Shell summary

What is Linux? What is GNU? - short version

Linux is a kernel for an operating system, i.e., the software which comunicates with the hardware. GNU is a collection of computer software, which forms an operating system. Together, they form the most common free operating systems used today. Usually written GNU/Linux, however, many speak only Linux, as the whole.

Why Linux - Free as in freedom

Linux is free - as in freedom, not as in gratis. You can copy the Linux code and create your own version of Linux, as long as you make that modification free as well. The details of this are not so simple, as are usually studied on licenses.

Kernel + whatever else you want

A main difference between Linux and the other most common operating systems - Mac's OSX and Microsoft's Windows - is that Linux is delivered in pieces. To have a working Linux system, you only need the kernel and a few GNU programs. What comes on top of that is up to your personal choices. A few of those choices are already pre-built in what is called a distribution (Ubuntu, Fedora, Debian, Arch Linux, etc.)

Shell -> Communication

The shell is the basic communication tool for Linux. Older people may remember DOS, which was similar. It is the simplest way to give your computer orders, as is very useful in many contexts.

- Graphical Interface: What's a click gonna do? And a double click? And a middle click and drag? Different GUIs can have different meanings for the same icons.
- Command Line: The help is very specific. It's usually faster. Less resources are required. Automation of almost anything is possible.

File system

To understand file systems, let's learn our first command.

Basic command 1: `ls`
Lists the contents of a folder

- `ls`: list the current folder
- `ls "some/path"`: list the contents of "some/path"

The root `/`

Unlike Windows, **everything** is under root. While in Windows there are drivers (C:, D:, E:, etc.), in Linux, the partitions of the hard drive are slightly hidden. There can't be two roots.

```
ls /
```

The `/bin`

`bin`, for binary, indicates where the *executables* are supposed to be. `/bin` is the main place for executables, but any folder named bin is (expected to be) used for binaries, and this `/bin` is for the system binaries.

```
ls /bin
```

The command `ls` is in this bin.

`/lib`

Location of the libraries, i.e., information that programs need to run. lib is like bin, there are many folders named lib.

`/usr`

This folder has many of the folders designed for users - plural, not a specific user. For instance,

- /usr/bin: This folder has more specific binaries.
- /usr/lib: This folder has more specific libraries.

`/home`

Folder for the users' home folders. When you open the terminal, your home folder is the current folder.

Basic command 2: `pwd`

`pwd` returns the current folder.

```
pwd
```

Basic command 3: `cd`

`cd` changes the current directory. In conjuction with `pwd` and `ls`, it's essentially all you need to navigate the Linux file system.

- `ls`: To see folders at the current folder
- `cd folder`: Changes the current directory to a folder in the current folder
- `cd /full/path`: Changes the current directory to a folder in a given full path
- `cd ..`: Returns to the previous folder in hierarchy.
- `cd`: Returns to your home folder.

Exercises

1. Navigate do the `\`, `\bin`, and `\usr` using the full path.
   Use `pwd` and `ls` in each location.
2. Navigate to your home folder by issuing the full path.
3. Go backwards to the `\home` folder, without using the full path.

Special folders

There are two special folders which are actually links to special places:

- `..`: Refers to the parent directory of the current directory, i.e., one folder above.
- `.`: Refers to the current directory. Usually unnecessary.

Another important point is the difference between an absolute and a relative path.

- An *absolute path* starts from the root: `/home/abel/images/my-cats.png`.
- A relative path (relative to the current folder) starts theoretically with ., but in pratice with nothing. From the `/home/abel` folder: `images/my-cats.png` or `./images/my-cats.png`.

Command options

Most of the shell commands accept additional arguments to indicate specific options. For instance:

```
ls -l
```

Lists with additional information

```
ls -d
```

Lists only directories

```
ls -a
```

Lists hidden files too

Obs: Hidden files on linux are prepended with a dot

```
ls -la
ls -l -a
```

Lists all files, including hidden, with additional information.

Basic command 4: `man`

This command gives the manual of a command.

```
man ls
```

Navigate the manual with

- up and down
- / to search
- q to quit

Exercises

1. What does ls -t does?
2. Find how to list files in reverse order.

Editing files on the terminal

On Linux, there are many tools to edit files in the terminal, although there isn't a reason to use a tool specifically designed for the terminal. One of them is nano.

```
nano
```

At nano's footnote there are some commands. They use the notation `^`, which usually means `Ctrl+`. For instance, `^X = Ctrl+x` quits nano and `^O` saves it.
If you write something, before quitting you'll be asked if you want to save your file.

