Basics of Linux Command Line

Ashwini Mathur [Asst. Professor]

#Reference or Source : Ubuntu Community or source **The Linux Command Line** **by** ***William Shotts***

Overview

The Linux command line is a **text interface** to your computer. Often referred to as the shell, terminal, console, prompt or various other names, it can give the appearance of being complex and confusing to use.

Agenda

- A little history of the command line
- How to access the command line from your own computer
- How to perform some basic file manipulation
- A few other useful commands
- How to chain commands together to make more powerful tools
- The best way to use administrator powers

A brief history lesson

During the formative years of the computer industry, one of the early operating systems was called Unix. It was designed to run as a **multi-user system** on mainframe computers, with users connecting to it remotely via individual *terminals*.

Obviously, therefore, any programs that ran on the mainframe had to produce text as an output and accept text as an input.

Compared with graphics, text is very light on resources. Even on machines from the 1970s, running hundreds of terminals across glacially slow network connections (by today's standards), users were still able to interact with programs quickly and efficiently.

This speed and efficiency is one reason why this text interface is still widely used today.

Each of these tasks required its own program or command:

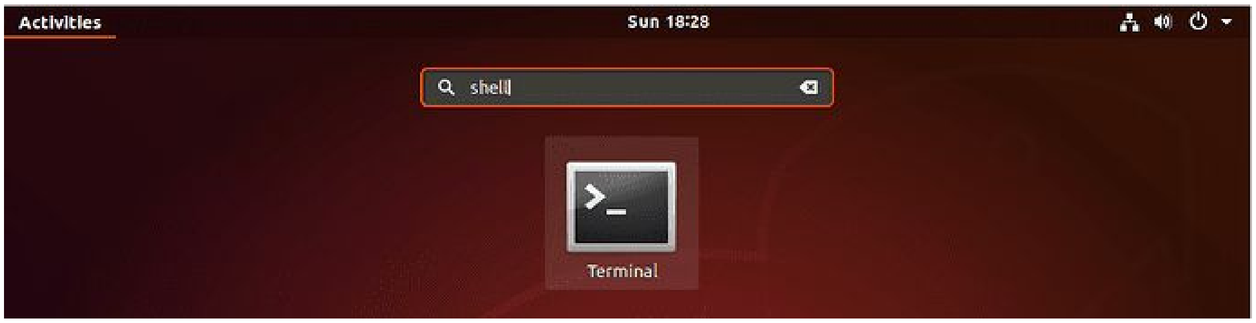
- change directories (`cd`),
- list their contents (`ls`),
- rename or move files (`mv`), and so on.

In order to coordinate the execution of each of these programs, **the user would connect to one single master program** that could then be used to launch any of the others. By wrapping the user's commands this "**shell**" program, as it was known, could provide common capabilities to any of them, such as the ability to pass data from one command straight into another.

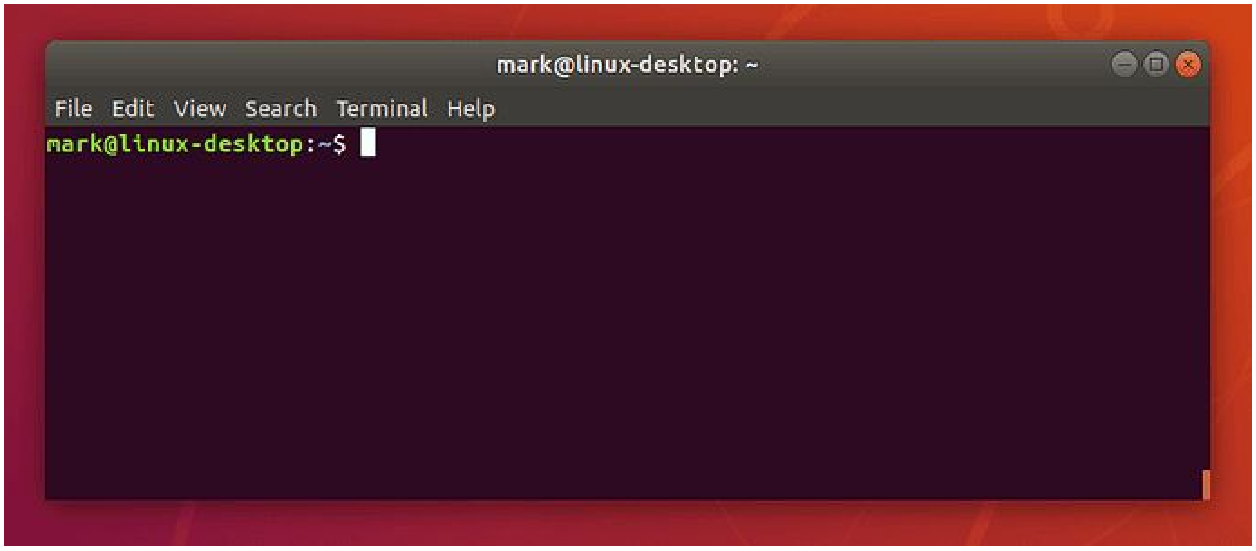
Modern Linux system you're most likely to be using a shell called `bash`.

Opening a terminal

"terminal", "command", "prompt" or "shell".



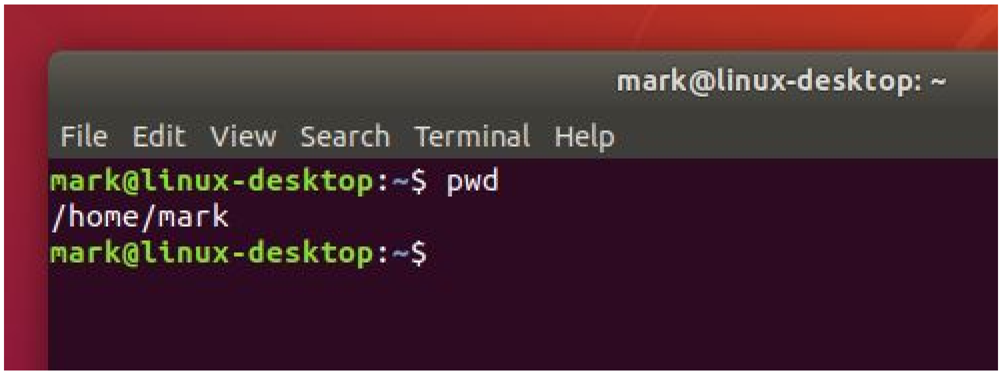
Other versions of Linux, or other flavours of Ubuntu. Most Linux systems use the same default keyboard shortcut to start it: **Ctrl-Alt-T.**



```
mark@linux-desktop:~$
```

`Pwd [Present working directory]`

You should see a directory path printed out (probably something like `/home/YOUR_USERNAME`)



```
mark@linux-desktop:~$ pwd
/home/mark
mark@linux-desktop:~$
```

A sense of location

What's a *working directory*?

