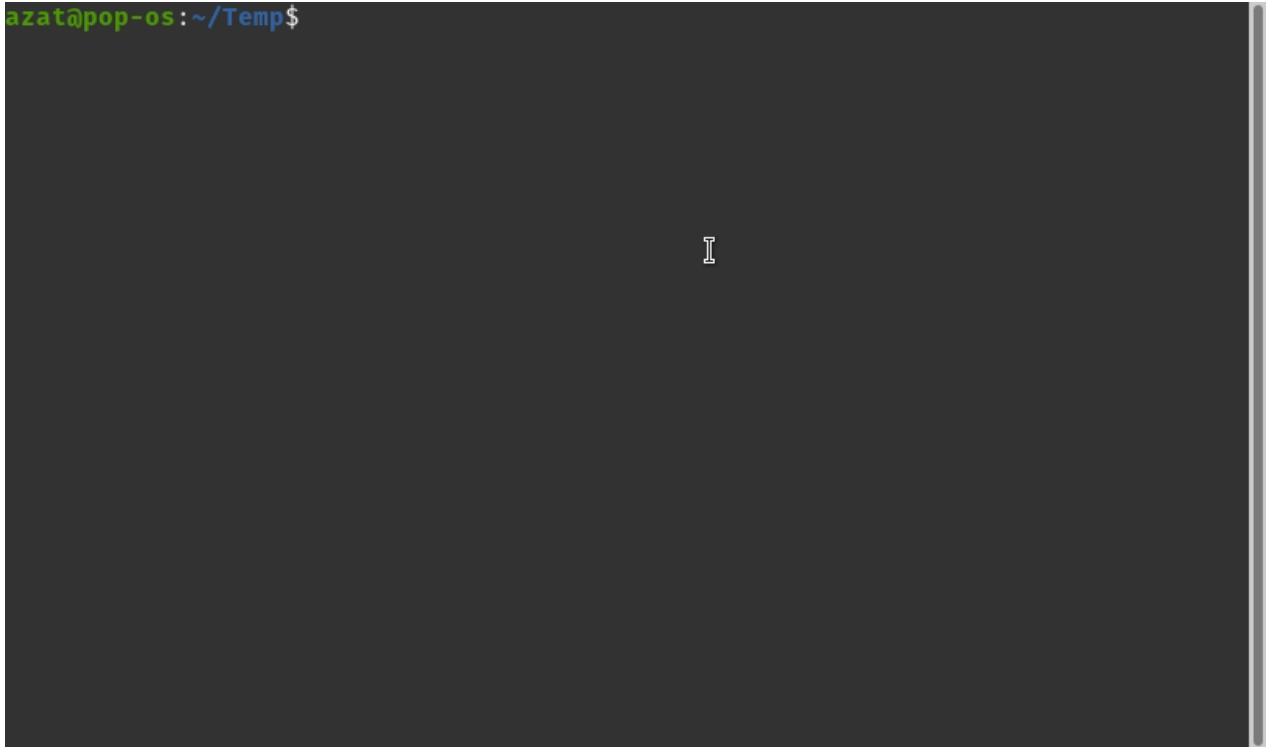
Egrep is a program which will search a given set of data and print every line which contains a given pattern, it is an extension of a program called grep. There are many options of egrep, for ie the `-v` option tells grep to instead print every line which does not match the pattern.

```
egrep [command line options] <pattern> [path]
```

Let's say we wished to identify every line which contained the string **mellon**

```
egrep 'mellon' mysampleddata.txt
```

```
azat@pop-os:~/Temp$
```



The basic behaviour of egrep is that it will print the entire line for every line which contains a string of characters matching the given pattern. This is important to note, we are not searching for a word but a string of characters.

Sometimes we want to know not only which lines matched but their line number as well.

```
azat@pop-os:~/Temp$
```



```
azat@pop-os:~/Temp$ egrep -n 'mellon' mysampleddata.txt
5:Mark watermellons 12
14:Oliver rockmellons 2
```

Or maybe we are not interested in seeing the matched lines but wish to know how many lines did match.

```
azat@pop-os:~/Temp$
```

```
azat@pop-os:~/Temp$ egrep -c 'mellon' mysampleddata.txt
2
```

## Regular expression overview

Here we are not going to talk about regular expressions a lot , but just some basics and overview. You should able to search google for regular expression and learn for other resources in detail.

You can also find very useful ebooks from here [AzatAI CS BOOKS](#)

Here are the outline of basic building blocks of regular expression's below then follow with a set of example's to demonstrate their usage.

- **.** (**dot**) - a single character.
- **?** - the preceding character matches 0 or 1 times only.
- **\*** - the preceding character matches 0 or more times.
- **+** - matches 1 or more times.
- **{n}** - matches exactly **n** times.
- **{n,m}** - matches at least **n** times and no more than **m** times.
- **[agd]** matches one of those included with in the square brackets.
- **[^agd]** the character is not one of those included with in the square brackets.
- **[c-f]** - the dash within the square brackets operates as a range. From **c** to **f**, then it is

c,d,e,or f.

- `()` - group several characters to behave as one.
- `|` (**pipe symbol**) - the original OR operation.
- `^` - matches the beginning of the line.
- `$` - matches the end of the line.

