Tips for a faster workflow in Linux

(version 0.1)

A. Ridolfi[1]

[1]Max-Planck-Institut für Radioastronomie MPIfR, Auf dem Hügel 69, D-53121 Bonn, Germany

1 Looping over files in bash

Very often you will have to loop some particular operation on a bunch of similar files. In this example, I will fold all the .dat files (PRESTO de-dispersed time series) with `prepfold`, at once:

```
> for f in *.dat; do prepfold -noxwin -timing ephemeris.par ${f}; done
```

Using this syntax, I also defined my own function "`loop`", in my `.bashrc`, this way:

```
function loop() { for file in $2; do $1 $file; done ; }
```

In this way, the previous command would be equivalently run like this:

```
> loop "prepfold -noxwin -timing ephemeris.par" "*.dat"
```

2 The `.inputrc` file

In addition to the famous `.bahsrc` file, there is another file in your home directory that can be very useful to customize your interaction with the terminal. This is the `.inputrc` file.
My personal `.inputrc` looks like this:

```
## arrow up
"\e[A":history-search-backward
## arrow down
"\e[B":history-search-forward
```

This will allow me to scroll back through my command history, with an "autocompletion" function. For instance, let's imagine I used the following commands, in the following order:

```
> echo "Ciao"
> echo "Sono"
> echo "Alessandro"
> emacs myfile.txt
> yes "ciao sono ale"
```

After, this, if I type:

```
> e
```

and then I press the [UP-ARROW] key on my keyboard, the terminal will first suggest me:

```
> emacs myfile.txt
```

Pressing [UP-ARROW] again, I will get:

```
> echo "Alessandro"
```

and again:

```
> echo "Sono"
```

If I type:

```
> y_
```

and then [UP-ARROW], then I will get:

```
> yes "ciao sono ale"
```

I guess there are many more customizations you can set with the `.inputrc`. It's up to you to look for them!

3 Fast, passwordless logins

I want to be able to connect via ssh to a frequently used server without needing to type in the password every time. It may seem a tiny amount of time saved but, believe me, when you have to open 5-6 terminals and have to type in the password each time, it can be annoying. So, here is how to save time. In the following example, I am working on my workstation, called `ryzen`, where I am the user `alex`. From this machine, I want to login as `ridolfi` to a server called `portal`. Schematizing:

```
alex@ryzen --> NO PASSWORD --> ridolfi@portal
```

To do so:

1. As user `alex` on the machine `ryzen`, type:

   ```
   ssh-keygen -t rsa
   ```

2. This commands generates a public key. Find it by typing:

3. As user `alex` on the machine `ryzen`, type:

   ```
   [alex@ryzen]> cat ~/.ssh/id_rsa.pub
   ```

   You will get an output similar to this, but most likely longer:

   ```
   ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEArB2U//
   9rppXz3b+X4iGYcB/AFygOr3duiagHxJVSFPar/
   RBmf6mEaTCT1ZOyJclX/VyHHHH== alex@ryzen
   ```

   Copy this string.

4. Now log into `portal` as `ridolfi`:

   ```
   > ssh ridolfi@portal
   ```

   and type in your password as usual. Open the file `authorized_keys` in your `/.ssh` folder with a text editor, for example `emacs`:

   ```
   ridolfi@portal> emacs ~/.ssh/authorized_keys
   ```

   and paste the key at the end of the file. Save it and logout.

From now on, by doing:

```
alex@ryzen> ssh ridolfi@portal
```

you will be able to login without being asked the password!

4 SSH through multiple computers

Very often I need to connect to a computer which is visible only from a server that interconnects with the rest of the internet. As a practical example, I need to work on the computer `miraculix`, which is only accessible from the server `portal`: From now on, by doing:

```
me −−−> portal −−−> miraculix
```

Now, I would like to access `miraculix` with a single command, and not doing a first `ssh` to `portal` and another `ssh` to `miraculix`. The syntax to do such a thing is:

```
> ssh −X −t ridolfi@portal ssh −X −t ridolfi@miraculix
```

If needed, I will be asked for the password(s) (but you can avoid this by using the passwordless logins, see Section 3), but then I will directly be in `miraculix`.