One important concept to understand is that the shell has a notion of a **default location in which any file operations will take place**. This is its working directory. If you try to create new files or directories, view existing files, or even delete them, the shell will assume you're looking for them in the current working directory unless you take steps to specify otherwise.

`cd` command, an abbreviation for ‘**change directory**’. Try typing the following:

```
cd /
```

```
pwd
```

Note that the directory separator is a forward slash ("/"),Too many roots Beware: although the “/” directory is sometimes referred to as the *root* directory.

root is also the name that has been used for the **superuser.**

The superuser, as the name suggests, **has more powers than a normal user**, From the root directory, the following command will move you into “home” directory (which is an immediate subdirectory of “/”):

```
cd home
pwd
```

To go up to the parent directory, in this case back to “/”, use the special syntax of two dots (`..`) when changing directory (note the space between `cd` and `..`)

```
cd ..
pwd
```

Typing `cd` on its own is a quick shortcut to get back to your home directory:

```
cd
```

```
pwd
```

You can also use `..` more than once if you have to **move up through multiple levels of parent directories**:

```
cd ../..
```

```
pwd
```

The **path** we used means “**starting from the working directory, move to the parent / from that new location move to the parent again**”.

So if we wanted to go straight from our home directory to the “etc” directory (which is directly inside the root of the file system), we could use this approach:

```
cd

pwd


cd ../../etc

pwd
```

#Very Important concept

Relative and absolute paths

Consider trying to `cd` into the “etc” folder. If you’re already in the root directory that will work fine:

```
cd /

pwd

cd etc

pwd
```

But what if you’re in your home directory?

```
cd

pwd

cd etc

pwd
```

You’ll see an error saying “No such file or directory”

#Note: using `..` will have different effects depending on where you start from. The path only makes sense *relative* to your working directory.

run `cd` on its own to go straight to your home directory.

`cd /` to switch to the root directory. In fact any path that starts with a forward slash is an *absolute* path.

```
cd

pwd

cd /etc

pwd
----------------------------------------------------------
```

`whoami` command will remind you of your username, in case you’re not sure:

```
whoami
cd /home/USERNAME/Desktop
pwd
```

Shortcut which works as an absolute path.

- Using “/” at the start of your path means “starting from the root directory”.
- Using the tilde character ("~") at the start of your path similarly means “starting from my home directory”.

```
cd ~
pwd
```

```
cd ~/Desktop
pwd
```

Creating folders and files

```
mkdir /tmp/tutorial
```

```
cd /tmp/tutorial
```

`mkdir` is short for ‘make directory’.

let’s create a few subdirectories:

```
mkdir dir1 dir2 dir3
```

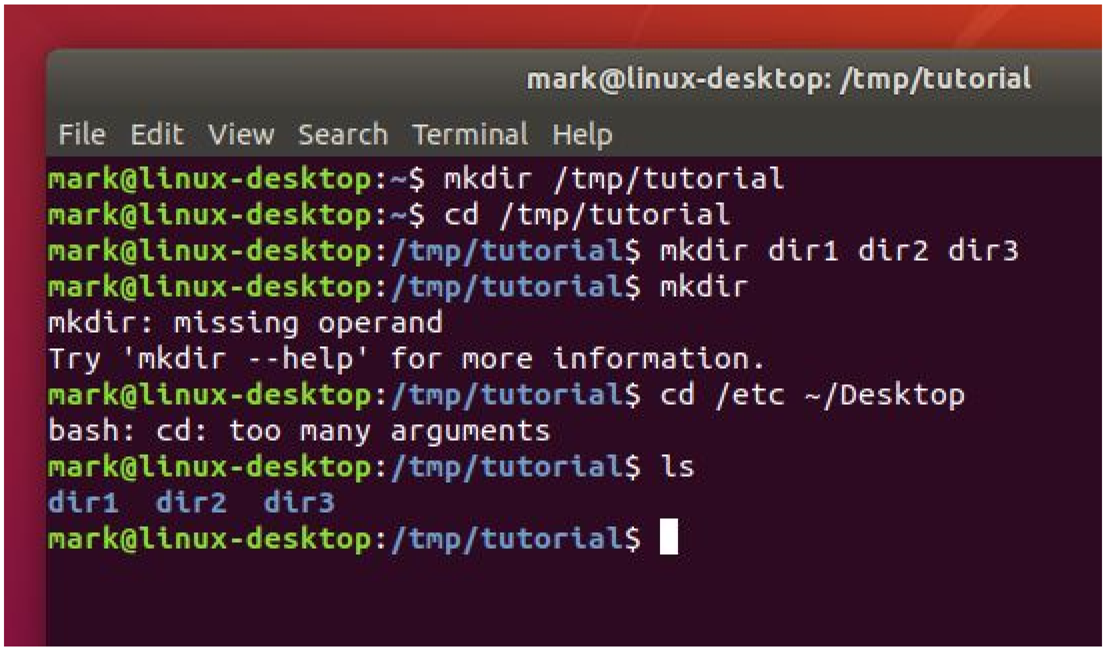
Note: The `mkdir` command expects at least one argument, whereas the `cd` command can work with zero or one, but no more. See what happens when you try to pass the wrong number of parameters to a command:

```
mkdir
```

```
cd /etc ~/Desktop
```

Back to our new directories. The command above will have created three new subdirectories inside our folder. Let’s take a look at them with the `ls` (list) command:

```
ls
```


mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help

```
mark@linux-desktop:~$ mkdir /tmp/tutorial
mark@linux-desktop:~$ cd /tmp/tutorial
mark@linux-desktop:/tmp/tutorial$ mkdir dir1 dir2 dir3
mark@linux-desktop:/tmp/tutorial$ mkdir
mkdir: missing operand
Try 'mkdir --help' for more information.
mark@linux-desktop:/tmp/tutorial$ cd /etc ~/Desktop
bash: cd: too many arguments
mark@linux-desktop:/tmp/tutorial$ ls
dir1  dir2  dir3
mark@linux-desktop:/tmp/tutorial$ 
```

Notice that `mkdir` created all the folders in one directory. It *didn't* create *dir3* inside *dir2* inside *dir1*, or any other nested structure. But sometimes it's handy to be able to do exactly that, and `mkdir` does have a way:

```
mkdir -p dir4/dir5/dir6
```

```
ls
```

This time you'll see that only *dir4* has been added to the list, because *dir5* is inside it, and *dir6* is inside that.

```
cd dir4
```

```
ls
```

```
cd dir5
```

```
ls
```

```
cd ../..
```

The “-p” that we used is called an *option* or a *switch* (in this case it means “create the parent directories, too”). Options are used to modify the way in which a command operates, allowing a single command to behave in a variety of different ways.

```
# Home assignment - Try yourself

mkdir --parents --verbose dir4/dir5

mkdir -p --verbose dir4/dir5

mkdir -p -v dir4/dir5

mkdir -pv dir4/dir5
```

Now we know how to create multiple directories just by passing them as separate arguments to the `mkdir` command. But suppose we want to create a directory with a space in the name?

```
mkdir another folder

ls
```

You probably didn’t even need to type that one in to guess what would happen: two new folders, one called *another* and the other called *folder*. If you want to

Enter the following commands to try out different ways to create folders with spaces in the name:

```
mkdir "folder 1"

mkdir 'folder 2'

mkdir folder\ 3

mkdir "folder 4" "folder 5"

mkdir -p "folder 6"/"folder 7"

ls
```

Tend to stick to letters and numbers, and use underscores ("_") or hyphens ("-") instead of spaces.