## Some examples

Let's say we wish to identify any line with two or more vowels in a row.

```
azat@pop-os:~/Temp$ egrep -c 'mellon' mysampleddata.txt
2
azat@pop-os:~/Temp$ egrep '[aeiou]{2,}' mysampleddata.txt
Robert pears 4
Lisa peaches 7
Anne mangoes 7
Greg pineapples 3
azat@pop-os:~/Temp$ clea
```

```
azat@pop-os:~/Temp$ egrep '[aeiou]{2,}' mysampleddata.txt
Robert pears 4
Lisa peaches 7
Anne mangoes 7
Greg pineapples 3
```

How about any line with a 2 on it which is not the end of the line.

```
azat@pop-os:~/Temp$ egr
```

I

```
azat@pop-os:~/Temp$ egrep '2.+' mysampleddata.txt  
Fred apples 20
```

The number 2 as the last character on the line.

```
azat@pop-os:~/Temp$ egrep '2.+' mysampleddata.txt  
Fred apples 20  
azat@pop-os:~/Temp$
```

I

1

```
azat@pop-os:~/Temp$ egrep '2$' mysampleddata.txt  
Mark watermellons 12  
Susy oranges 12  
Oliver rockmellons 2
```

And now each line which contains either 'is' or 'go' or 'or'.

```
azat@pop-os:~/Temp$ egrep '2.+' mysampleddata.txt
Fred apples 20
azat@pop-os:~/Temp$ egrep '2$' mysampleddata.txt
Mark watermellons 12
Susy oranges 12
Oliver rockmellons 2
azat@pop-os:~/Temp$
```

1

I

```
azat@pop-os:~/Temp$ egrep 'is|or|go' mysampleddata.txt
Susy oranges 5
Susy oranges 5
Susy oranges 5
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Anne mangoes 7
```

Maybe we wish to see orders for everyone who's name begins with A - K.



```

azat@pop-os:~/Temp$ egrep '2.+' mysampleddata.txt
Fred apples 20
azat@pop-os:~/Temp$ egrep '2$' mysampleddata.txt
Mark watermellons 12
Susy oranges 12
Oliver rockmellons 2
azat@pop-os:~/Temp$ egrep 'is|or|go' mysampleddata.txt
Susy oranges 5
Susy oranges 5
Susy oranges 5
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Anne mangoes 7
azat@pop-os:~/Temp$ █

```

1

I

```

azat@pop-os:~/Temp$ egrep '^[A-K]' mysampleddata.txt
Fred apples 20
Anne mangoes 7
Greg pineapples 3
Betty limes 14

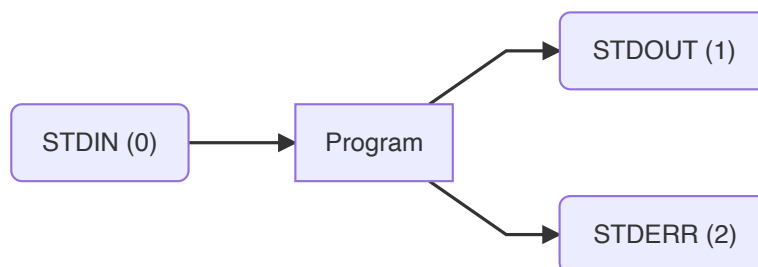
```

## Piping and redirection

### Introduction

Every command we ran on the command line automatically has three data streams connected to it.

- STDIN (0) - Standard input (data fed into the program)
- STDOUT (1) - Standard output (data printed by the program, defaults to to the terminal)
- STDERR (2) - Standard error (for error messages, also defaults to the terminal)



Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

We'll demonstrate piping and redirection below with several examples but these mechanisms will work with every program on the command line, not just the ones we have used in the examples.

## Redirecting to a file

Normally, we will get our output on the screen, which is convenient most of the time, but sometimes we may wish to save it into a file to keep as a record, feed into another system, or send to someone else. The greater than operator ( > ) indicates to the command line that we wish the programs output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen. Let's see an example.

```
azat@pop-os:~/Temp$ egrep '2.+' mysampleddata.txt
Fred apples 20
azat@pop-os:~/Temp$ egrep '2$' mysampleddata.txt
Mark watermellons 12
Susy oranges 12
Oliver rockmellons 2
azat@pop-os:~/Temp$ egrep 'is|or|go' mysampleddata.txt
Susy oranges 5
Susy oranges 5
Susy oranges 5
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Anne mangoes 7
azat@pop-os:~/Temp$ egrep '^[A-K]' mysampleddata.txt
Fred apples 20
Anne mangoes 7
Greg pineapples 3
Betty limes 14
azat@pop-os:~/Temp$ clear
```

```
azat@pop-os:~/Temp$ ls
Almaty      blahblah.txt  file2.txt  foofoo      main.azt      ok.request
azt.md      bob.md        firstfile  images      mysampleddata.txt  secret.mp4
blafiii.txt demo          fool       love.mp3    ok            video.mepg
azat@pop-os:~/Temp$ ls > myoutput
azat@pop-os:~/Temp$ cat myoutput
Almaty
azt.md
blafiii.txt
blahblah.txt
bob.md
demo
file2.txt
firstfile
fool
foofoo
images
```

```
love.mp3
main.azt
myoutput
mysampleddata.txt
ok
ok.request
secret.mp4
video.mepg
azat@pop-os:~/Temp$
```

## Saving to an existing file

If we redirect to a file which does not exist, it will be created automatically for us. If we save into a file which already exists, however, then its contents will be cleared, then the new output saved to it.

