A Brief Introduction to the Linux Shell for Data Science

Aris Anagnostopoulos

1 Introduction

Here we will see a brief introduction of the Linux command line or *shell* as it is called. Linux is a Unix-like operating system, which is open source. There exist several distributions Linux that one can install for free, among which Ubuntu is probably the most popular for single users. Although Linux has a graphical environment, programmers use very often the shell to perform some tasks, thus it is useful to have some experience with it. In these notes we will see some of the basics to be able to work with the shell evenif you have never worked on it, and we will then see some commands for performing some simple mining tasks.

If you want to install Linux in your computer, you can do it by creating a *virtual machine* inside your computer, for example by using VirtualBox.

Although Python and R were initially developped on the Unix environment, now there exist versions for all popular operating systems as well, so you most probably will not need to install Linux.

2 Some Commands to Warm Up

When you open the shell you see the *command prompt* or *command line*, which is waiting for your command. For example, typing

```
pwd
```

shows you the current directory, for example, somehting like

```
/home/studente16
```

Similar to other operating systems, the *filesystem* of Linux is organized into *directories* or *folders*, in a tree-like structure. The *root directory* is the `/`, which has several subdirectories, one of which (in the above example) is the directory `home`, which n turn contains the directory `stedente16`.

To see the list of files in the current directory type

```
ls
```

Without any parameters, the command `ls` shows the contents of the current directory. To see the contents of another directory, use

`ls` *`directory`*

For example

```
ls /
```

shows the contents of the root directory and

```
ls /home
```

shows the contents of directory `/home`.

There are two ways to specify a directory. One is to specify an *absolute path* way, as we did here. An absolute path specifies the complete directory path starting from the root, so it starts with `/`. The second way is to specify a *relative path*. A relative path is with respect to the current directory. For example, if the

current directory is `/home/studente16`, then typing

```
ls Music
```

shows the list of files in directory

```
/home/studente16/Music
```

and

```
ls Music/Rock
```

shows the list of files in directory

```
/home/studente16/Music/Rock
```

This example shows the typical structure of a command. It starts with the program that we run (here `ls`), and it may have zero or more arguments, separated by one or more space characters. Some of the arguments are options that we pass to the program, they usually start with a `-`, and change the behavior of the program; we will see them later.

Let's return to our discussion on directories. Each directory, has two special directories: `.` is the current directory. `..` is the directory one level up. For example, typing

```
ls .
```

shows the contents of the current directory, similarly to just typing `ls`. (`.` is useful for other commands.) To type the contents of the directory one level up (i.e., one level closer to the root), type

```
ls ..
```

This will show the contents of `/home`, and

```
ls ../matlab
```

shows the contents of `/home/matlab`.

To change the current directory one can use the command `cd`. The syntax is

```
cd directory
```

where again *`directory`* can be absolute or relative. For example, typing

```
ls ..
```

has the same result as typing

```
ls /home
```

and makes `/home` the current directory, something that we can verify with `pwd`, as we saw in the beginning.

To get help for a command you can use the command `man` (for manual). For example,

```
man ls
```

will show the manual page for command `ls`. You can use the space key to move to the next page, and `q` to exit.

```
man pwd
```

will show the manual page for `pwd`, whereas

```
man cd
```

will produce an error. The reasong is that `cd` is what we call an *internal command*, and its manual page can be found somewhere else[1]

[1]It can be found if reading the manual page of the bash shell, by typing `man bash`. The discussion about what is a *shell* is

Let us go back to the command `ls`. Except for parameters, we can also *pass* to a command some options, which change the behavior of the program. Options are usually specified by putting a -. Try to type

```
ls -l
```

The option *-l* says to the command `ls` that we want a long listing. Let us now explain the information from this listing. Each line corresponds to a file that exists in the directory and looks like this:

```
drwxrw-r- 1 studente16 studente16 4096 Oct 2 06:07 Music/
```

Let us explain the information here. The first character gives information about the type of the file. `d` means that the file is a directory, `-` means that the file is a regular file. The next 9 characters, show the permissions of the files and we will talk about them in Section 3. You can ignore the number after that (here `1`). Then next two fields are the owner of the file and his group; we will talk more about it in Section 3, as well. Next we have the size of the file in bytes, then the date and time that it was last modified, and finally the name of the file (`Music`). Because the file is a directory file it also the `/` at the end, something also indicated in the beginning of the line with the `d`, as we said.