Creating files using redirection

Capture the output of that command as a text file that we can look at or manipulate further. We need to do is to add the greater-than character (">") to the end of our command line, followed by the name of the file to write to:

```
ls > output.txt
```

This time there's nothing printed to the screen, because the output is being redirected to our file instead. If you just run `ls` on its own you should see that the *output.txt* file has been created. We can use the `cat` command to look at its content:

```
cat output.txt
```

Let's look at another command, **`echo`**:

```
echo "This is a test"
```

Yes, `echo` just prints its arguments back out again (hence the name).

```
echo "This is a test" > test_1.txt

echo "This is a second test" > test_2.txt

echo "This is a third test" > test_3.txt

ls
```

`cat` is more than just a file viewer - its name comes from '**concatenate**', meaning "to **link together**".

If you pass more than one filename to `cat` it will output each of them, one after the other, as a single block of text:

```
cat test_1.txt test_2.txt test_3.txt
```

A question mark ("?") can be used to indicate "any single character" within the file name.

An asterisk ("*") can be used to indicate "zero or more characters".

These are sometimes referred to as "wildcard" characters. A couple of examples might help, the following commands all do the same thing:

```
cat test_1.txt test_2.txt test_3.txt
```

```
cat test_?.txt
```

```
cat test_*
```

```
cat t* >> combined.txt
```

```
echo "I've appended a line!" >> combined.txt
```

```
cat combined.txt
```

In order to see the whole file we now need to use a different program, called a *pager* (because it displays your file one “page” at a time). The standard pager of old was called `more`, because it puts a line of text at the bottom of each page that says “–More–” to indicate that you haven’t read everything yet. These days there’s a far better pager that you should use instead: because it replaces `more`, the programmers decided to call it `less`.

```
less combined.txt
```

A note about case

Unix systems are case-sensitive, that is, they consider “A.txt” and “a.txt” to be two different files. If you were to run the following lines you would end up with three files:

```
echo "Lower case" > a.txt
```

```
echo "Upper case" > A.TXT
```

```
echo "Mixed case" > A.txt
```

Note:

`cd ..` to move the working directory back again.

```
ls dir1
```

Now suppose it turns out that file shouldn't be in *dir1* after all. Let's move it back to the working directory. We could `cd` into *dir1* then use `mv combined.txt ..` to say "move *combined.txt* into the parent directory". But we can use another path shortcut to avoid changing directory at all. In the same way that two dots (`..`) represents the parent directory, so a single dot (`.`) can be used to represent the current working directory.

(note the space before the dot, there are *two* parameters being passed to `mv`):

```
mv dir1/*
```

#Try Yourself ..

The `mv` command also lets us move more than one file at a time. If you pass more than two arguments, the last one is taken to be the destination directory and the others are considered to be files (or directories) to move. Let's use a single command to move *combined.txt*, all our *test_n.txt* files and *dir3* into *dir2*. There's a bit more going on here, but if you look at each argument at a time you should be able to work out what's happening:

```
mv combined.txt test_* dir3 dir2
```

```
ls
```

```
ls dir2
```

With *combined.txt* now moved into *dir2*, what happens if we decide it's in the wrong place again? Instead of *dir2* it should have been put in *dir6*, which is the one that’s inside *dir5*, which is in *dir4*. With what we now know about paths, that’s no problem either:

```
mv dir2/combined.txt dir4/dir5/dir6
```

```
ls dir2
```

```
ls dir4/dir5/dir6
```

Notice how our `mv` command let us move the file from one directory into another, even though our working directory is something completely different. This is a powerful property of the command line: no matter where in the file system you are, it’s still possible to operate on files and folders in totally different locations.

Since we seem to be using (and moving) that file a lot, perhaps we should keep a copy of it in our working directory. Much as the `mv` command moves files, so the `cp` command copies them (again, note the space before the dot):

```
cp dir4/dir5/dir6/combined.txt .
```

```
ls dir4/dir5/dir6
```

```
ls
```

Great! Now let’s create another copy of the file, in our working directory but with a different name. We can use the `cp` command again, but instead of giving it a directory path as the last argument, we’ll give it a new file name instead:

```
cp combined.txt backup_combined.txt
```

```
ls
```

That’s good, but perhaps the choice of backup name could be better. Why not rename it so that it will always appear next to the original file in a sorted list. The traditional Unix command line handles a rename as though you’re *moving* the file from one name to another, so our old friend `mv` is the command to use. In this case you just specify two arguments: the file you want to rename, and the new name you wish to use.

```
mv backup_combined.txt combined_backup.txt
```

```
ls
```

This also works on directories, giving us a way to sort out those difficult ones with spaces in the name that we created earlier. To avoid re-typing each command after the first, use the Up Arrow to pull up the previous command in the history. You can then edit the command before you run it by moving the cursor left and right with the arrow keys, and removing the character to the left with Backspace or the one the cursor is on with Delete. Finally, type the new character in place, and press Enter or Return to run the command once you’re finished. Make sure you change both appearances of the number in each of these lines.

```
mv "folder 1" folder_1
```

```
mv "folder 2" folder_2
```

```
mv "folder 3" folder_3
```

```
mv "folder 4" folder_4
```

```
mv "folder 5" folder_5
mv "folder 6" folder_6
```

```
ls
```