We can instead get the new data to be appended to the file by using the double greater than operator (>>).

```
azat@pop-os:~/Temp$ ls
Almaty      blahblah.txt  file2.txt  foofoo      main.azt      ok.request
azt.md      bob.md        firstfile  images      mysampleddata.txt  secret.mp4
blafiii.txt demo          foo1       love.mp3    ok            video.mepg
azat@pop-os:~/Temp$ ls > myoutput
azat@pop-os:~/Temp$ cat myoutput
Almaty
azt.md
blafiii.txt
blahblah.txt
bob.md
demo
file2.txt
firstfile
foo1
foofoo
images
love.mp3
main.azt
myoutput
mysampleddata.txt
ok
ok.request
secret.mp4
```

```
azat@pop-os:~/Temp$ file azt.md >>myoutput
azat@pop-os:~/Temp$ tac myoutput
azt.md: ASCII text, with very long lines
video.mepg
secret.mp4
ok.request
ok
mysampleddata.txt
myoutput
main.azt
love.mp3
```

```
images
foofoo
fool
firstfile
file2.txt
demo
bob.md
blahblah.txt
blafiii.txt
azt.md
Almaty
azat@pop-os:~/Temp$
```

## Redirecting from a file

If we use the less than operator (<) then we can send data the other way. We will read data from the file and feed it into the program via its STDIN stream.

```
video.mpeg
azat@pop-os:~/Temp$ file azt.md >>myoutput
azat@pop-os:~/Temp$ tac myoutput
azt.md: ASCII text, with very long lines
video.mpeg
secret.mp4
ok.request
ok
mysampleddata.txt
myoutput
main.azt
love.mp3
images
foofoo
fool
firstfile
file2.txt
demo
bob.md
blahblah.txt
blafiii.txt
azt.md
Almaty
azat@pop-os:~/Temp$ clea
```

```
azat@pop-os:~/Temp$ wc -l myoutput
20 myoutput
azat@pop-os:~/Temp$ wc -l < myoutput
20
```

A lot of programs (as we've seen in previous sections) allow us to supply a file as a command line argument and it will read and process the contents of that file.

Whenever we use redirection or piping, the data is sent anonymously. So in the above example, wc received some content to process, but it has no knowledge of where it came from so it may not print this information. As a result, this mechanism is often used in order to get ancillary data (which may not be required) to not be printed.

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

```
azat@pop-os:~/Temp$ wc -l myoutput
20 myoutput
azat@pop-os:~/Temp$ wc -l < myoutput
20
azat@pop-os:~/Temp$
```

```
azat@pop-os:~/Temp$ wc -l < azt.md >>myoutput
azat@pop-os:~/Temp$ tac myoutput
39
azt.md: ASCII text, with very long lines
video.mepg
secret.mp4
ok.request
ok
mysampleddata.txt
myoutput
main.azt
love.mp3
images
foofoo
fool
firstfile
file2.txt
demo
bob.md
blahblah.txt
blafiii.txt
azt.md
Almaty
```

## Redirecting the STDERR

Now let's look at the third stream which is Standard Error or STDERR. The three streams actually have numbers associated with them (in brackets in the list at the top of the page). STDERR is stream number 2 and we may use these numbers to identify the streams. If we place a number before the > operator then it will redirect that stream (if we don't use a number, like we have been doing so far, then it defaults to stream 1).

```
azat@pop-os:~/Temp$ wc -l < azt.md >>myoutput
azat@pop-os:~/Temp$ tac myoutput
39
azt.md: ASCII text, with very long lines
video.mpeg
secret.mp4
ok.request
ok
mysampleddata.txt
myoutput
main.azt
love.mp3
images
foofoo
foo1
firstfile
file2.txt
demo
bob.md
blahblah.txt
blafiii.txt
azt.md
Almaty
azat@pop-os:~/Temp$ cle
```

```
azat@pop-os:~/Temp$ cd uthjg 2>errors.txt
azat@pop-os:~/Temp$ cat errors.txt
bash: cd: uthjg: No such file or directory
```

Maybe we wish to save both normal output and error messages into a single file. This can be done by redirecting the STDERR stream to the STDOUT stream and redirecting STDOUT to a file.

We redirect to a file first then redirect the error stream. We identify the redirection to a stream by placing an & in front of the stream number (otherwise it would redirect to a file called 1).

```
azat@pop-os:~/Temp$ cd hjhkjhclea
```

I

1

```
azat@pop-os:~/Temp$ ls -l ddaisd > myoutput2 2>&1  
azat@pop-os:~/Temp$ cat myoutput2  
ls: cannot access 'ddaisd': No such file or directory
```

## Piping

So far we've dealt with sending data to and from files. Now we'll take a look at a mechanism for sending data from one program to another. It's called piping and the operator we use is ( | ) (found above the backslash ( \ ) key on most keyboards).