3 Users, Groups, and Permissions

Linux has a system of users and permissions. When a user logs in with her password she has some permissions, that is, she can edit some files or some directories but not others. Furthermore a user can belong to one or more groups, but we will not deal with this here. Each file belongs to a user. When executing the command `ls -l` and we obtain the list

```
drwxr-x-- 1 studente16 studente16 4096 Oct 2 06:07 Music/
```

then the third field (`studente16`) is the name of the user who owns the directory `Music`. (In our case there exists also a *group* with the name `studente16` to which the directory `Music` belongs and this is indicated by the fourth field.)

Each file has some permissions, which are given by the first field. As we said, the first character determines the type of the file. The other nine characters define the permissions on the file. They are grouped into three groups of three characters: the first group defines the permissions of the owner of the file, the second group defines the permissions of the users that belong to the group of the file (the one given by the fourth field), and the third, the permissions for everyong else. In each group there are three characters, which indicate whether the corresponding set of users has a permission or not. They are `r`, `w`, `x`. When we see the character then this means that the corresponding group of users can **r**ead/**w**rite/e**x**ecute the file. For instance, in the above example, you (the user `studente16`) can read the file, write (i.e., modify it), and execute it. Execute, for a regular file, means to run it as a program. For a directory it means that one can enter there and access files there; usually for directories either both `r` and `x` are active or none. Users who also belong to the group `studente16` have the rights to read the directory `Music` but not modify it; for example they can enter there but there cannot create a new file there. Finally, all the other users cannot access at all the directory `Music`.

An owner of a file can change its permissions using the command `chmod`. You can use the man page for more information.

Finally, there exists a *super user*, called *root*, who is typically the administrator of the system. The administrator has the right to read everyone's files, change the permissions, and so on. Typically, unless one installs her own version of Linux, she does not have super-user privileges.

4 Simple File Operations

Let us now see some simple ways to operate on files. First we create a new file by typing

```
man ls > help.txt
```

a more advanced topic and we will not talk about it here.

This command creates a new file, which contains the manual page of the command `ls`. It uses what is called *redirection* and says to the system to send the output instead of the screen to a file called `help.txt`. Redirection commands turn out to be very useful, and we will study them more in Section 5.

Type `ls -l` to see the newly created file. To check its contents one can use the the `cat` command:

```
cat help.txt
```

which simply shows the contents of the file. Here it is more useful to use the command

```
more help.txt
```

which can be used for longer files, and displays one window screen at a time.

Some commands allow to read parts of the file.

```
head help.txt
```

shows the top 10 lines of the file. One can change the default value of 10 by passing it as a parameter. For instance,

```
head -15 help.txt
```

will show the top 15 lines. Similarly, to show the bottom 15 lines you can use

```
tail -15 help.txt
```

Often one would like to find a particular piece of text. For this there exists the command `grep`. The format is

```
grep text file
```

For example,

```
grep long help.txt
```

shows all the lines that contain the word `long`. Note that if the text contains some space then we need to enclose it into quotes, otherwise the command will take the second word as the filename. Therefore, to search for `long listing` we should type

```
grep "long listing" help.txt
```

The command `wc` (stands for word count) shows some statistics. If we type

```
wc help.txt
```

we obtain

```
210 939 7689 help.txt
```

which tells us that the file `help.txt` has 210 lines, 939 words, and 7689 characters. `wc` can take several parameters, one of whcih is the `-l`, which stands for "lines." If you type

```
wc -l help.txt
```

the program returns just the first number, the number of lines.

There are of course a variety of editors usually installed. In our system we have the `gedit`. Therefore,

```
gedit help.txt
```

creates a new window with an editor, which has opened `help.txt`.

To copy a file you use

```
cp source destination
```

Try

```
cp help.txt newhelp.txt
```

and by using the `ls` command you can see the new file created. `cp` can be also used to copy a file from one directory to another.

To rename a file you can use the `mv` command, similarly to the `cp`. Thus,

```
mv newhelp.txt myhelp.txt
```

renames the file `newhelp.txt` that we had created to `myhelp.txt`. `mv` can be also used to move files from one directory to another.

To remove a file use the `rm` command:

```
rm myhelp.txt
```

Be careful, because usually there is no way to undo.

5 Redirection

In the previous section we saw that the command

```
man ls > help.txt
```

instead of sending the output to the screen it redirects it to a file.

Every program when it runs has what is called the *standard input*, which it can use to obtain some input, and the *standard output*, which is where it can send the output. By default, the standard input is the keyboard and the standard output is the window in the computer screen. When, later in Python, we use the `print` command to print something to the screen, it is being sent to the standard output.