Do now: Write some lines and save it.

Basic command 5: `cat`

cat shows the contents of a file.

Basic command 6: `cp`

cp copies a file to another place.

Basic command 7: `rm`

Remove a file, or files, or possibly folders.

Basic command 8: `mkdir`

Creates a new folder.

Basic command 9: `rmdir`

Removes an empty folder.

Basic command 10: `mv`

mv can move a file or folder to another place or be used to rename a file or folder.

Exercises

1. Create a file named food.txt including a list of foods, one per line.
   - Copy `food.txt` to a file named `random.txt`.
   - Remove file `food.txt`.
   - Create a folder name `personal`.
   - Move `random.txt` to folder `personal`.
   - Rename folder `personal` to `foods`.
   - Rename `random.txt` to `food.txt` leaving it inside folder `foods`.

Playing with the downloaded data set

Do it now: Download the data set.
Do it now: Navigate to the `data_process_example` folder.

In this folder there is program `process_data`, which we'll use to process the data on the folder `data`.
First, let's see the first lines of a data file.

Basic command 11: `head`

Head prints the first lines of a file

```
head data/data001.dat
```

Looking at the data, we see that it's a set of (x,y) points. These were all simulated, and don't matter here.

The program `process_data` doesn't have a graphical interface, so we have to run it from the terminal. Unlike the other commands so far, simply running `process_data` will not work. This is related to an environment variable named `PATH`. Running a command on the shell, involves the process of searching this command on the folders on PATH. Usually `/bin`, `/usr/bin`, `/usr/local/bin`, etc.
Hopefully, we can circunvent that by indicating the path of the command. Since the command is in the current folder, the path is

```
./process_data
```

If everything worked out, then an error message appeared. Try running

```
./process_data data/data001.dat
```

A summary is printed out.

Suppose you want to store this summary. The usual way to do this is to redirect the output to the first program to a file.

Intermediate command 1: `>`

The command `>` redirects the output of a program to a file.

```
mkdir output
./process_data data/data001.dat > output/data001.out
cat output/data001.out
```

Basic command 12: `wc`

This command counts the number of bytes, words or lines of a file.

```
wc -l output/data001.out
```

counts the number of lines on file `output/data001.out`.

Intermediate command 2: `|` (pipe)

The command `|` redirects the output of a program to another program. For instance, to count the number of programs in the folder `/bin`, we can list the contents of `/bin`, and then `wc -l` the result.

```
ls /bin | wc -l
```

Exercises:

1. Count the number of files on the folder `data`.
2. Run the program `process_data` on data 002 to 005 storing the result of each run to a correspondent .out file.

Working with loops

The shell, like many programming languages, support loops. One syntax for it is

```
for VAR in LIST
do
    CMDs
done
```

Here, `VAR` is a variable, which can be accessed with `$VAR`. For instance, to run the program with files 1 and 2, we can issue

```
for file in data/data001.dat data/data002.dat
do
    ./process_data $file
done
```

Notice that we are not saving the outputs anymore. For that we need to extract the pure file name from `data/data001.dat`. That can be done with

```
basename data/data001.dat .dat
```

To run this result and store it in a variable, we use `var=$(cmd)`.

Here is the result:

```
for file in data/data001.dat data/data002.dat
do
    fname=$(basename $file .dat)
    ./process_data $file > output/$fname.out
done
```

Now, our real objective is to run this for all file in data. We can use *wildcards* for this job. `data*.dat` will specifically for for the types of file we are working. I'll use `echo` to show some different wildcards, though.

```
for f in data/data*; do echo $f; done
for f in data/data*1.dat; do echo $f; done
for f in data/data*1*.dat; do echo $f; done
for f in data/data?1?.dat; do echo $f; done
for f in data/data?{1,2}?.dat; do echo $f; done
for f in data/data?{1,2,3}?.dat; do echo $f; done
for f in data/data?{1,2,3}{1,2}.dat; do echo $f; done
```

- `*` means any;
- `?` means any single digit;
- `{a,b}` means any in the set.

Probably the lesson will be over by now

Otherwise, I'll try to cover the advanced commands `grep` and `find`, or creating scripts.