What this operator does is feed the output from the program on the left as input to the program on the right.

```
azat@pop-os:~/Temp$
```

I

```
azat@pop-os:~/Temp$ ls
```

```
Almaty      bob.md      firstfile  love.mp3    mysampled  
data.txt    video.mepg  
azt.md      demo        fool       main.azt    ok  
blafiiii.txt errors.txt  foofoo     myoutput    ok.request  
blahblah.txt file2.txt  images     myoutput2   secret.mp4
```

```
azat@pop-os:~/Temp$ ls | head -3
```

```
Almaty  
azt.md  
blafiiii.txt
```

We may pipe as many programs together as we like.

```
azat@pop-os:~/Temp$ ls
```

```
Almaty      bob.md      firstfile  love.mp3    mysampled  
data.txt    video.mepg  
azt.md      demo        fool       main.azt    ok  
blafiiii.txt errors.txt  foofoo     myoutput    ok.request  
blahblah.txt file2.txt  images     myoutput2   secret.mp4
```

```
azat@pop-os:~/Temp$ ls | head -3
```

```
Almaty  
azt.md  
blafiiii.txt
```

```
azat@pop-os:~/Temp$
```

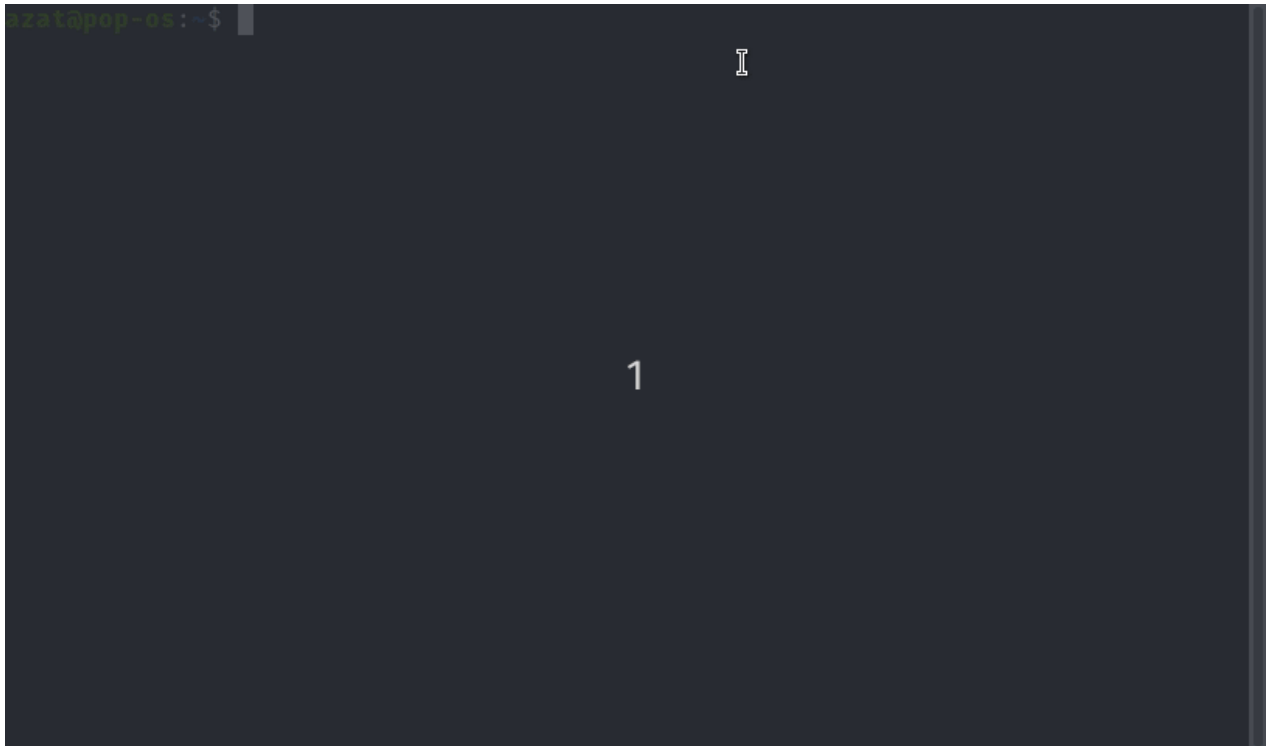
I



```
azat@pop-os:~/Temp$ ls | head -3
Almaty
azt.md
blafiii.txt
azat@pop-os:~/Temp$ ls | head -3 | tail -1
blafiii.txt
```

Any command line arguments we supply for a program must be next to that program.

You may combine pipes and redirection too.



```
azat@pop-os:~$ ls | head -3 | tail -1 >myoutput
azat@pop-os:~$ cat myoutput
Downloads
```

There are many things you can achieve with piping and these are just a few of them. With experience and a little creative thinking I'm sure you'll find many more ways to use piping to make your life easier.

In this example we are sorting the listing of a directory so that all the directories are listed first.

```
azat@pop-os:~/Temp$
```

```
-n, --lines=[+]NUM
```

output the last NUM lines, instead of the last 10; or use -n +NUM to output starting with line NUM

```
azat@pop-os:~/Temp$
```

**ls -l** long lists the directory. However the first line is some information like `total 36`.

**tail** is the opposite of **head**, it lists from back to front. And `-n +2` means, take data from the second line, and by which we do not want `total 36`.