The user of a program can change this default behavior, as we did previously. We use `>` to say to the program being executed (actually we say this to the operating system) to set the standard output to the file specified. If the file does not exist then a new file will be created, with content the output of the program. If the file already exists, then it will get overwritten. (Be careful! This is a common way to lose by mistake a file that you needed!)

Another way to redirect the output is using `>>`. If the file does not exist, then is has the same behavior as `>`. However, if the file exists, it opens and it appends the output to it. Try

```
echo Linux is cool!
```

which returns

```
Linux is cool!
```

The command `echo` simply prints to the standard output whatever is passed as an arguments. (It is not very useful from the command line and it is mostly used inside programs to print something.) Now try

```
echo Linux is cool!  > linux.txt
```

We now created a file `linux.txt` containing the text `Linux is cool!`, as we can check using `cat`.

```
echo Linux is fun!  > linux.txt
```

We have now overwrited the file and now `linux.txt` contains the phrase `Linux is fun!`. If now we type

```
echo Linux is useful!  >> linux.txt
```

we can check and see that we have added the phrase `Linux is useful!` at the end of `linux.txt`.

We can also redirect the standard input using `<`. Let us see some examples. Many of the commands that we have seen take a file as a parameter (`cat`, `head`, `tail`, `wc`, etc.). If we do not specify the file, then the programs take the input from the standard input. To see this, type simply

```
cat
```

Since the default standard input is the keyboard, the program waits for you to type. Type something and press <Enter>. You see that whatever you typed gets at the output. You can continue typing, and everything gets repeated. Thus, without any arguments, `cat` simply copies the standard input to the standard output. To finish, press together <Ctrl>-D, which says to `cat` that we reached the end of the file. We can now redirect the input and type

```
cat < linux.txt
```

We see as output the content of `linux.txt`. Note that the behavior is the same as if we type

```
cat linux.txt
```

However, what happens is different. In this case, it is the program `cat` that sees that there is an argument, takes it, opens the file with that name, and sends the content to the standard output. Instead, in the case that we use `<`, the program `cat` does not even see any argument! When you type the command, the operating system sees the `< linux.txt`, and runs the program `cat`, after setting the standard input to come from the file instead of the keybord.

It is also possible to combine the redirections. For example, just running

```
cat < linux.txt > unix.txt
```

simply copies the file `linux.txt` to `unix.txt`.

There is also another useful redirection which is called the *pipe* and is specified by `|`. A pipe take command in both sides, and sends the standard output of the command on the left side to the standard input of the command of the right one. Therefore, if you try

```
man ls | grep long
```

you will get as output the lines of the `ls` command that contain the text `long`. (Recall that if you pass two arguments, `grep` searches for the text specified in the first one to the file specified in the second one. Similarly to `cat`, if you give only one argument, then `grep` will search for the text that you gave into the standard input.) You can also chain more than two redirections. For example, typing

```
man ls | grep long | wc -l
```

will display the number of lines in the manual page of `ls` that contain the text `long`.

6 A Few More Hints

The following are also useful when working the Linux shell.

The command `clear` clears the shell window.

Previous commands: You can go to the previous commands and move between them using the up (↑) and down (↓) keys.

Autocomplte: When you start typing a file, pressing <Tab> it autocompletes the filename.

Break: If you want to stop a program running you use <Ctrl>-C. Try to close a program normally, as with this you may loose data if they have not been saved, but sometimes it is needed.

As we said, arguments are passed to a program and they are separated with one or more <space> characters. But what if we want to pass an argument that has a space? We did that before, when we used

```
grep "long listing" help.txt
```

If we put some text into quotes then it is taken as a single argument. Thus, the previous command passes two arguments to `grep`.

Except for the regular characters (letters, numbers, etc.), which are the characters that we can see in the screen, there are also *special characters*, such as the *tab* or the *new line* characters. To show to a program

that we want these characters, we use what is called an *escape sequence*. Escape sequences are words that start with the backslash \. The main ones that we will need are

- `\t`: Tab. We will use the tab character very often to separate different fields in a file.
- `\n`: New line. It is an invisible character that says that a line finishes and a new line must start. Thus, it exists in the end of every line.
- `\r`: Return character. There is a difference with files that are created in Linux and Windows systems. In the former the end-of-line character is, as we said, `\n`. Instead, files created in Windows, the end-of-line is marked by the combination of two characters, `\r\n`. Sometimes when you write programs you must make sure you handle both cases, although often this is taken care by the programming language.
- `\\`: This is the standard \ character.
- `\"`: This is the quotes character; we will see right away why we need it.

