UNIX 101

or "How to feel like a true hack3r"

Tutorial

November 15, 2021

Contents

1 Foreword

1.1 Objective

This tutorial should teach you the basics of working with a terminal. You are not expected to fully assimilate everything that this document is about in details, but you should at least understand the basics of every notion. You can come back to this tutorial whenever you want to refresh your mind.

If you work thoroughfully, at the end of this tutorial, you should know:

- What the syntax of a command is
- A few useful commands
- How to navigate in a filesystem
- How to create, edit and remove files
- What is a packet manager and how to use one
- How to create your own executable scripts

Even thought you may already have a bit of knowledge on the topics we will study, we will start again from the beginning. It will help you assimilate completely the basics and provide you a stronger basis to learn new things.

1.2 Preliminary setup

You are going to need a UNIX-like operating system for this tutorial. A Linux distribution is preferred but a Mac can also do the job (OS X is a variant of FreeBSD).

If your operating system is a Windows, the safest way is for you to start a virtual machine running a Linux distribution although you could also run the Windows Subsystem for Linux[1]. It allows one to run the latest versions of Ubuntu on a Windows 10 machine, but it is still new, so at your own risk!

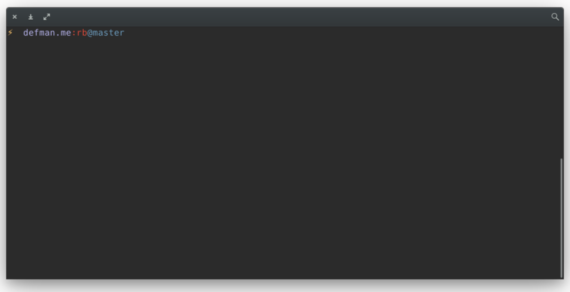
2 Terminal and shell

2.1 A terminal?

To be precise, a *terminal emulator* is a program reading the user's keyboard input.

Inside a terminal runs a *shell*. This is a program that works in pair with a terminal. The shell's job is basically to read the text it is given as input and interpret it into actual actions. Of course, the shell does not understand random text input; it tries to interpret what the user inputs into commands. We will detail the syntax of a command later on.

[1] `https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux`



```
⚡ defman.me:rb@master
```

Figure 1: This is what a terminal looks like

2.2 Opening a terminal

Let us try it now! Open a new terminal. On a fresh Ubuntu install, the shortcut is `Ctrl+T`. On a Mac, you should be able to launch one by first pressing `Cmd+Space` and then typing "Terminal". You should be prompted with a window like the following:

```
nico@nico-VirtualBox:~$
```

This is called the prompt. Typically, the `$` at the end of the line represents the position where you start typing text. From now on, the examples on this tutorial will all start with that `$`.

2.3 Anatomy of a command and manual

First, we will call the command `ls` (list). This program, susprinsigly, allows one to get the *list* of files in the current directory.

```
$ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

That is great! But wait, do not leave yet, there is more. One can alter a command's behavior by adding arguments. See after:

```
$ls /
bin/    home/             lost+found/  root/  sys/      vmlinuz.old@
boot/   initrd.img@       media/       run/   tmp/
cdrom/  initrd.img.old@   mnt/         sbin/  usr/
dev/    lib/              opt/         snap/  var/
etc/    lib64/            proc/        srv/   vmlinuz@
```

Here, the first argument of the command was `/`. Thus, the command `ls` changed its behavior to list the root directory (`/`) instead of the current directory.

Last but not least, arguments can take the form of flags. One can recognize a flag because it begins by a dash (`-`). In the following example, we use the flag `-l`:

```
$ls -l /
total 119
drwxr-xr-x   2 root root  4096 déc.  23 19:02 bin/
drwxr-xr-x   4 root root  3072 déc.  23 19:07 boot/
drwxrwxr-x   2 root root  4096 févr. 26  2018 cdrom/
drwxr-xr-x  22 root root  4400 janv.  4 22:38 dev/
drwxr-xr-x 153 root root 12288 déc.  23 22:52 etc/
drwxr-xr-x   4 root root  4096 févr. 26  2018 home/
lrwxrwxrwx   1 root root    33 déc.  23 19:05 initrd.img -> boot/
   initrd.img-4.15.0-43-generic
lrwxrwxrwx   1 root root    33 déc.  23 19:05 initrd.img.old -> boot/
   initrd.img-4.15.0-33-generic
drwxr-xr-x  25 root root  4096 sept.  5 09:17 lib/
drwxr-xr-x   2 root root  4096 févr. 26  2018 lib64/
drwx------   2 root root 16384 févr. 26  2018 lost+found/
drwxr-xr-x   5 root root  4096 mai   15  2018 media/
drwxr-xr-x   7 root root  4096 juin  20  2018 mnt/
drwxr-xr-x   6 root root  4096 juin   4  2018 opt/
dr-xr-xr-x 290 root root     0 janv.  4 19:41 proc/
drwx------  11 root root  4096 janv.  4 21:49 root/
drwxr-xr-x  34 root root  1100 janv.  4 19:42 run/
drwxr-xr-x   2 root root 12288 déc.  23 19:02 sbin/
drwxr-xr-x   5 root root  4096 sept.  1 19:16 snap/
drwxr-xr-x   2 root root  4096 févr. 15  2017 srv/
dr-xr-xr-x  13 root root     0 janv.  4 19:42 sys/
drwxrwxrwt  15 root root 20480 janv.  5 02:28 tmp/
drwxr-xr-x  12 root root  4096 mars  29  2018 usr/
drwxr-xr-x  14 root root  4096 févr. 15  2017 var/
lrwxrwxrwx   1 root root    30 déc.  23 19:05 vmlinuz -> boot/vmlinuz
   -4.15.0-43-generic
lrwxrwxrwx   1 root root    30 déc.  23 19:05 vmlinuz.old -> boot/
   vmlinuz-4.15.0-33-generic
```

That is quite a lot of information! The option `-l` allows one to get much more information than with a simple `ls`.

To check out the full list of possible arguments you can use with ls, use the command `man ls`. The `man` command gives information about the command given as argument. From now on, this will be your best friend. You can quit the man by pressing `q`. **Before asking for help, always check out the manual**. The answer to your question will, in 95% of the cases, be findable inside.

Also, at the top of every man page is the prototype (*"synopsys"*) of the command. It will help you figure out how you should call it, in which order the arguments can be put and which arguments are mandatory for the command to run.

2.4 Demystifying "commands"

Actually, the term *command* is not accurate.

When one launches the command `ls`, the program at location `/bin/ls` is executed by the shell. Then, in this case, entering in the shell `ls` is the same as entering `/bin/ls`, but faster.

Using `which ls` tells you where the program `ls` is located.

```
$which ls
/bin/ls
```

One simply has to list the content of the directory `/bin` to verify that `ls` is a program that lies there:

```
$ls -l /bin
total 13044
(...)
-rwxr-xr-x 1 root root   56152 mars    2  2017 ln
-rwxr-xr-x 1 root root  111496 sept. 22  2016 loadkeys
-rwxr-xr-x 1 root root   48128 mars   26  2019 login
-rwxr-xr-x 1 root root  453856 juin   27 17:49 loginctl
-rwxr-xr-x 1 root root  105136 mars   21  2019 lowntfs-3g
-rwxr-xr-x 1 root root  126584 mars    2  2017 ls
-rwxr-xr-x 1 root root   77280 oct.  10 11:34 lsblk
(...)
```

As you can see, the directory `/bin` contains the program `ls`!

By the way, `which` is also a program, so entering `which which` in the shell works! Feel free to try.

However, some commands do not depend on the programs on your filesystem. They are contained within the shell tool set, literally "built in" the shell. Thus, we call them... built-in commands[2].

Now, how does `which`, or your shell, knows where to find the program your commands refer to? Well, there is no black magic in UNIX system; the shell does not magically know that the program `ls` lives in `/bin`, for example. There is a special variable that the shell keeps track of: the `PATH`.