**sort** sorts the results as the directories first.

How about if we use `head` rather than `tail` ?

```
azat@pop-os:~/Temp$ ls -l | head | sort
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blafiii.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blahblah.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 bob.md
-rw-r--r-- 1 azat azat    0 Jan  9 20:16 Almaty
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 file2.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 fool
-rw-r--r-- 1 azat azat 3399 Jan  9 12:17 azt.md
-rw-r--r-- 1 azat azat   43 Jan 10 18:39 errors.txt
-rw-r--r-- 1 azat azat   64 Jan  9 11:44 firstfile
total 36
```

`head` dose not have the `-n +NUM` option.

In this example we will feed the output of a program into the program less so that we can view it easier.

```
azat@pop-os:~/Temp$ ls -l | head | sort
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blafiii.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blahblah.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 bob.md
-rw-r--r-- 1 azat azat    0 Jan  9 20:16 Almaty
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 file2.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 fool
-rw-r--r-- 1 azat azat 3399 Jan  9 12:17 azt.md
-rw-r--r-- 1 azat azat   43 Jan 10 18:39 errors.txt
-rw-r--r-- 1 azat azat   64 Jan  9 11:44 firstfile
total 36
azat@pop-os:~/Temp$ ls -l | head | sort
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blafiii.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blahblah.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 bob.md
-rw-r--r-- 1 azat azat    0 Jan  9 20:16 Almaty
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 file2.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:21 fool
-rw-r--r-- 1 azat azat 3399 Jan  9 12:17 azt.md
-rw-r--r-- 1 azat azat   43 Jan 10 18:39 errors.txt
-rw-r--r-- 1 azat azat   64 Jan  9 11:44 firstfile
total 36
azat@pop-os:~/Temp$ clear
```

```
ls -l | less
```

```
total 36
-rw-r--r-- 1 azat azat    0 Jan  9 20:16 Almaty
-rw-r--r-- 1 azat azat 3399 Jan  9 12:17 azt.md
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blafiii.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 blahblah.txt
-rw-r--r-- 1 azat azat    0 Jan  9 20:10 bob.md
-rw-r--r-- 1 azat azat   43 Jan 10 18:39 errors.txt
```

```
-rw-r--r-- 1 azat azat 0 Jan 9 20:21 file2.txt
-rw-r--r-- 1 azat azat 64 Jan 9 11:44 firstfile
-rw-r--r-- 1 azat azat 0 Jan 9 20:21 fool
drwxr-xr-x 2 azat azat 4096 Jan 9 10:16 foofoo
drwxr-xr-x 2 azat azat 4096 Jan 9 20:41 images
-rw-r--r-- 1 azat azat 0 Jan 9 20:18 love.mp3
-rwxr--r-- 1 azat azat 0 Jan 9 21:15 main.azt
-rw-r--r-- 1 azat azat 214 Jan 10 18:36 myoutput
-rw-r--r-- 1 azat azat 54 Jan 10 18:42 myoutput2
-rw-r--r-- 1 azat azat 242 Jan 9 22:35 mysampleddata.txt
drwxr-xr-x 3 azat azat 4096 Jan 9 10:19 ok
-rw-r--r-- 1 azat azat 0 Jan 9 20:10 ok.request
-rw-r--r-- 1 azat azat 0 Jan 9 20:18 secret.mp4
-rw-r--r-- 1 azat azat 0 Jan 9 20:20 video.mepg
(END)
```

Identify all files in your home directory which the group has write permission for.

```
ls -l ~ |grep '^.....w'
```

## Summary

- > save the output to a file.
- >> Append the output to a file.
- < Read input from a file.
- 2> Redirect error message.
- | Send the output from one program as input to the another program.
- **Streams** Every program you may run on the command line has 3 streams, STIN, STDOUT and STDERR.

## Activities

Let's mangle some data:

- First off, experiment with saving output from various commands to a file. Overwrite the file and append to it as well. Make sure you are using a both absolute and relative paths as you go.
- Now see if you can list only the 20th last file in the directory /etc.
- Finally, see if you can get a count of how many files and directories you have the execute permission for in your home directory.

## Process Management

---

### Introduction

A program is a series of instructions that tell the computer what to do. When we run a program, those instructions are copied into memory and space is allocated for variables and other stuff required to manage its execution. This running instance of a program is called a process and its processes which we manage.