Deleting files and folders

Warning

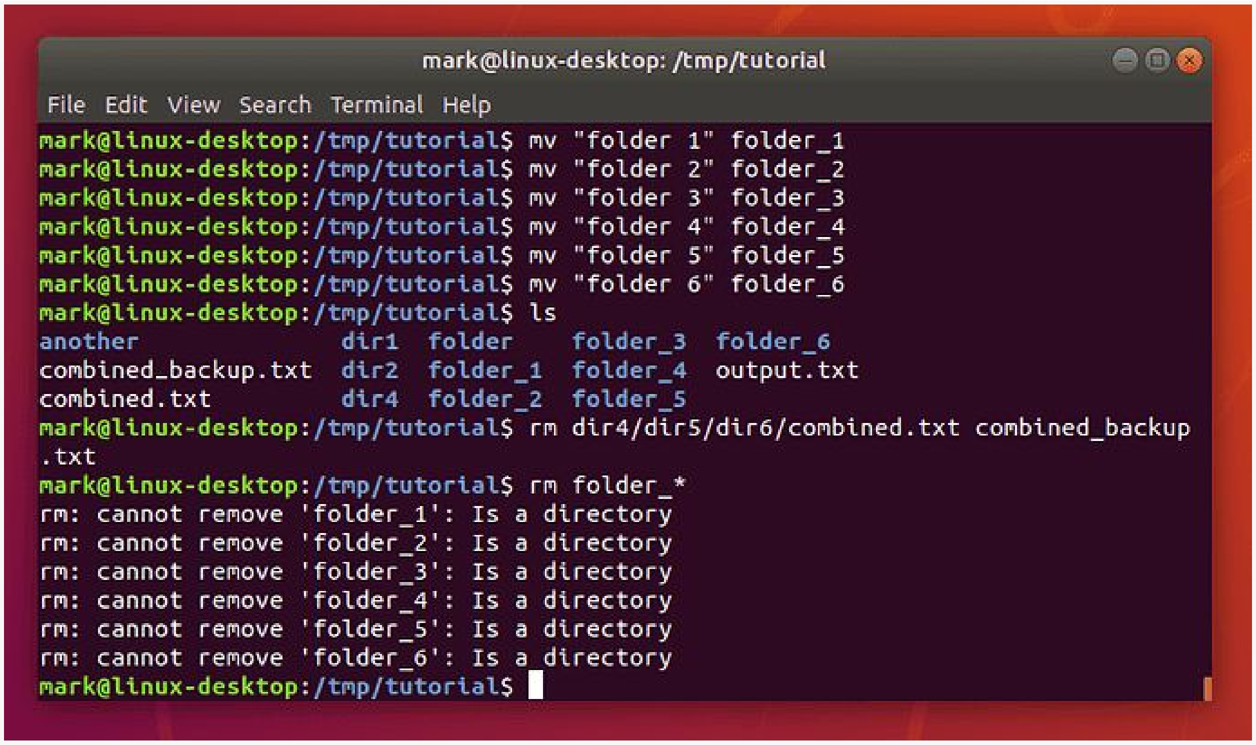
In this next section we’re going to start deleting files and folders. To make absolutely certain that you don’t accidentally delete anything in your home folder, use the `pwd` command to double-check that you’re still in the */tmp/tutorial* directory before proceeding.

Now we know how to move, copy and rename files and directories. Given that these are just test files, however, perhaps we don’t really need three different copies of *combined.txt* after all. Let’s tidy up a bit, using the `rm` (remove) command:

```
rm dir4/dir5/dir6/combined.txt combined_backup.txt
```

Perhaps we should remove some of those excess directories as well:

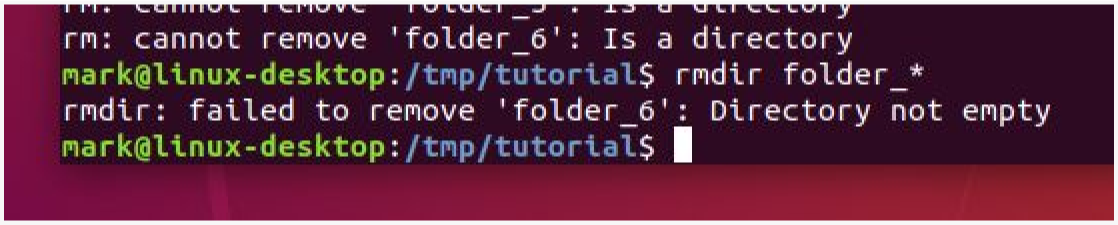
```
rm folder_*
```


mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help

```
mark@linux-desktop:/tmp/tutorial$ mv "folder 1" folder_1
mark@linux-desktop:/tmp/tutorial$ mv "folder 2" folder_2
mark@linux-desktop:/tmp/tutorial$ mv "folder 3" folder_3
mark@linux-desktop:/tmp/tutorial$ mv "folder 4" folder_4
mark@linux-desktop:/tmp/tutorial$ mv "folder 5" folder_5
mark@linux-desktop:/tmp/tutorial$ mv "folder 6" folder_6
mark@linux-desktop:/tmp/tutorial$ ls
another              dir1  folder    folder_3  folder_6
combined_backup.txt  dir2  folder_1  folder_4  output.txt
combined.txt         dir4  folder_2  folder_5
mark@linux-desktop:/tmp/tutorial$ rm dir4/dir5/dir6/combined.txt combined_backup
.txt
mark@linux-desktop:/tmp/tutorial$ rm folder_*
rm: cannot remove 'folder_1': Is a directory
rm: cannot remove 'folder_2': Is a directory
rm: cannot remove 'folder_3': Is a directory
rm: cannot remove 'folder_4': Is a directory
rm: cannot remove 'folder_5': Is a directory
rm: cannot remove 'folder_6': Is a directory
mark@linux-desktop:/tmp/tutorial$ 
```

There’s an `rmdir` (remove directory) command that will do the job for us instead:

`rmdir folder_*`



```
rm: cannot remove 'folder_6': Is a directory
mark@linux-desktop:/tmp/tutorial$ rmdir folder_*
rmdir: failed to remove 'folder_6': Directory not empty
mark@linux-desktop:/tmp/tutorial$ 
```

Well that’s a little better, but there’s still an error. If you run `ls` you’ll see that most of the folders have gone, but *folder_6* is still hanging around. As you may recall, *folder_6* still has a *folder 7* inside it, and `rmdir` will only delete empty folders.

Prevent you from accidentally deleting a folder full of files when you didn't mean to.

In this case, however, we *do* mean to. The addition of options to our `rm` or `rmdir` commands will let us perform dangerous actions without the aid of a safety net! In the case of `rmdir` we can add a `-p` switch to tell it to also remove the parent directories. Think of it as the counterpoint to `mkdir -p`. So if you were to run `rmdir -p dir1/dir2/dir3` it would first delete *dir3*, then *dir2*, then finally delete *dir1*. It still follows the normal `rmdir` rules of only deleting empty directories though, so if there was also a file in *dir1*, for example, only *dir3* and *dir2* would get removed.

A more common approach, when you're really, *really*, really sure you want to delete a whole directory and anything within it, is to tell `rm` to work recursively by using the `-r` switch, in which case it will happily delete folders as well as files. With that in mind, here's the command to get rid of that pesky *folder_6* and the subdirectory within it:

```
rm -r folder_6
```

```
ls
```

Remember: although `rm -r` is quick and convenient, it's also dangerous. It's safest to explicitly delete files to clear out a directory, then `cd ..` to the parent before using `rmdir` to remove it.

Important Warning

Unlike graphical interfaces, `rm` doesn't move files to a folder called "trash" or similar. Instead it deletes them totally, utterly and irrevocably.

