Last edited by **Isabel F Freitas** 11 months ago

Useful Linux Scripts

`ERROR:  [Errno 98] Address already in use`

1. Listen all events running on a port eg.8000: `lsof -i :8000`
2. Detect the process listening on the port and then: `sudo kill -9 <PID>`

- `sudo apt update`
- `sudo apt-get upgrade`

1. PS Command (process):

https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/

```
450  ps -ef|grep postgres
```

---

2. Netstat Command:

https://www.lifewire.com/netstat-command-2618098

```
454  netstat -nlp |grep postgres
```

---

3. Home Directory:

```
456  cd ~
```

---

4. Useful Links:

https://www.ubuntupit.com/the-50-best-linux-commands-to-run-in-the-terminal/

https://opensource.com/article/17/7/20-sysadmin-commands

https://www.thegeekstuff.com/2010/11/50-linux-commands/

http://www.informit.com/articles/article.aspx?p=2858803

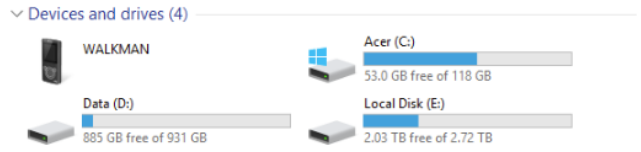
https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/

- **Unix filesystem:** https://en.wikipedia.org/wiki/Unix_filesystem#Conventional_directory_layout

In the first mission, when we defined the command prompt, we mentioned a "current working directory." The **current working directory** (or just **working directory**) is the directory where our terminal session is located. Whenever a shell is running, it is located somewhere in the directory structure. Given that in our case the prompt shows us the current working directory, this means that out current working directory is `/home/dq` and, as we'll see below, this indicates that `dq` is a subdirectory of the `home` folder. Before we learn how to read the string of characters `/home/dq` , we need to learn a bit more about Unix directory structures.

Contrary to what happens in Windows, Unix-like operating systems do not have the concept of "driver letter." In Windows, each driver letter representes a storage device — basically its own file system — and so it is common to have more than one root. Where as in *nix systems there is only one root, called the **root directory**, represented by a single slash: `/` . This is one of the characteristics of Unix-like systems' directory structures. There are other conventions whose details vary by implementation, but these implementations end up being very, very similar.

The diagram displayed in the previous screen is a graphical sample of a Linux directory structure. In fact, it's a sample of the filesystem we're using in this course. Below, we see a screenshot of Windows 10 File Explorer displaying the roots of a computer.



Now we are able to fully understand `/home/dq` . The first slash represents the root directory. `home` is a subdirectory of the root directory — this relationship is translated into characters by `/home` . Similarly, `dq` is a directory contained in `home` , hence `/home/dq` .

Through a different lens, `/home/dq` is also a **path**, a sequence of symbols that specifies the location of a file or directory in the directory structure. You can also think of it as the way to go from `/` to the directory `dq` in the maze that is the diagram we saw on the last screen. The character `/` when not in the beginning of path is called a **directory separator**.

> In our case, our command prompt contains the current working directory. What if it didn't? How would we know where in the directory structure are we located?
>
> There's a command to help us with that: `pwd` .

https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

https://en.wikipedia.org/wiki/File_system_permissions

---

5. History and Re-run command:

```
/home/dq$ history
    1  date
    2  alsoNotRandom
    3  diff -y augustus veruca
    4  diff augustus veruca
    5  clear #-end-3ac2d77569804ff7843d077a1f7f5283-
    6  history
    7  diff augustus veruca
    8  diff -y augustus veruca
    9  clear
   10  history
/home/dq$ !8
```

The output begins by printing what command was ran and then we see the result of running the command. Running `history` again shows us that `diff -qy augustus tv` is saved into the command history, not `!7` .

Here's part of the output of running `history` after running `!7` .

```
2  date
3  alsoNotRandom
4  diff -y augustus veruca
5  diff augustus veruca
6  diff --side-by-side augustus augustus
7  diff -qy augustus tv
8  history
9  diff -qy augustus tv
10  history
```

We can also reference commands counting from the end, just like you can with Python lists. In the example above:

- To to run the last command (`10 history`), we can run `!-1` . Or alternatively `!!` .
- To run the next-to-last command (`9 diff -qy augustus tv`) we can run `!-2` .
- And so on.

Many people find that having a screen filled with irrelevant information is distractive. This happens frequently after using `history` . To clear the screen, we can use the command `clear` .