## top what is currently running ?

Linux, like many other operating systems, is a multi tasking operating system. This means that many processes are running at the same time, and even many users are running at the same time. If we would like to get a snapshot of what is currently happening on the system, we can use `top` command.

```
azat@pop-os:~/Temp$ ls -l ~ | grep '^.....w'
azat@pop-os:~/Temp$
```

```
top - 11:20:31 up 1:25, 1 user, load average: 0.13, 0.03, 0.01
Tasks: 203 total, 1 running, 202 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.6 us, 3.1 sy, 0.0 ni, 93.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 1981.1 total, 143.2 free, 836.4 used, 1001.6 buff/cache
MiB Swap: 4095.5 total, 4095.5 free, 0.0 used. 975.5 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1289	azat	20	0	159400	45624	12152	S	4.7	2.2	0:29.70	Xorg
1695	azat	-2	0	2692344	333136	73660	S	4.7	16.4	0:33.59	gnome-she+
2568	azat	20	0	631112	47368	33792	S	2.3	2.3	0:05.24	gnome-ter+
744	message+	20	0	8800	5860	3856	S	0.7	0.3	0:14.97	dbus-daem+

1959	azat	20	0	550708	28420	19560	S	0.7	1.4	0:21.32	prlcc
1	root	20	0	101580	10980	8012	S	0.3	0.5	0:01.09	systemd
1974	azat	20	0	350160	14804	9812	S	0.3	0.7	0:02.44	prlsga
18198	azat	20	0	21560	3752	3112	R	0.3	0.2	0:00.04	top
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0+
9	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu+
10	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd+
11	root	20	0	0	0	0	I	0.0	0.0	0:02.15	rcu_sched
12	root	rt	0	0	0	0	S	0.0	0.0	0:00.02	migration+
13	root	-51	0	0	0	0	S	0.0	0.0	0:00.00	idle_inje+

Some tasks (process) is sleeping means, they are waiting until a particular event occurs, which they will then act upon.

Line 3 is the CPU usage.

Line 4 is the memory usage. (You should have to worry about that)

Line 5 is the virtual memory : (SWAP) , you should increase it's size if your computer has used too much of them.

How about searching google for SWAP ?

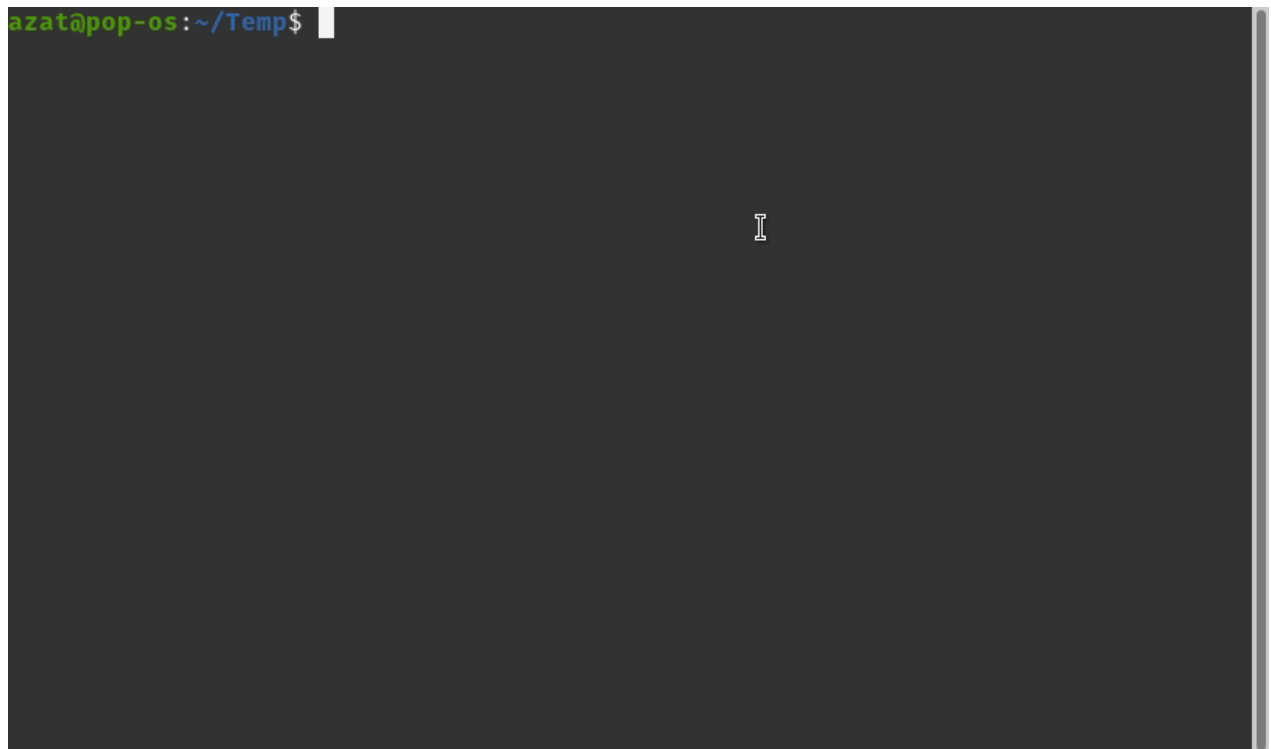
Line 7 to - : Listing of the most resource intensive process on the system ( in order of resource usage). This list will update in real time and so is interesting to watch to get and idea of what is happening on your system.

The two important coloumns to consider are memory and CPU usage. If either of these is hight for particular process over a period of time ,it may worth looking into why this is so. The user column shows who owns the process and the PID column identifies a process's Process ID.

Top will give you a realtime view of the system and only show the number of processes which will fit on the screen.

## ps Process

Another program to look at process is called `ps` which stands for process. In it's normal usage it will show you just the processes running in your current terminal( which is usually not very much). If we add the argument `aux` then it will show a complete view which is a bit more helpful.



```
azat@pop-os:~/Temp$ ps
  PID TTY          TIME CMD
 2580 pts/0    00:00:00 bash
18117 pts/0    00:00:00 ps
```

`ps` prints the current terminal process.