Pipelining

The `wc` (word count) command can tell us that, using the `-l` switch to tell it we only want the **line count** (it can also do character counts and, as the name suggests, word counts):

```
wc -l combined.txt
```

Similarly, if you wanted to know how many files and folders are in your home directory, and then tidy up after yourself, you could do this:

```
ls ~ > file_list.txt

wc -l file_list.txt

rm file_list.txt
```

That method works, but creating a temporary file to hold the output from `ls` only to delete it two lines later seems a little excessive. Fortunately the Unix command line provides a shortcut that avoids you having to create a temporary file, by taking the output from one command (referred to as *standard output* or *STDOUT*) and feeding it directly in as the input to another command (*standard input* or *STDIN*). It's as though you've connected a pipe between one command's output and the next command's input, so much so that this process is actually referred to as *piping* the data from one command to another. Here's how to pipe the output of our `ls` command into `wc`:

```
ls ~ | wc -l
```

Notice that there's no temporary file created, and no file name needed. Pipes operate entirely in memory, and most Unix command line tools will expect to receive input from a pipe if you don't specify a file for them to work on. Looking at the line above, you can see that it's two commands, `ls ~` (list the contents of the home directory) and `wc -l` (count the lines), separated by a vertical bar character ("|"). This process of piping one command into another is so commonly used that the character itself is often referred to as the *pipe* character, so if you see that term you now know it just means the vertical bar.

Note that the spaces around the pipe character aren't important, we've used them for clarity, but the following command works just as well, this time for telling us how many items are in the */etc* directory:

```
ls /etc|wc -l
```

Phew! That's quite a few files. If we wanted to list them all it would clearly fill up more than a single screen. As we discovered earlier, when a command produces a lot of output, it's better to use `less` to view it, and that advice still applies when using a pipe (remember, press q to quit):

```
ls /etc | less
```

Going back to our own files, we know how to get the number of lines in *combined.txt*, but given that it was created by concatenating the same files multiple times, I wonder how many unique lines there are? Unix has a command, `uniq`, that will only output unique lines in the file. So we need to `cat` the file out and pipe it through `uniq`. But all we want is a line count, so we need to use `wc` as well. Fortunately the command line doesn't limit you to a single pipe at a time, so we can continue to chain as many commands as we need:

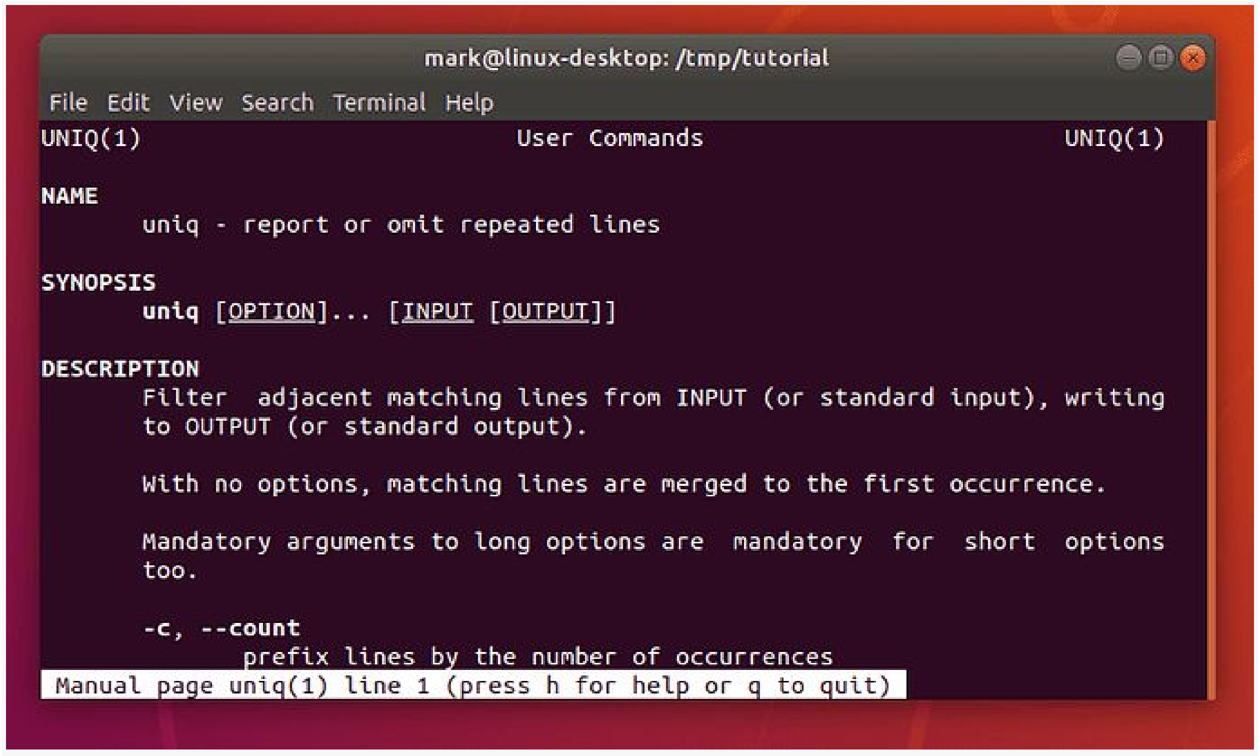
```
cat combined.txt | uniq | wc -l
```

That line probably resulted in a count that's pretty close to the total number of lines in the file, if not exactly the same. Surely that can't be right? Lop off the last pipe to see the output of the command for a better idea of what's happening. If your file is very long, you might want to pipe it through `less` to make it easier to inspect:

```
cat combined.txt | uniq | less
```

It appears that very few, if any, of our duplicate lines are being removed. To understand why, we need to look at the documentation for the `uniq` command. Most command line tools come with a brief (and sometimes not-so-brief) instruction manual, accessed through the `man` (manual) command. The output is automatically piped through your pager, which will typically be `less`, so you can move back and forth through the output, then press q when you're finished:

```
man uniq
```


mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help

```
UNIQ(1)                          User Commands                         UNIQ(1)

NAME
       uniq - report or omit repeated lines

SYNOPSIS
       uniq [OPTION]... [INPUT [OUTPUT]]

DESCRIPTION
       Filter  adjacent matching lines from INPUT (or standard input), writing
       to OUTPUT (or standard output).

       With no options, matching lines are merged to the first occurrence.

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

       -c, --count
              prefix lines by the number of occurrences
 Manual page uniq(1) line 1 (press h for help or q to quit)
```

Because this type of documentation is accessed via the `man` command, you'll hear it referred to as a "man page", as in "check the man page for more details". The format of man pages is often terse, think of them more as a quick overview of a command than a full tutorial. They're often highly technical, but you can usually skip most of the content and just look for the details of the option or argument you're using.

The `uniq` man page is a typical example in that it starts with a brief one-line description of the command, moves on to a synopsis of how to use it, then has a detailed description of each option or parameter. But whilst man pages are invaluable, they can also be inpenetrable. They're best used when you need a reminder of a particular switch or parameter, rather than as a general resource for learning how to use the command line. Nevertheless, the first line of the DESCRIPTION section for `man uniq` does answer the question as to why duplicate lines haven't been removed: it only works on *adjacent* matching lines.