```
$echo $PATH # Show the content of PATH variable
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
   games:/usr/local/games
```

It references all of the location where binaries are supposed to sit, separated by a colon (`:`). When the shell encounters a non-builtin command, like `ls`, it will simply check in all of the locations referenced in the path, one by one, if there is a program named as the command. If there is one, it will start it; if not, it will raise an error.

2.5 Why the command line?

Now you may be wondering why bother using fastidious commands to simply show a list of files. It would be way simpler to use a graphical tool like Mac's Finder or Windows's file explorer!

Well, even though it is more complicated to do basic things, the diversity of options shipped with every command allows one to do much more than simply with a graphical user interface (GUI). It is also possible to combine options, widening the range of possibilites. The philosophy of UNIX is for each command to be able to do a limited set of things, but to do it perfectly, with performance in mind.

A few other reasons as to why knowing how to work on a command line is important:

– Many technical actions cannot be done graphically, as a lot of software engineering/sysadmin programs only have a command line interface (CLI)
– Scripts are essential to computer engineering and are nothing but a list of sequential commands to execute
– Thanks to their variety of options, command line programs allow one to run complex tasks in a simple line. Imagine how tiresome it would be to copy only .jpeg files of size <100Kb from one huge directory to another using Window's file manager? With the CLI, it is doable with a simple command or a minimal script

The bottom line is: in computer science, everybody uses the command line. Just like knowing how to code, knowing how to use the command line is essential in the daily job of anyone working in that field.

[2]**This question**'s responses might give you a deeper understanding of the difference between these two types of commands

3 Core filesystem utilities

Now that you have an idea on how to use commands and most importantly on where to find documentation, let us dig into the basics. In this section, you will learn how to navigate in your filesystem and how to play with it. Do not forget to take a look at the manual in case of doubt (type `man` followed by the name of the command you need documentation for).

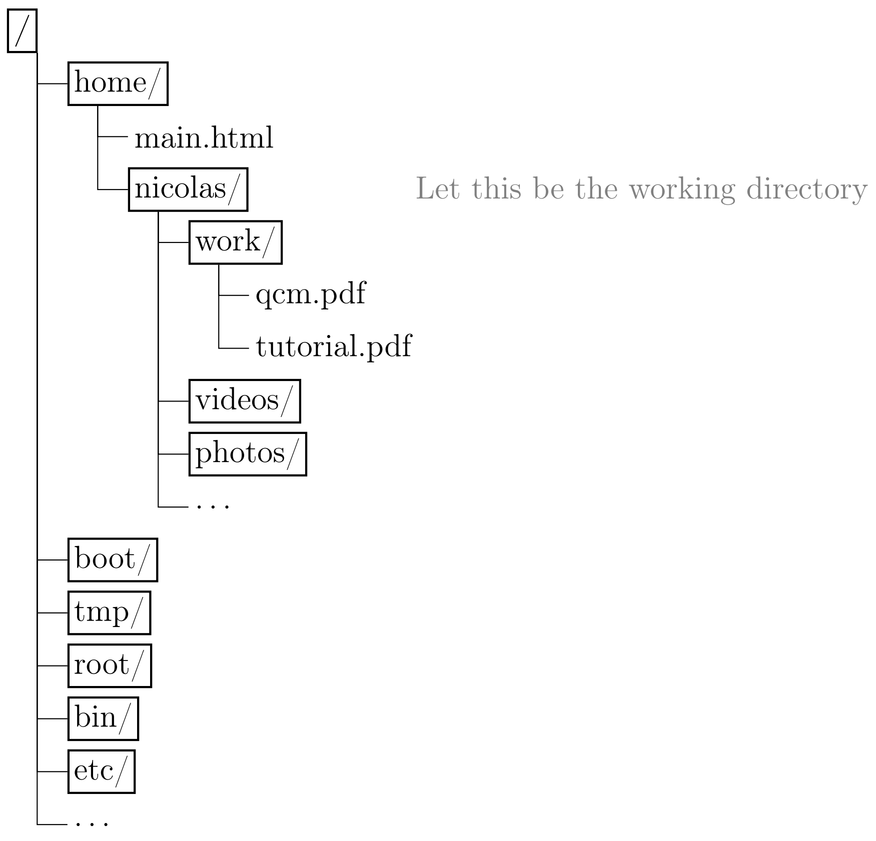
Before we start using commands, please read thoroughfully the next section.

3.1 Absolute and relative paths

The most common use case for shells -and the one we are going to use the most during this tutorial- are interactive-mode shells. When opening a terminal, you are dropped in an interactive shell. When launched in interactive mode, the shell accepts commands, executes them and gives the user control back to run more commands. Additionally, in that mode, the shell maintains a state regarding the filesystem.

The current working directory is hence something that can change depending on commands input by the user. One can "move" from one directory to an other and the shell will keep track of that.

Now, directories and files can be referenced **in regards to the current directory** or **in regards to the root of the filesystem**. The former kind of reference is called **relative** and the latter **absolute**. An absolute path to a file or directory will stay the same, no matter what is the current directory of the shell; a relative one will depend on what is the current directory of the shell. Using absolute paths is often safer to use because you cannot get them wrong; however, they can get long, which makes them tiring to write down. Relative paths, on the other hand, are easier to write because they are shorter but are more error prone.



Considering this example of filesystem, for a shell with its working directory being `/home/nicolas/`, there are two possible ways to reference the tutorial file:

- Absolute path: `/home/nicolas/work/tutorial.pdf`
- Relative path: `./work/tutorial.pdf`

3.2 Filesystem navigation commands

Let us review the commands that allow you to move around your filesystem.

3.2.1 ls ("list")

This is an easy one, we have just learned about it in the last section. Try to:

1. List the content of the current directory, including the hidden files[3]. What do you think the `.` and `..` directories refer to? Remember to use the manual to learn about the things you do not know
2. List the content of `/etc`. What do you think is inside?

[3]In UNIX, hidden files' filenames begin with a "."

3. List the content of `/etc` and all of its subdirectories in one command

3.2.2 pwd ("print working directory")

This command is super simple. It simply gives you the full path of your shell's current directory.

Try it! Open a new terminal and type `pwd`.