In Linux we use two types of quotes, double quotes (") and single quotes (’). The difference has to do with the escape characters. The double quotes recognize the escape characters, whereas the single do not.[2] Thus, if we want to search a file `names.txt` for `Maria<Tab>Greco` we should use

```
grep "Maria\tGreco" names.txt
```

If instead we use the single quotes

```
grep ’Maria\tGreco’ names.txt
```

we search for all lines containing the text `Maria\tGreco`. If we want to search for double quotes, we need to either use single quotes, or to escape the double quotes. Thus, to search file `data mining.txt` for `What is "data"?`

```
grep "What is \"data\"?" "data mining.txt"
```

or

```
grep ’What is "data"?’  "data mining.txt"
```

Notice that the file `data mining.txt` has a space inside, so we put it into quotes to pass it as one argument to `grep`. (We can use, of course, either double or single quotes.) For the text string, in the second case we just type the quotes without needing to escape them, as we did when we use the double quotes.

7 Some Simple Analysis

We will now see how we can use Linux tools to do some simple data-mining tasks. Let us start by downloading a file that we will be using. It can be found here:

`http://aris.me/contents/teaching/data-mining-ds-2015/protected/beers.txt`

One can download it by using a web browser. Instead we can download it with the command `wget`. `wget` allows to download multiple files, follow links and so on, so it can be very useful. Here we will just use it to download our file. Run

```
wget -user username -password pass url
```

where *username* and *pass* are the username and password that we said in class (needed because the file is password protected), and *url* is the address of the file written above. After you wait a bit the file will be downloaded.

[2]Note that this is different from what happens in Python. In Python, both of them are equivalent, and they behave like the double quotes in Linux, although there is a way to make them behave like the Linux:w single quotes.

After we download it we can get some information. Using `head beers.txt` we can see that it contains two fields separated by tab character: a name of the beer and a score. To make sure that the fields are separated by the tab character, and not spaces, we can use the option `-T` of `cat`, which prints `^I` whenever it sees a tab. We don't want to do it to the entire file, so we use a pipe.

```
head beers.txt | cat -T
```

Try also

```
wc -l beers.txt
```

and see that the file has about 3 million beer ratings.

Let us assume that we want to find the 10 beers with the highest number of ratings. How can we do it? We will comnine three commands. `sort` orders (alphabetically or numerically) the lines of the input. `uniq` removes duplicate lines and produces counts if asked. `cut` extracts one or more fields.

Let us start with `sort`. If we run

```
sort beers.txt
```

we obtain a list of the beers sorted alphabetically. (Press `<Ctrl>-C` to stop.) We can combine the output with the command `uniq`. This command takes the input and removes contiguous lines that are the same, leaving only one copy of the line. In addition, if we add the option `-c`, we obtain also the number of times that the line was repeated. By sorting the beers, we have put all the ratings of the same beer next to each other, thus with `uniq` we can see how many repetitions we have.

However there is a small issue: Two ratings that have different scores will lead to different lines, so `uniq` will not be able to distinguish them. Thus we would like to obtain only the beer name and ingore the score. For this we use the `cut` command, with which we can extract the first field:

```
cut -f 1 beers.txt
```

takes the input and extracts the first field (assumes that the fields are separated with the tab character, as in our case) and sends it to the output.

We can now combine all these commands:

```
cut -f 1 beers.txt | sort | uniq -c
```

The output is the list of beers, with the number of times that each appears on the left. We want the top-10 beers, so we can sort once again (numerically) and obtain the top-10. The full command is the following:

```
cut -f 1 beers.txt | sort | uniq -c | sort -nr | head
```

We have passed two options to the second `sort` by specifying `-nr`. The option `-n` says that we want to order the lines numerically (so that 10 appears after 9). The option `-r` (we can specify either `-n -r` or `-nr`) says that we want to do reverse sorting, so that we obtain the order from the highest to the smallest. The final output gives us the values that we wanted.

8 What's Next

We just saw the very basics. There are several other commands that one can use: `curl`, `sed`, `awk`, `paste`, `join`, variants of `grep`, ... You can look at the manual pages for these commands or search online.

We will not cover them, as we will start using Python to do more advanced operations, although knowing them allows us sometimes to perform some operations without spending time writing an entire program.