The question, then, is how to rearrange the lines in our file so that duplicate entries are on adjacent lines. If we were to sort the contents of the file alphabetically, that would do the trick. Unix offers a `sort` command to do exactly that. A quick check of `man sort` shows that we can pass a file name directly to the command, so let's see what it does to our file:

```
sort combined.txt | less
```

You should be able to see that the lines have been reordered, and it's now suitable for piping straight into `uniq`. We can finally complete our task of counting the unique lines in the file:

```
sort combined.txt | uniq | wc -l
```

As you can see, the ability to pipe data from one command to another, building up long chains to manipulate your data, is a powerful tool, as well as reducing the need for temporary files, and saving you a *lot* of typing. For this reason you'll see it used quite often in command lines. A long chain of commands might look intimidating at first, but remember that you can break even the longest chain down into individual commands (and look at their man pages) to get a better understanding of what it's doing.

Many manuals

Most Linux command line tools include a man page. Try taking a brief look at the pages for some of the commands you've already encountered: `man ls`, `man cp`, `man rmdir` and so on. There's even a man page for the man program itself, which is accessed using `man man`, of course.

The command line and the superuser

One good reason for learning some command line basics is that instructions online will often favour the use of shell commands over a graphical interface. Where those instructions require changes to your machine that go beyond modifying a few files in your home directory, you'll inevitably be faced with commands that need to be run as the machine's administrator (or *superuser* in Unix parlance). Before you start running arbitrary commands you find in some dark corner of the internet, it's worth understanding the implications of running as an administrator, and how to spot those instructions that require it, so you can better gauge whether they're safe to run or not.

The superuser is, as the name suggests, a user with super powers. In older systems it was a real user, with a real username (almost always "root") that you could log in as if you had the password. As for those super powers: *root* can modify or delete any file in any directory on the system, regardless of who owns them; *root* can rewrite firewall rules or start network services that could potentially open the machine up to an attack; *root* can shutdown the machine even if other people are still using it. In short, *root* can do just about *anything*, skipping easily round the safeguards that are usually put in place to stop users from overstepping their bounds.

Of course a person logged in as *root* is just as capable of making mistakes as anyone else. The annals of computing history are filled with tales of a mistyped command deleting the entire file system or killing a vital server. Then there's the possibility of a malicious attack: if a user is logged in as *root* and leaves their desk then it's not too tricky for a disgruntled colleague to hop on their machine and wreak havoc. Despite that, human nature being what it is, many administrators over the years have been guilty of using *root* as their main, or only, account.

Don’t use the root account

If anyone asks you to enable the *root* account, or log in as *root*, be very suspicious of their intentions.

In an effort to reduce these problems many Linux distributions started to encourage the use of the `su` command. This is variously described as being short for ‘superuser’ or ‘switch user’, and allows you to change to another user on the machine without having to log out and in again. When used with no arguments it assumes you want to change to the *root* user (hence the first interpretation of the name), but you can pass a username to it in order to switch to a specific user account (the second interpretation). By encouraging use of `su` the aim was to persuade administrators to spend most of their time using a normal account, only switch to the superuser account when they needed to, and then use the `logout` command (or Ctrl-D shortcut) as soon as possible to return to their user-level account.

By minimising the amount of time spent logged in as *root*, the use of `su` reduces the window of opportunity in which to make a catastrophic mistake. Despite that, human nature being what it is, many administrators have been guilty of leaving long-running terminals open in which they’ve used `su` to switch to the *root* account. In that respect `su` was only a small step forward for security.

Don’t use su

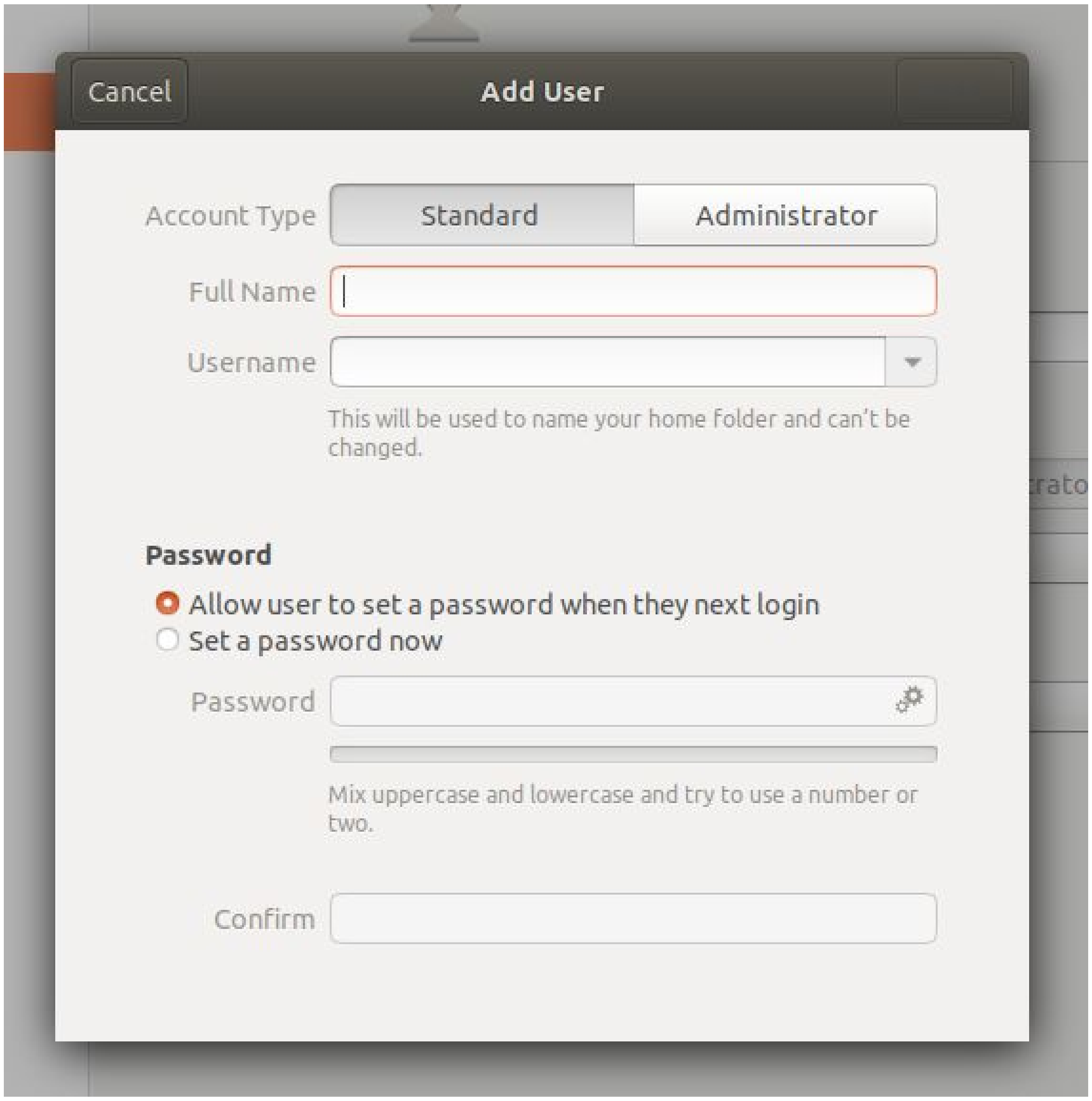
If anyone asks you to use `su`, be wary. If you’re using Ubuntu the *root* account is disabled by default, so `su` with no parameters won’t work. But it’s still not worth taking the risk, in case the account has been enabled without you

realising. If you are asked to use `su` with a username then (if you have the password) you will have access to all the files of that user, and could accidentally delete or modify them.
When using `su` your entire terminal session is switched to the other user. Commands that don't need *root* access, something as mundane as `pwd` or `ls`, would be run under the auspices of the superuser, increasing the risk of a bug in the program causing major problems. Worse still, if you lose track of which user you're currently operating as, you might issue a command that is fairly benign when run as a user, but which could destroy the entire system if run as *root*.