5 Rsync via intermediate server

Sometimes I want to transfer some files from a computer that is accessible only via an intermediate server. For example, I want to copy files located in a directory called `data` in the computer `miraculix`. The latter can be seen only after logging into the server `portal`. The command `rsync` allows me to do so, by using the option "`-e`":

```
> rsync −e "ssh alex@portal ssh" alex@miraculix:/data ./
```

6 Screen

The linux command `screen` can be considered the equivalent of a VNC session, but on a terminal, rather than on a graphical interface. With `screen`, you can launch processes or work on a machine, disconnect, and re-connect later to easily resume your work. This is very good, for example, if you have to run some heavy code that takes hours to finish. You can launch the code from your office from inside a screen, go home, have your dinner, and check the progress from your place by simply attaching to that screen. You can also create as many screens as you want,for multiple work sessions. How to use it? Here are some basic commands.

1. Create a new screen.

   ```
   > screen −S my_screen
   ```

   The option -S is followed by the name of the new screen I want to create. After creating a screen, I am immediately *inside* the screen itself. Here I can launch my process. I can then disconnect from the screen by typing CTRL+A CTRL+D.

2. List the existing screens.

```
> screen −ls

There are screens on:
698.test_screen (Detached)
711.my_screen (Detached)
```

The existing screens are `698.test_screen` and `711.my_screen` The prefix number is added automatically to the name I gave.

3. Attach to an existing screen.

```
> screen −x 711.my_screen
```

This way I will ”connect” (or ”attach”) to the existing screen and I will be able to resume my work there. Obviously, you must be logged into the machine where I created the screen, to be able to attach to it.

7 Managing processes in background

Sometimes it happens that I am editing a file on `emacs` but then I need to quickly check something on the terminal. Instead of quitting `emacs` and re-opening the file, I can just put emacs into background. This is done by typing CTRL+Z from `emacs` (or from whatever program you are using).
When I put a process into background, its execution is halted. That means that, if that process was doing some operations, these will be stopped. To keep the process in the background, but allowing it to run anyways, I can type:

```
> bg
```

To bring the process back to the foreground, thus being able to interact with it, I will instead use the command:

```
> fg
```

I can put more than a process to background, by by typing CTRL+Z from each of them. To see which programs are currently in background, I use:

```
> jobs
[1] Stopped emacs ciao
[2]− Stopped top
[3]+ Stopped less .bash_profile
```

Each program is identified by a number on the left (1, 2, 3) and its status is reported. ”Stopped” means that its execution is halted. The identifiers are used to specify which process you want to control with `bg` and `fg`:

```
> fg 3
```

8 Executable python scripts

When I write a python code, I don’t like to run it by invoking python:

```
> python my_code.py
```

Too long. Instead, I would like it to behave as an executable, exactly like a normal compiled C code. To do so, puth the string

```
#!/usr/bin/env python
```

at the very first line of your python code. After that, give your code executable permissions.

```
> chmod a+x my_code.py
```

From now on, I will be able to run the code by simply typing:

```
> ./my_code.py
```

or, if I add its location in my PATH environment variable, by:

```
> my_code.py
```

from any directory.

9 Useful linux commands

Here is a collection of useful Linux commands.

1. `paste`: glue together the columns of different text files.
2. `cut`: select specific character ranges from the lines of a file.
3. `awk`: perform sophisticated operations on text files.
4. `top` / `htop`: manage your processes.
5. `du`: check how large are your files and directories.
6. `df`: check space usage of attached drives.
7. `find`: look for files.
8. `grep`: look for words inside text files.
9. `cat` / `more` / `less`: view text files.
10. `alias`: make shortened versions of your most commonly used commands.
11. `wc`: count lines, words and characters in a text file.
12. `tee`: count lines, words and characters in a text file.

10 Preventing accidental file deletion

In your `.bashrc` you can set an environment variable that prevents you from inadvertently re-use `rm` commands:

```
export HISTIGNORE="*rm␣*:pwd"
```

With another one, you can avoid having duplicate entries in your history:

```
export HISTCONTROL="ignoredups:ignoredups:ignorespace"
```

With the command `ignorespace` you will tell bash to not remember any command that started with a space. This can be useful if you execute a dangerous command (other than those specified in HISTIGNORE) that you intentionally want your system to forget.