Useful Commands:

- print working directory: `pwd`
- list: `ls`

- list all even hidden files: `ls -ll -a`
- history: `history`
- run the previous command: `!!`
- run a command from history: `! number_of_the_command`

On the previous screen, you ran the command `pwd`. It's an acronym that stands for *print working directory*. It prints to the terminal window the path of the directory we're located in. By default, the directory we will be in when we open our terminal window will always be `/home/<username>`, where `<username>` is a placeholder for the user account of whoever happens to be logged in the shell.

The above-mentioned directory is known as the **home directory** and it is where people usually save their files. Your home directory is `/home/dq`. Note that **the home directory is not the directory** `home`.

Now that we know where we are and the path out of the maze, it's time to know how we can tell what the contents of the directory we are currently in are. There is a command that lists the contents of any directory that we can access, it is called `ls` (short for list) and it is one of the most commonly used commands. It also accepts options and arguments. Below we see the result of running `ls -p /dev` in the terminal to the right.

```
/home/dq$ ls -p /dev
core  full     null  pts/    shm/    stdin   tty      zero
fd    mqueue/  ptmx  random  stderr  stdout  urandom
```

Let's break down this command:

- First comes the command `ls`.
- It is followed by the option `-p`. This option signals what items are directories by appending a slash to the end of their names, allowing us to tell directories from regular files apart.
- We end with `/dev`, which is an argument. This directory is the location of special (or device) files. It is advisable that you do not mess with it, lest you break your OS or worse, lest you break Dataquest!

We see that it is possible to list the content of directories in which we are not located. This isn't always the case, as we might lack permissions to access certain directories.

When we are located in a folder whose contents we wish to list, the argument is optional. In the example above, if we were located in `/dev`, running `ls -p` would yield the same result.

Let's try this command.

- list (`ls`) + hidden files (`-A`) + human readable format (`-h`) + list(`-l`): `ls -Ahl`
- change

In the last exercise, we asked you to use a new option, namely `-a`. This option is different from `-A`, but it is very similar. Relative to the command `ls -A /home/dq/prize_winners` that you ran on a previous screen, the command you ran in the last exercise printed two aditional files: `.` and `..`. If you look at the first character in their respective rows, you'll see a `d`. These are actually directories — very special directories.

Informally, they mean "current directory" and "parent directory" respectively. This is true regardless of where in the directory structure you are! So `cd ..` will take you to the parent directory of your current directory, while `cd .` will just take you to where you already are. In the following example, we start in `/home/learn` and then:

1. We move to the parent directory of our starting directory by running `cd ..`
2. We print the current directory.
3. We re-enter the `learn` directory.
4. We move up to the parent directory of the parent directory of `/home/learn`.

```
/home/learn$ cd ..
/home$ pwd
/home
/home$ cd learn
/home/learn$ cd ../..
/$
```

Running `cd ~` will take you to your home directory.

```
/$ cd ~
/home/dq$
```

And so will `cd`.

```
/home/dq$ cd /
/$ cd
/home/dq$
```

`cd ~` is a special case of the more general `cd ~[username]` which takes us to the home directory of the user we insert instead of `[username]`.

```
/home/dq$ cd ~learn
/home/learn$
```

The command `cd -` takes us to the penultimate argument we used with `cd`. This is useful when need to switch back and forth between two directories that do not have a parent-child relationship.

```
/home/learn$ cd -
/home/dq$ cd -
/home/learn$
```

- make directory: `mkdir <folder_name>`
- remove directory: `rmdir <folder_name>`

- remove file: `rm <file_name>`
- remove directory: `rm -R <directory_name>`

```
rmdir brets
mkdir brats
rmdir dir2 dir3 dir5
```

- copy files from one folder to another: `cp source_files destination`
- copy whole directory: `cp -R <directory_name> <destination_file>`

Syntax

- Creating a directory called `my_dir` : `mkdir my_dir`
- Deleting an empty directory called `my_dir` : `rmdir my_dir`
- Creating a copy of file `my_file1` as `my_file2` : `cp my_file1 my_file2`
- Copying files interactively: `cp - i source destination`
- Create a copy of directory `my_dir1` as `my_dir2` : `cp my_dir1 my_dir2`
- Deleting file `my_file` : `rm my_file`
- Deleting the non-empty directory `my_dir` : `rm- R my_dir`
- Moving `my_file` to `my_dir` : `mv my_file my_dir` .
- Renaming `my_file` as `our_file` : `mv my_file our_file` .