Better to disable the *root* account entirely and then, instead of allowing long-lived terminal sessions with dangerous powers, require the user to specifically request superuser rights on a per-command basis. The key to this approach is a command called `sudo` (as in "switch user and do this command").

`sudo` is used to prefix a command that has to be run with superuser privileges. A configuration file is used to define which users can use `sudo`, and which commands they can run. When running a command like this, the user is prompted for *their own* password, which is then cached for a period of time (defaulting to 15 minutes), so if they need to run multiple superuser-level commands they don't keep getting continually asked to type it in.

On a Ubuntu system the first user created when the system is installed is considered to be the superuser. When adding a new user there is an option to create them as an administrator, in which case they will also be able to run superuser commands with `sudo`. In this screenshot of Ubuntu 18.04 you can see the option at the top of the dialog:


Cancel

Add User

Account Type [✓ Standard] Administrator

Full Name

Username

This will be used to name your home folder and can't be changed.

Password

[✓ Allow user to set a password when they next login]

Set a password now

Password

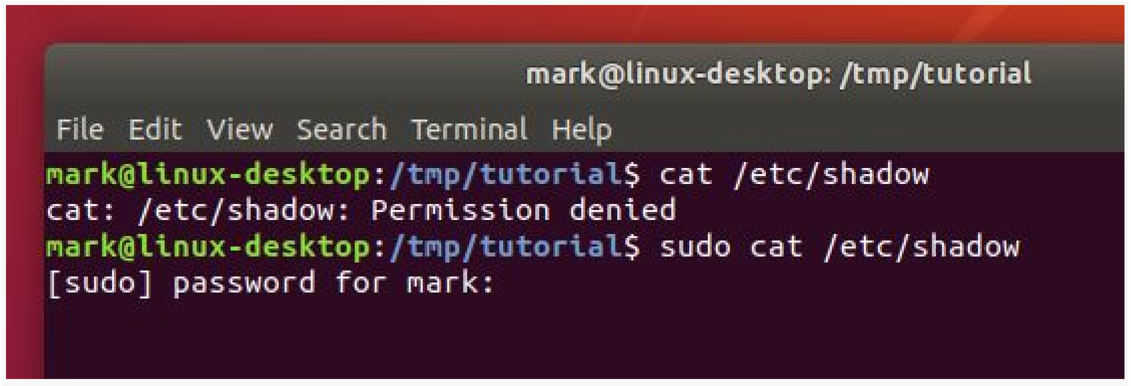
Mix uppercase and lowercase and try to use a number or two.

Confirm

Assuming you're on a Linux system that uses `sudo`, and your account is configured as an administrator, try the following to see what happens when you try to access a file that is considered sensitive (it contains encrypted passwords):

```
cat /etc/shadow
```

```
sudo cat /etc/shadow
```


mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help

```
mark@linux-desktop:/tmp/tutorial$ cat /etc/shadow
cat: /etc/shadow: Permission denied
mark@linux-desktop:/tmp/tutorial$ sudo cat /etc/shadow
[sudo] password for mark:
```

If you enter your password when prompted you should see the contents of the `/etc/shadow` file. Now clear the terminal by running the `reset` command, and run `sudo cat /etc/shadow` again. This time the file will be displayed without prompting you for a password, as it's still in the cache.

Be careful with sudo

If you are instructed to run a command with `sudo`, make sure you understand what the command is doing before you continue. Running with `sudo` gives that command all the same powers as a superuser. For example, a software publisher's site might ask you to download a file and change its permissions, then use `sudo` to run it. Unless you know exactly what the file is doing, you're opening up a hole through which malware could potentially be installed onto your system. `sudo` may only run one command at a time, but that command could itself run many others. Treat any new use of `sudo` as being just as dangerous as logging in as *root*.

For instructions targeting Ubuntu, a common appearance of `sudo` is to install new software onto your system using the `apt` or `apt-get` commands. If the instructions require you to first add a new software repository to your system, using the `apt-add-repository` command, by editing files in `/etc/apt`, or by using a "PPA" (Personal Package Archive), you should be careful as these sources are not curated by Canonical. But often the instructions just require you to install software from the standard repositories, which should be safe.

Installing new software

There are lots of different ways to install software on Linux systems. Installing directly from your distro's official software repositories is the safest option, but sometimes the application or version you want simply isn't available that way. When installing via any other mechanism, make sure you're getting the files from an official source for the project in question.

Indications that files are coming from outside the distribution's repositories include (but are not limited to) the use of any of the following commands: `curl`, `wget`, `pip`, `npm`, `make`, or any instructions that tell you to change a file's permissions to make it executable.