```
azat@pop-os:~/Temp$ ps a
  PID TTY          STAT TIME  COMMAND
 1284 tty2      Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SH
 1289 tty2      Sl+    1:16 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1
 1435 tty2      Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --
systemd
 2580 pts/0      Ss     0:00 bash
 4784 pts/1      Ss     0:00 bash
17399 pts/1      S+     0:00 man ps
17412 pts/1      S+     0:00 pager
18247 pts/0      R+     0:00 ps a
```

`ps a` show process for all users. (Processes attached to the terminal)

```

azat@pop-os:~/Temp$ ps au
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
azat         1284   0.0   0.3 173640   6420 tty2    Ssl+  10:13   0:00 /usr/lib/gdm3/g
azat         1289   0.4   2.2 159912  45624 tty2    Sl+   10:13   1:16 /usr/lib/xorg/X
azat         1435   0.0   0.7 201808  15380 tty2    Sl+   10:13   0:00 /usr/lib/gnome-
azat         2580   0.0   0.2  20648   5324 pts/0    Ss    10:14   0:00 bash
azat         4784   0.0   0.2  20644   5040 pts/1    Ss    10:22   0:00 bash
azat        17399   0.0   0.1  19364   3864 pts/1    S+    14:46   0:00 man ps
azat        17412   0.0   0.1  18188   2548 pts/1    S+    14:46   0:00 pager
azat        18447   0.0   0.1  21216   3396 pts/0    R+    14:52   0:00 ps au
azat@pop-os:~/Temp$

```

`ps au` show process for all user , and also show the process's user/owner.

```

azat@pop-os:~/Temp$ ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1   0.0   0.5 101580  10980 ?        Ss    10:13   0:01 /sbin/init
spld
root           2   0.0   0.0      0      0 ?        S     10:13   0:00 [kthreadd]
root           3   0.0   0.0      0      0 ?        I<    10:13   0:00 [rcu_gp]
root           4   0.0   0.0      0      0 ?        I<    10:13   0:00 [rcu_par_gp]
root           6   0.0   0.0      0      0 ?        I<    10:13   0:00
[kworker/0:0H-k
root           9   0.0   0.0      0      0 ?        I<    10:13   0:00
[mm_percpu_wq]
root          10   0.0   0.0      0      0 ?        S     10:13   0:00 [ksoftirqd/0]
root          11   0.0   0.0      0      0 ?        R     10:13   0:07 [rcu_sched]
root          12   0.0   0.0      0      0 ?        S     10:13   0:00 [migration/0]
root          13   0.0   0.0      0      0 ?        S     10:13   0:00
[idle_inject/0]
root          14   0.0   0.0      0      0 ?        S     10:13   0:00 [cpuhp/0]
root          15   0.0   0.0      0      0 ?        S     10:13   0:00 [cpuhp/1]
root          16   0.0   0.0      0      0 ?        S     10:13   0:00
[idle_inject/1]
root          17   0.0   0.0      0      0 ?        S     10:13   0:00 [migration/1]
root          18   0.0   0.0      0      0 ?        S     10:13   0:00 [ksoftirqd/1]
root          20   0.0   0.0      0      0 ?        I<    10:13   0:00
[kworker/1:0H-k
root          21   0.0   0.0      0      0 ?        S     10:13   0:00 [kdevtmpfs]

```

`ps aux` show process for all user, including process's owner, also show processes that not attached to terminal.

It does give quite a bit of output so people usually pipe the output to **grep** to filter out just the data they are after. We will see in the next bit an example of this.



## kill Killing a crashed process

When a program crashes, it can be quite annoying. However, we can easily kill a process and then reopen it, for example let's assume that our Firefox stuck when we try to open <https://azat.ai> , and so , to start we first need to identify the Firefox's process ID:

```
azat@pop-os: ~/Temp$
```

I

1

```
azat@pop-os:~/Temp$ ps aux | grep 'firefox'
azat      19751  3.2 13.5 2775580 274256 pts/1    Sl+   14:58   0:04
/usr/lib/firefox/firefox
azat      19795  1.0  7.8 2482740 158632 pts/1    Sl+   14:59   0:01
/usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -prefsLen 1 -
prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      19832  0.4  5.2 2397172 106896 pts/1    Sl+   14:59   0:00
/usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -prefsLen 6975
-prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      19937  0.2  4.1 2384400 83704 pts/1    Sl+   14:59   0:00
/usr/lib/firefox/firefox -contentproc -childID 4 -isForBrowser -prefsLen 11762
-prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      20290  0.0  0.0  18664   980 pts/0      S+    15:01   0:00 grep --
color=auto firefox
```

As we can easily, the Firefox's main process ID is 19571 , let's kill it now (let's be polite first, ask the program quit itself) :

```
azat@pop-os:~/Temp$
```

I

```
azat@pop-os:~/Temp$ ps aux|grep 'firefox'
azat      19751  1.6 13.8 2775580 280788 pts/1    Sl+   14:58   0:04
/usr/lib/firefox/firefox
azat      19795  0.4  7.8 2482740 158972 pts/1    Sl+   14:59   0:01
/usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -prefsLen 1 -
prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      19832  0.2  5.2 2397172 107024 pts/1    Sl+   14:59   0:00
/usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -prefsLen 6975
-prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      19937  0.1  4.1 2384400 83704 pts/1    Sl+   14:59   0:00
/usr/lib/firefox/firefox -contentproc -childID 4 -isForBrowser -prefsLen 11762
-prefMapSize 202500 -parentBuildID 20191205203726 -greomni
/usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appdir
/usr/lib/firefox/browser 19751 true tab
azat      20738  0.0  0.0  18664   848 pts/0     S+    15:03   0:00 grep --
color=auto firefox
azat@pop-os:~/Temp$ kill 19751
```