Concepts

- It's not easy to restore files after we delete them from the command line.
- We need to be very careful when using `rm` , `cp` , and `mv` as they might cause us to lose important files.

- list all folders and the subfolder od that folder: `ls *`
- list all folders that have a v: `ls *v`
- list anything that is 4 characters long: `ls ????`

We mentioned that the wildcard `*` behaves like the regex pattern `.*` , despite being different in some cases. In the regular expressions mission, you learned that `.` by itself matches (almost) any character exactly one time, while appending `*` makes it match (almost) any character any number of times.

There's also a wildcard equivalent of `.` – again, with some differences. It is the character `?` .

The wildcard `?` matches any character exactly one time. For example, if we use the pattern `?its` , it will match any filename that is four characters long and ends with `its` . Other examples include `@its` , `fits` , and `hits` , as well as many others.

Here is another example with `ls` .

```
/home/learn$ ls w??t
```

```
west
```

We used the pattern `w??t` , which matches any four-character word that starts with `w` and ends with `t` . Because `west` is the only file/directory with a name that conforms to these rules, the output is `west` .

- find command: `ls * csv`
- Magic commands:

We also have a wildcard that allows us to match specific characters, as opposed to any character. We call it the *square brackets wildcard*. In order to match either `a`, `i`, or `u`, we can use the wildcard `[aiu]`. This will match only one occurrence, just like `?`.

Say we want to list all files or directory content with names that start with either `a`, `i`, or `u`. We can do so by running `ls [aiu]*`. The wildcard `[aiu]` will ensure that the name of the file/directory starts with one of the listed vowels, while `*` will match everything that follows.

We must pass the characters we wish to match into `[]` without separators, as any character will be inside `[]` will take its literal meaning (there are exceptions for specific **sequences** of characters, we'll let you know about them when we learn about variables in the shell in a later course).

Let's see an example. We're going to list all directory content and files with names that satisfy the following requirements:

- First letter must be a vowel.
- Is four characters long.
- Last letter must be `s`, `t`, or `u`.

```
/home/learn$ ls [aAiIuUeEoO]??[stu]
```

```
East
```

Let's break this pattern down:

- First letter must be a vowel: begin pattern with `[aAiIuUeEoO]`.
- Is four characters long: ensure that pattern matches four characters by using `?` appropriately.
- Last letter must be `s`, `t`, or `u`: end pattern with `[stu]`.

Since lowercase letters are different characters from uppercase ones, we need to take this into account in the patterns we create.

The behavior of `[]` is similar in regular expressions. If you recall from regular expressions, the pattern `[^aiu]` matches any character that isn't `a`, `i`, or `u`. Wildcards are endowed with a similar power, only it is `!` instead of `^`. So, for example, the pattern `Data[!qQ]uest` will match any variation of the pattern `Data?uest`, in which the fifth character **is not** `q` **and is not** `Q`.

We should also be mindful of what happens when we use a pattern that doesn't match anything together with `ls`.

```
/home/learn$ ls ???
```

```
ls: cannot access ???: No such file or directory
```

Since there aren't any files or directories in the working directory with names that are three characters long, `???` didn't match any filename, and so `???` was passed as an argument to `ls` without its special meaning. This led the shell to try and list a file/directory that was literally called `???`, hence the error message.

Make sure you're in `brats`.

1. List all the files and directory content with names that do not end with a lowercase vowel.

- `/home/dq/brats$ ls *[!aeiou]`

- Copy files from a path to a different one using `rsync` and `--exclude`: `rsync -avr --exclude '*.docx' /mnt/c/test1/* /mnt/c/Users/Isabel\ F\ Freitas/documentation/mkdocs/docs/`

```
897  rsync -avr --exclude '*.docx' /mnt/c/test1/*  /mnt/c/Users/Isabel\ F\ Freitas/documentation/mkdocs/docs/
```

- using grep to find a specific command in history: `history | grep -e rsync` (this command will look through all the history and find any commands that match `rsync`

```
(docvenv) isabel@LAPTOP-7I5Q0I4A /mnt/c/Users/Isabel F Freitas/documentation/doc $ history | grep -e rsync
  897  rsync -avr --exclude '*.docx' /mnt/c/test1/*  /mnt/c/Users/Isabel\ F\ Freitas/documentation/mkdocs/docs/
```