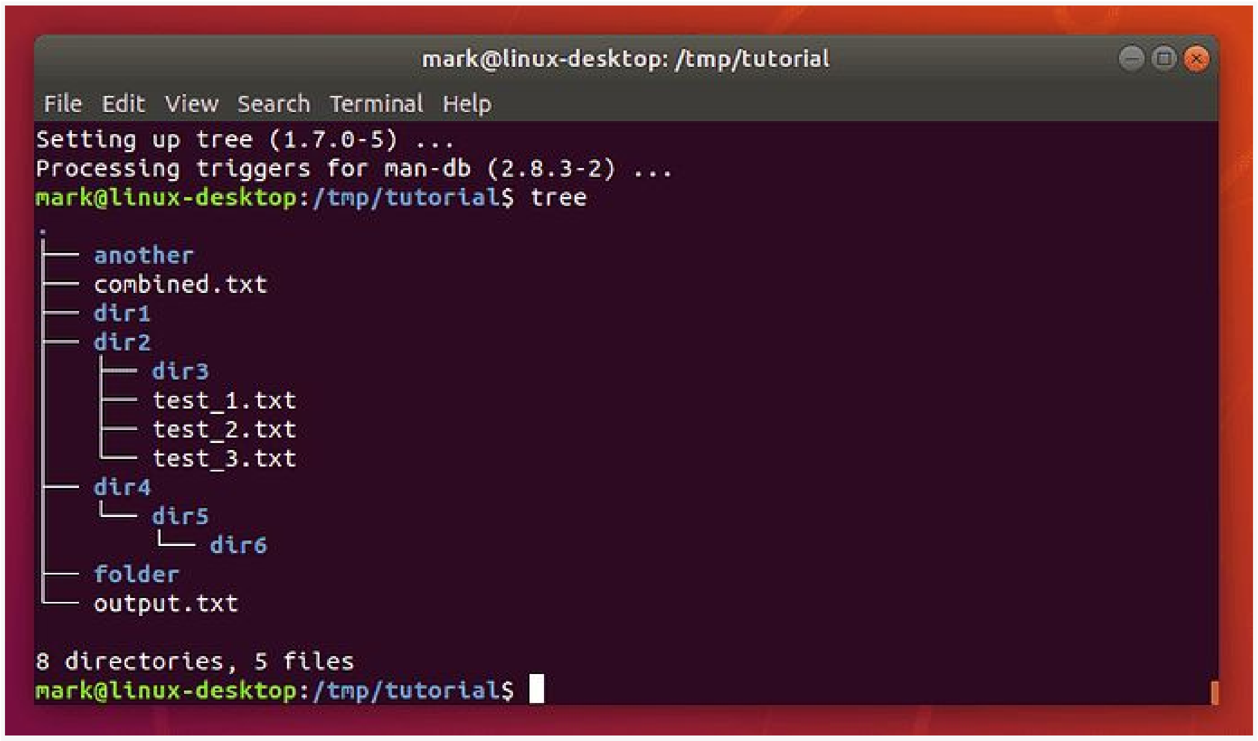
Increasingly, Ubuntu is making use of "snaps", a new package format which offers some security improvements by more closely confining programs to stop them accessing parts of the system they don't need to. But some options can reduce the security level so, if you're asked to run `snap install` with any parameters other than the name of the snap, it's worth checking exactly what the command is trying to do.
Let's install a new command line program from the standard Ubuntu repositories to illustrate this use of `sudo`:

```
sudo apt install tree
```

Once you've provided your password the `apt` program will print out quite a few lines of text to tell you what it's doing. The `tree` program is only small, so it shouldn't take more than a minute or two to download and install for most users. Once you are returned to the normal command line prompt, the program is installed and ready to use. Let's run it to get a better overview of what our collection of files and folders looks like:

```
cd /tmp/tutorial
```

```
tree
```


mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help

```
Setting up tree (1.7.0-5) ...
Processing triggers for man-db (2.8.3-2) ...
mark@linux-desktop:/tmp/tutorial$ tree
.
├── another
├── combined.txt
├── dir1
├── dir2
│   ├── dir3
│   ├── test_1.txt
│   ├── test_2.txt
│   └── test_3.txt
├── dir4
│   └── dir5
│       └── dir6
├── folder
└── output.txt

8 directories, 5 files
mark@linux-desktop:/tmp/tutorial$
```

Going back to the command that actually installed the new program (`sudo apt install tree`) it looks slightly different to those you've see so far. In practice it works like this:

1. The `sudo` command, when used without any options, will assume that the first parameter is a command for it to run with superuser privileges. Any other parameters will be passed directly to the new command. `sudo`'s switches all start with one or two hyphens and must immediately follow the `sudo` command, so there can be no confusion

about whether the second parameter on the line is a command or an option.

2. The command in this case is `apt`. Unlike the other commands we've seen, this isn't working directly with files. Instead it expects its first parameter to be an instruction to perform (`install`), with the rest of the parameters varying based on the instruction.
3. In this case the `install` command tells `apt` that the remainder of the command line will consist of one or more package names to install from the system's software repositories. Usually this will add new software to the machine, but packages could be any collection of files that need to be installed to particular locations, such as fonts or desktop images.

You can put `sudo` in front of any command to run it as a superuser, but there's rarely any need to. Even system configuration files can often be viewed (with `cat` or `less`) as a normal user, and only require *root* privileges if you need to edit them.

Beware of sudo su

One trick with `sudo` is to use it to run the `su` command. This will give you a *root* shell even if the *root* account is disabled. It can be useful when you need to run a series of commands as the superuser, to avoid having to prefix them all with `sudo`, but it opens you up to exactly the same kind of problems that were described for `su` above. If you follow any instructions that tell you to run `sudo su`, be aware that every command after that will be running as the *root* user.

In this section you've learnt about the dangers of the *root* account, and how modern Linux systems like Ubuntu try to reduce the risk of danger by using `sudo`. But *any* use of superuser powers should be considered carefully. When following instructions you find online you should now be in a better position to spot those commands that might require greater scrutiny.

8. Hidden files

Before we conclude this tutorial it’s worth mentioning *hidden files* (and folders). These are commonly used on Linux systems to store settings and configuration data, and are typically hidden simply so that they don’t clutter the view of your own files. There’s nothing special about a hidden file or folder, other than it’s name: simply starting a name with a dot (".") is enough to make it disappear.

```
cd /tmp/tutorial

ls

mv combined.txt .combined.txt

ls
```

You can still work with the hidden file by making sure you include the dot when you specify its file name:

```
cat .combined.txt

mkdir .hidden

mv .combined.txt .hidden

less .hidden/.combined.txt
```

If you run `ls` you’ll see that the `.hidden` directory is, as you might expect, hidden. You can still list its contents using `ls .hidden`, but as it only contains a single file which is, itself, hidden you won't get much output. But you can use the `-a` (show all) switch to `ls` to make it show everything in a directory, including the hidden files and folders:

```
ls

ls -a

ls .hidden

ls -a .hidden
```

Notice that the shortcuts we used earlier, `.` and `..`, also appear as though they're real directories.

As for our recently installed `tree` command, that works in a similar way (except without an appearance by `.` and `..`):

```
tree

tree -a
```

Switch back to your home directory (`cd`) and try running `ls` without and then with the `-a` switch. Pipe the output through `wc -l` to give you a clearer idea of how many hidden files and folders have been right under your nose all this time. These files typically store your personal configuration, and is how Unix systems have always offered the capability to have system-level settings (usually in `/etc`) that can be overridden by individual users (courtesy of hidden files in their home directory).

You shouldn't usually need to deal with hidden files, but occasionally instructions might require you to `cd` into `.config`, or edit some file whose name starts with a dot. At least now you'll understand what's happening, even when you can't easily see the file in your graphical tools.

Cleaning up

We've reached the end of this tutorial, and you should be back in your home directory now (use `pwd` to check, and `cd` to go there if you're not). It's only polite to leave your computer in the same state that we found it in, so as a final step, let's remove the experimental area that we were using earlier, then double-check that it's actually gone:

```
rm -r /tmp/tutorial
```

```
ls /tmp
```

As a last step, let's close the terminal. You can just close the window, but it's better practice to log out of the shell. You can either use the `logout` command, or the Ctrl-D keyboard shortcut. If you plan to use the terminal a lot, memorising Ctrl-Alt-T to launch the terminal and Ctrl-D to close it will soon make it feel like a handy assistant that you can call on instantly, and dismiss just as easily.

[#Detailed Source : https://ubuntu.com/tutorials/command-line-for-beginners#1-overview]