```
$pwd
/home/nicolas
```

This is what is called your *home*. Every user of the computer has its own home directory, where he can put his personal files.

3.2.3 cd ("change directory")

This command allows you to move from one directory to another. It is similar to double-clicking on the icon of a directory.

1. Try to open the man for this command. What happens?
   `cd` is a builtin command (cf. 2.4). `man` works only with actual programs installed on your system. To get help for this kind of commands, you can use `help my_command`.
2. What is the difference between writing `cd /bin` and `cd bin`?
3. Move anywhere in your filesystem. Then, type `cd` without any argument. Where are you now?
4. Move anywhere in your filesystem. Then, type `cd ~`. Where are you now?
5. Move to `/etc`, then move to `/usr`. Finally, type `cd -`. Where are you now?

3.3 File manipulation

Great, we know how to move in our filesystem. Let us start manipulating files and directories!

3.3.1 mkdir ("make directory")

As its name indicates it, this command allows one to create a new directory.

1. Move to your home directory and create a directory named `code`
2. Move to your home directory and, with a single `mkdir` command, create the two directories `code/python` and `code/python/exo1`

3.3.2 rmdir ("remove directory")

As we just saw how to *make* a directory, this allows one to *remove* one. Note that this command only works on empty directories.

3.3.3 cp ("copy")

This one allows one to copy files and directories from a directory to another.

1. Create a new directory and copy the file `/etc/hosts` inside
2. Create a new directory and recursively copy all of the files and directories from `/etc` inside. Can you guess why warnings are printed to the screen?

Also, remember this in the previous section?

> *Imagine how tiresome it would be to copy only .jpeg files of size <100Kb from one huge directory to another using Window's file manager?*

With `cp`, combined with a wildcard[4], we can super easily copy all files which end with `.conf`. It would not be too complicated to only select files of size <100Kb, but let's keep this for another time.

This example demonstrates how to copy all files which end with `.conf` from `/etc/` to `/tmp/mdr`.

```
$mkdir /tmp/mdr # Create a new directory called mdr, in /tmp/
$cp /etc/*.conf /tmp/mdr/ # Copy all files that end with.conf from /
   etc/ to /tmp/mdr/
$ls --format=single-column /tmp/mdr/ # Verify that the files have
   been correctly copied
adduser.conf
apg.conf
appstream.conf
brltty.conf
ca-certificates.conf
debconf.conf
deluser.conf
fuse.conf
fwupd.conf
gai.conf
hdparm.conf
host.conf
idmapd.conf
insserv.conf
kernel-img.conf
kerneloops.conf
ld.so.conf
lftp.conf
libao.conf
```

[4] https://www.shell-tips.com/2006/11/04/using-bash-wildcards/

```
libaudit.conf
logrotate.conf
ltrace.conf
mke2fs.conf
mtools.conf
nsswitch.conf
pam.conf
pnm2ppa.conf
popularity-contest.conf
request-key.conf
resolv.conf
rsyslog.conf
sensors3.conf
signond.conf
smi.conf
sysctl.conf
ucf.conf
updatedb.conf
usb_modeswitch.conf
```

3.3.4 mv ("move")

The `mv` command allows one to move a file or directory somewhere else. It is similar to a "Cut&Paste". It can also be used to rename a file.

1. Create a new directory, copy the file `/etc/hosts` inside and rename it `file1`.
2. Duplicate `file1` and create another directory. With one command, move both files to that new directory

3.3.5 rm ("remove")

This command allows one to remove a file from the filesystem.

Be careful when using it as you will not be prompted a warning when you are attempting to delete a file. What is more, there is no "Trash bin" in the command line world. Once a file is deleted, it is gone for good.

1. Remove every file and directory you put in your home directory in the previous exercises

3.3.6 touch

Last but not least, the command `touch` allows one to change the time of last modification of a file. Honestly, it is not very useful, but it is possible to use it to create empty files:

```
$mkdir test
$cd test
$touch hello
$ls
hello
$
```

3.4 Peek inside files

We have manipulated directories and files but we still do not know how to show their content. Let us dig into this now.

3.4.1 cat ("catenate", it seems)

This command simply displays the content of the files given as argument.

1. If you run a Linux distribution, read the content of `/etc/os-release`
2. If you run another system, read the content of `/etc/shells` instead
3. What do you think the file `/etc/os-release` refers to?
4. Read the content of `/etc/hosts` and `/etc/services` in just one command
5. Using an option of cat, find out how many lines are inside the file `/etc/services`[5]

3.4.2 head & tail

These two commands are similar to `cat` but only display the beginning (head) or the end (tail) of a file. It can be useful to read the beginning or the end huge files. The number of lines displayed default to 10, but it can be changed with an option.

1. Display the first 20 lines of `/etc/protocols`
2. Display the last two bytes of `/etc/hosts`. Display the last byte of `/etc/hosts`. Why do you think you get this result? Using the `-E` option of cat, identify what the last byte of this file is

[5]There is a command called `wc` which is specialized in counting the number of lines, but let us not use this one for now :)

3.4.3 less

This program is a pager. It allows one to look into a file without having to load all of its content into memory (hence the name, it *paginates*). It is extremely useful when dealing with huge files, like servers logs. This pager is invoked using `less path_to_file` and provides the user a way to:

1. Navigate in between the lines: press *up* and *down* arrow keys or *j* and *k*)
2. Move to top or bottom: press *g* or *G*
3. Search for the occurrences of a work: press */*, write your word, press *Enter* and navigate through the occurrences using *n* and *N*
4. And many more...

Use *h* to get some help inside the pager and press *q* to leave.

The design of `less` might make you think of `man`. That is because `man` is also a pager. The shortcuts we just went through can also be used when reading a man page! Useful when trying to look for specific information :)

4 File editor, package manager and permissions

Great, we have seen many things. But we are still not able to edit files. Let us remediate to this

4.1 File editor

There are many file editors[6] available in the terminal world. Pick one in the following list.

vim: This is a highly configurable text editor. It is lightweight and designed to optimize writing code. There is an incredible set of shortcut and commands to your disposal, which allow you to do tons of advanced things.
Also, it is an in-terminal software, meaning that you will stay in the same terminal window as the one you currently are in when you invoke it.
Still, beware, because it is not an easy task to get started with using **vim**. The controls are not-intuitive[7], so it is a pain to get started with it.

nano: Just like **vim**, **nano** is an in-terminal software. It is simpler to use but provides less features.

gedit: It is a primitive GUI editor bundled with GNOME[8]. It aims at simplicity and is perfect for performing simple editing tasks.

Atom: A GUI editor. It provides a beautiful and clean interface. One can also install plugins and themes to customize it. It is described as a *hackable text editor*: combine plugins and your configuration to make it your own.

[6]We will not discuss IDEs in this tutorial
[7]Many are similar to the pager's
[8]`https://en.wikipedia.org/wiki/GNOME_Shell`

Only downside: it is a bit slow at times.
sublimeText: Very similar to Atom, but in general faster.

4.2 Choose your fighter!

If you do not feel at ease and want to go for the easiest thing, go for **gedit**.

If you feel adventurous and wish to take the difficult path, go for **vim**. Knowing how to do basic stuff with this tool is always useful as pretty much every UNIX machine in the world has at least **vim** or **vi** (simpler vim)[9]. You can get away with using vim thanks to the following tips:

- Do not use the mouse, it will not work well
- Upon opening a file, hit `i` to begin writing text: you will enter insert mode
- When you are done and want to quit, hit `ESC`, then write `:wq` to save&quit or `:q!` to quit without saving
- Pay attention to the mode you are currently in: if it is written `INSERT` at the bottom left of the screen, that means that you are in insert mode and you can type text. If it is written something else, hit `ESC` to go back the default mode

If you do not feel comfortable learning how to use an in-terminal text editor like vim, but still want to use something sophisticated, my advice for you is to pick either **Atom** or **sublimeText**.

4.3 You chose gedit or nano!

Chances are that **nano** and **gedit** are already installed in your system. To check, simply use the command `which`. This command allows one to get the location of the program.

```
$which nano
/bin/nano
$which gedit
/usr/bin/gedit
```

You are ready to edit files! You can skip the next sections dealing with text editors. Still, read the content of those as they contain information that you need to know.

4.4 You chose vim!

Chances are that **vim** is not installed in your system. To check, simply use the command `which`. This command allows one to get the location of the program.

[9]Useful when you have are controling a server remotely and only have access to a terminal!

```
$which vim
$
```

As you can see, `which` prints nothing, so that means that the command `vim` is not found by your shell. It most likely means that the program is not installed. This is the perfect opportunity to learn how to install programs in UNIX-like systems!

4.4.1 Package manager

A package manager is a tool allowing one to install, uninstall and update *packages*. A package is simply an archive containing source code and/or applications.

Ubuntu's default packet manager is `apt`. Mac OS does not have one, but you can install `brew`[10].

A look at the man indicates us that we can install a package using `apt install package_name` (`brew install package_name` for Mac users).

```
$apt install vim
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
   Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-
   frontend), are you root?
$
```

Oops, that did not work. It seems that we do not have sufficient permissions! We can bypass this using the `sudo` command. It allows one to launch commands as administrator (*root*). Keep this in mind, we will talk about permissions in the next section.

```
# Note: while typing your password, nothing gets printed to the
   screen. Don't panic, that's normal, simply type your password and
   hit enter :)
$sudo apt install vim
[sudo] password for nico: Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  vim-runtime
Suggested packages:
  ctags vim-doc vim-scripts
```

[10]`https://brew.sh/`

```
The following NEW packages will be installed:
  vim vim-runtime
0 upgraded, 2 newly installed, 0 to remove and 297 not upgraded.
Need to get 6 589 kB of archives.
After this operation, 32,0 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Here, we enter `Y` and press enter to validate. Informational text is displayed and we get control back. Let us check that we installed `vim` correctly:

```
$which vim
/usr/bin/vim
$
```

Great, it worked!

Refer to the manual for more information on `apt` (how to remove packages, how to upgrade software and how to update the list of available packages).

4.4.2 Installing packages manually

In the previous section, we studied how to install programs with `apt` or `brew`.