```
kill [signal] <PID>
```

It is the number next to the owner of the process that is the PID (Process ID). We will use this to identify which process to kill. To do so we use a program which is appropriately called **kill**.

Ok , how do you think ? Of course not all the programs are quite polite that some of them will not quite, let's check whether Firefox still working :

```
azat@pop-os:~/Temp$
```

I

You know what, that's quite bad that I asked Firefox to quite politely but however, it did not fully, so this time, I wanna make sure that Firefox killed, completely .

When we just use `kill <PID>` actually the kill sends the default signal `1` to the process, asking very politely to leave the process quit itself. If it doesn't work, we can bit not polity at all like, changing the signal of `9` , which really means make sure the process is well and truly gone.

For your understanding, here is a table of Linux SIGNAL actions and their portable numbers.

Signal	Portable number	Default Action	Description
SIGHUP	1	Terminate	Hangup
SIGINT	2	Terminate	Terminal interrupt signal
SIGQUIT	3	Terminate (core dump)	Terminal quit signal
SIGILL	4	Terminate (core dump)	Illegal instruction
SIGTRAP	5	Terminate (core dump)	Trace/breakpoint trap
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGBUS	7	Terminate (core dump)	Bus error (bad memory access)
SIGFPE	8	Terminate (core dump)	Erroneous arithmetic operation
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
..	..	..	..

- Control+C (control character `intr`) sends SIGINT which will interrupt the application. Usually causing it to abort, but this is up to the application to decide.
- Control+Z (control character `susp`) sends SIGTSTP to a foreground application, effectively putting it in the background, suspended. This is useful if you need to break out of something like an editor to go and grab some data you needed. You can go back into the application by running `fg` (or `%x` where `x` is the job number as shown in `jobs`).

Normal users may only kill processes which they are the owner for. The root user on the system may kill anyones processes.

Linux actually runs several virtual consoles. Most of the time we only see console 7 which is the GUI but we can easily get to the others. To switch between consoles you use the keyboard sequence **CTRL + ALT + F**.

## Foreground and background jobs

You probably won't need to do too much with foreground and background jobs but it's worth knowing about them just for those rare occasions. When we run a program normally they are run in the foreground. Most of them run to completion in a fraction of a second as well. Maybe we wish to start a process that will take a bit of time and will happily do it's thing without intervention from us (processing a very large text file or compiling a program for instance). What

we can do is run the program in the background and then we can continue working. We'll demonstrate this with a program called `sleep`. All sleep does is wait a given number of seconds and then quit. We can also use a program called `jobs` which lists currently running background jobs for us.

```
jobs
```

```
azat@pop-os:~/Temp$ ps aux|grep 'firefox'
azat    21400  0.0  0.0 18664  984 pts/0    S+   15:07   0:00 grep --color=au
to firefox
azat@pop-os:~/Temp$
```

```
azat@pop-os:~/Temp$ sleep 5
azat@pop-os:~/Temp$ jobs
```

Now we will run the same command as above but instead put an **ampersand (&)** at the end of the command then we are telling the terminal to run this process in the background.

```
azat@pop-os:~/Temp$
```

```
azat@pop-os:~/Temp$ sleep 5 &
[1] 27824
azat@pop-os:~/Temp$
[1]+  Done                  sleep 5
azat@pop-os:~/Temp$
```

We can move jobs between the foreground and background as well. If you press **CTRL +z** then the currently running foreground process will be paused and moved into the background. We can then use a program called `fg` which stands for foreground to bring background process into the foreground.

```
fg <job number>
```

```
azat@pop-os:~/Temp$ sleep 15 &
[1] 30089
azat@pop-os:~/Temp$ sleep 10
^Z
[2]+  Stopped                  sleep 10
azat@pop-os:~/Temp$ jobs
[1]-  Running                  sleep 15 &
[2]+  Stopped                  sleep 10
azat@pop-os:~/Temp$ fg 2
sleep 10
[1] Done                      sleep 15
azat@pop-os:~/Temp$
```

```
azata@pop-os:~/Temp$ sleep 5 &
[1] 27824
azata@pop-os:~/Temp$
[1]+  Done                  sleep 5
azata@pop-os:~/Temp$ clea
```

1

## Summary

- **top**  
View real-time data about processes running on the system.
- **ps**  
Get a listing of processes running on the system.
- **kill**  
End the running of a process.
- **jobs**  
Display a list of current jobs running in the background.
- **fg**  
Move a background process into the foreground.
- **ctrl + z**  
Pause the current foreground process and move it into the background.
- **Control**  
We have quite a bit of control over the running of our programs.

## Activities

Time for some fun:

- First off, start a few programs in your desktop. Then use `ps` to identify their PID and kill them.
- Now see if you can do the same, but switch to another virtual console first.
- Finally, play about with the command `sleep` and moving processes between the foreground

and background.