Keep in mind that not every program is available in the official package repositories. This is the case for **Atom** and **sublimeText**. We will detail in this section how to manually install a software.

So, let us say we want to install **Atom** on a fresh Ubuntu system.

```
$sudo apt install atom
[sudo] password for nicolas:
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package atom
$
```

Here, the error message is pretty explicit: `apt` could not find a package containing the program `atom`. We will have to install it ourselves.

Going to their official site, we can download a `.deb` file: `https://atom.io/download/deb`. Once saved to our system, we will simply call the following `dpkg`[11] command to install it.

```
$ls Downloads
atom-amd64.deb
$sudo dpkg -i Downloads/atom-amd64.deb
[sudo] password for nicolas:
Selecting previously unselected package atom.
(Reading database ... 364547 files and directories currently
   installed.)
Preparing to unpack Downloads/atom-amd64.deb ...
Unpacking atom (1.33.1) ...
Setting up atom (1.33.1) ...
Processing triggers for desktop-file-utils (0.22-1ubuntu5.2) ...
Processing triggers for bamfdaemon (0.5.3~bzr0+16.04.20180209-0
   ubuntu1) ...
Rebuilding /usr/share/applications/bamf-2.index...
Processing triggers for gnome-menus (3.13.3-6ubuntu3.1) ...
Processing triggers for mime-support (3.59ubuntu1) ...
$which atom
/usr/bin/atom
```

There you have it!

4.4.3 Finally, editing a file!

Now that your favorite editor is installed, simply invoke it with the name of the file you want to create:

```
$cd Documents
# This works with gedit and all the other tools. Simply replace "vim"
    by the name of your text editor
$vim my_document # or gedit my_document or nano my_document or atom
   my_document...
# Now, either a pop-up appears or your terminal is replaced by the
   text editor. In most cases you CANNOT use your terminal until the
   text editor is closed
(...editing...)
# Once you save your modifications and close the text editor, you can
    re-use your terminal
```

[11] Actually, `apt` uses `dpkg` underneath the calls to its high-level API

```
$cat my_document
Hello!
$
```

4.5 Permissions and root user

Remember in last section when we encountered a *Permission denied* error? Let us explain what happened here.

4.5.1 File permissions

```
$apt install vim
apt install vim
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
   Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-
   frontend), are you root?
$
```

It seems that, for some reason, `apt` needed to open the file `/var/lib/dpkg/lock-frontend`.

In UNIX-like systems, each file is attributed a set of permissions:

– read (**r**, value **4**): this permission allows one to get the content of a file or to list the content of a directory
– write (**w**, value **2**): this allows one to edit the content of a file[12]
– execute (**x**, value **1**): this last permission allows one to execute a file if it is a program or a script. If it is a directory, this means that the user can move to it

There is thus a total of 8 different possible combinations:

4.5.2 File ownership

In UNIX-like systems, we say that a file is owned by a user. If the user `nicolas` creates a file, he will be the owner.

Linking that distinction to the mechanism of file permissions, a file in your filesystem is supposed to have three triplets of permissions: one for the user which owns it, one for the group of the user which owns it and lastly one for every other user of the system.

[12]Not renaming or destruction of a file. Permissions for this kind of operation are a bit more complex: `https://unix.stackexchange.com/a/230508`

	Textual representation	Numerical representation
No permission	**—**	**0**
Execute-only	**–x**	**1**
Write-only	**-w-**	**2**
Write & execute	**-wx**	**3** (2 + 1)
Read-only	**r–**	**4**
Read & execute	**r-x**	**5** (4 + 1)
Read & write	**rw-**	**6** (4 + 2)
Read & write & execute	**rwx**	**7** (4 + 2 + 1)

Table 1: The 8 file permissions combinations

This "triple triplet" is in the following order: *user, group, others.*

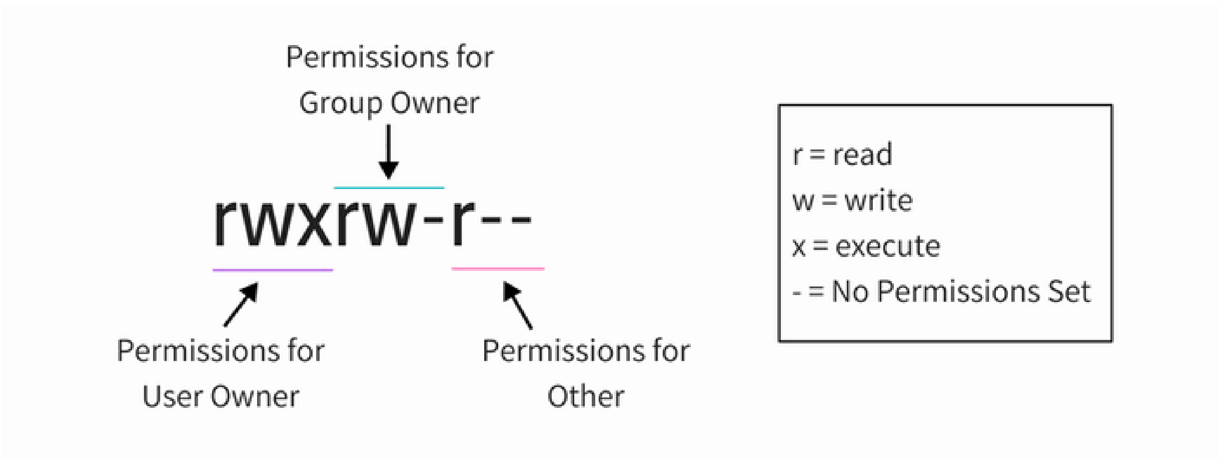


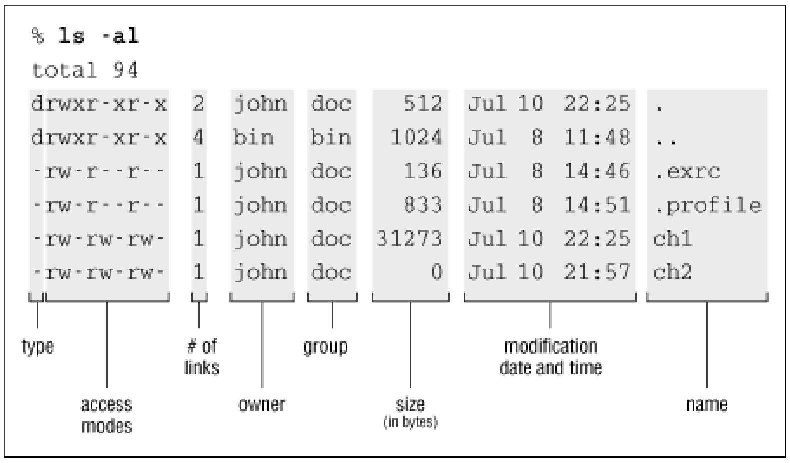
Figure 2: Permissions of a file

4.5.3 Showing a file's permissions

The `-l` option of `ls` allows one to show files in the *long-listing format*, including the owner of the file and the file permissions.

Let us try:

```
$touch my_file
$ls -l my_file
-rw-rw-r-- 1 nicolas nicolas 0 janv.  7 01:48 my_file
```



```
% ls -al
total 94
drwxr-xr-x  2  john  doc    512  Jul 10  22:25  .
drwxr-xr-x  4  bin   bin   1024  Jul  8  11:48  ..
-rw-r--r--  1  john  doc    136  Jul  8  14:46  .exrc
-rw-r--r--  1  john  doc    833  Jul  8  14:51  .profile
-rw-rw-rw-  1  john  doc  31273  Jul 10  22:25  ch1
-rw-rw-rw-  1  john  doc      0  Jul 10  21:57  ch2
```

Figure 3: Long listing format

Here, the owner is `nicolas` and so is the group of the owner. The access rights say that I can read or modify the file and so can the members of the group `nicolas`. However, users not in the group `nicolas` only have read access.

Let us see another example:

```
$which mkdir
/bin/mkdir
$ls -l /bin/mkdir
-rwxr-xr-x 1 root root 76848 mars   2  2017 /bin/mkdir*
```

The file `/bin/mkdir`, which I invoke using the command `mkdir`, is owned by the `root` user (i.e the administrator). The `root` user can read it, modify it and execute it. The members of its group can only read it and execute it. I can also read it and execute it. Let us verify:

```
# Read permission
$head -c 4 /bin/mkdir
ELF$

# Write permission
$ rm /bin/mkdir
rm: remove write-protected regular file '/bin/mkdir'? y
rm: cannot remove '/bin/mkdir': Permission denied

# Execute permission
```

```
$/bin/mkdir /tmp/my_new_directory
$ls -d /tmp/my_new_directory
/tmp/mydir/
$
```

Knowing this and the fact that `sudo my_command` launches `my_command` as the `root` user, can you troubleshoot why we could not open the file `/var/lib/dpkg/lock` as a simple user?

```
ls -lh /var/lib/dpkg/lock
-rw-r----- 1 root root 0 janv.  6 22:32 /var/lib/dpkg/lock
```

4.5.4 Modifying permissions

The file owner and the root user can both modify the permissions of a file. One calls the `chmod` command for that.

There are different ways to invoke `chmod`. The following lines are examples that will help you understand how one changes the three triplets of permissions for a file.

Octal representation: The first digit represents the right of the owner, the second digit the rights of its group and the last one the rights associated to all the other users

```
# Set read-only rights for everybody
$chmod 444 my\_file
# Set read-write-execute rights for the user, read-write for the
   users of its group and read-only for other users
$chmod 764 my\_file
```

Symbolic representation: With symbolic representation, you do not set permissions, you add (+) or remove (-) them to the **u**ser, **g**roup and/or **o**thers.

```
# Add write permission for the other users
$chmod o+w my_file
# Remove execute permission for owning user and its group
$chmod ug-x my_file
# Add read-write permissions to user and its group and read for
   others
$chmod ug+rw o+r my_file
# If invoked without u, g or o, it defaults to ugo: this adds the
    execute permission on the file for all users
```

```
$chmod +x my_file
# This is similar to the following (a stands for all)
$chmod a+x my_file
```

It is also possible to modify the file owner using the command `chown`. We will not go throught this one though.

4.6 Creating your own programs

Did you think creating your own scripts was a difficult task? Actually, you have pretty much all the cards in your hands right now to create your very own programs. You know how to create a file, edit it and set it executable.

4.6.1 A shell script

Let us create our very own shell script now! Using your favorite text editor, create a file called `my_ls.sh`.

Although this is not absolutely needed, it is a good practice to start a script with a *shebang*. A *shebang* is a line indicating which program is supposed to run the script. It is made of a sharp (`#`), a bang (`!`) and the full path of the program which will run the script.

A shell script can be run using `bash`, the shell that we have been using since the beginning of this tutorial, or any other shell (`sh`, `zsh`, etc).

First of all, locate bash's full path. Open a terminal and type the following:

```
$which bash
/bin/bash
```

Then, write the shebang at the beginning of your script:

```
#! /bin/bash
```

Then, you can start building your script. A shell script is simply a list of commands. Shell being a script language, there exists conditional statement, loops and many much more. For now, we have not studied the subject enough, so let us simply call a simple command with fancy options: a recursive `ls` with long-listing format, also displaying hidden files.

```
#! /bin/bash

```

```
ls -Rahl
```

That's it. We have our first program, a `ls` on steroids! Now, the syntax to launch a program is as follows[13]:

```
$./my_ls.sh
bash: ./my_ls.sh: Permission denied
```

Oops, something is wrong. We forgot to make the file executable. Easy, let us use `chmod`.

```
$chmod +x my_ls.sh
$./my_ls.sh
(huge output)
```

It works! :)

Now, this is a very simple example, but remember that Shell is a full-fledged language: there are conditions, loops, arrays and all. You can build real applications with it!

4.6.2 A python script

You have seen how to write a shell script, but it was not that useful nor impressive, since you may not have sufficient knowledge in this scripting language.

Python being an interpreted language, one can write Python scripts just as Shell scripts. There is below the code for a number guessing game. With basic notions of Python, you should be able to get what it is doing easily. We will make a script out of this snippet, and run it inside the terminal.

First, install Python3:

```
# For Ubuntu users
$sudo apt install python3-pip
```

```
# For Mac OS users
$brew install python3
```

Now, find the absolute path to Python3's binary:

[13] It also is possible to directly call the executable: `/bin/bash my_ls.sh` or `/bin/python3 my_python_program.py`

```
$which python3
/usr/bin/python3
```

Finally, create a file named `guessing_game.py` and paste the following code snippet inside (**click here**). Do not forget to change the shebang if it does not fit the location of your python3 binary.

```python
#! /usr/bin/python3

# The random package is needed to choose a random number
import random

# Define the game in a function
def guess_loop():
    # This is the number the user will have to guess, chosen randomly
        in between 1 and 100
    number_to_guess = random.randint(1, 100)
    print("I have in mind a number in between 1 and 100, can you find
        it?")

    # Replay the question until the user finds the correct number
    while True:
        try:
            # Read the number the user inputs
            guess = int(input())

            # Compare it to the number to guess
            if guess > number_to_guess:
                print("The number to guess is lower")
            elif guess < number_to_guess:
                print("The number to guess is higher")
            else:
                # The user found the number to guess, let's exit
                print("You just found the number, it was indeed",
                   guess)
                return
        # A ValueError is raised by the int() function if the user
           inputs something else than a number
        except ValueError as err:
            print("Invalid input, please enter an integer")
```

```
# Launch the game
guess_loop()
```

Save the file, make it executable using `chmod` and launch it:

```
$./guessing_game.py
I have in mind a number in between 1 and 100, can you find it?
50
The number to guess is higher
75
The number to guess is lower
63
The number to guess is lower
57
The number to guess is lower
53
The number to guess is higher
55
The number to guess is higher
56
You just found the number, it was indeed 56
$
```

There it is! You actually *ran Python code*. It might look like nothing, but believe it or not, what servers do all around the world is nothing but run code like this snippet[14].

You can now write your own Python applications and run them in your *own* environment. You do not have to rely on online platforms anymore.

Regarding Bash and Linux, there are still many things to learn, but this is a good start. Chances are that you will not assimilate everything from this tutorial, it is pretty dense after all. But do not hesitate to come back to this resource every once in a while to refresh your mind :)

[14]Maybe stuff more useful than a guessing game, but still!