Unix manipulation tutorial

Experiential Data science for Undergraduate Cross-disciplinary Education

Dr. Kim Dill-McFarland, U. of British Columbia

Contents

Unix manipulation tutorial

Learning objectives

- Use Unix commands to manipulate files including mkdir, cp, mv, and rm
- Apply equivalent file paths in Unix commands
- Define best practices for directory and file names

Manipulating files and directories

On your computer, you can open the File Explorer/Finder and perform a number of manipulations on files or directories including creating, copying, moving, or deleting. However, you can also accomplish all of these tasks via command line in quicker, more reproducible ways.

Let's explore some of these in Unix. Open your terminal program of choice to follow along.

Creation

mkdir

The `mkdir` command creates a new, empty directory at the specified path.

First we'll orient ourselves on my computer. I'll move to the Desktop and list its current contents.

```
cd Desktop/
ls -F
```

```
Notes.docx      To do.docx      temp/
```

You will see a list of files on your Desktop. The above files and directories are on mine.

Then we can make a new directory like so.

```
mkdir example/
ls -F
```

```
Notes.docx      To do.docx      example/        temp/
```

If we list in this directory, we can see that it is empty.

```
ls example/
```

Movement

cp (file)

`cp` copies files or directories, leaving the original copy intact in its original location.

You can practice with any file on your computer. Here, we will download the data used in our R tutorials as an example. Remember, we are on the Desktop.

```
write.csv(
  read.csv("https://raw.githubusercontent.com/EDUCE-UBC/workshop_data/master/Saanich_Data.csv"),
  "Saanich_Data.csv", row.names=FALSE)
```

Now that we have the file on our Desktop

```
ls -F
```

```
Notes.docx              To do.docx              temp/
Saanich_Data.csv        example/
```

Note that the terminal will wrap a long list across multiple lines to fit the current window size.

We can copy it into our example directory using the basic syntax

```
cp [source] [target]
```

```
cp Saanich_Data.csv example/
```

And view it in both its original and new locations.

```
ls -F
```

```
Notes.docx              To do.docx              temp/
Saanich_Data.csv        example/
```

```
ls example/
```

```
Saanich_Data.csv
```

mv (file)

In contrast to `cp`, the `mv` command moves files or directories, thus destroying the copy in the original location. However, their syntax and usage are the same.

```
mv Saanich_Data.csv example/
```

And now we see that our csv file no longer exists on the Desktop.

```
ls -F
```

```
Notes.docx      To do.docx      example/        temp/
```

```
ls example/
```

```
Saanich_Data.csv
```

Note that there was no warning or error when the terminal overwrote the copy of ***`Saanich_Data.csv`*** *in the* ***`example/`*** *folder with the one from the* ***`Desktop/`****. In this case, this is okay as the files are exact copies. However, be careful of this when dealing with your own files and versions.*

cp and mv (directory)

In order to, instead, copy or move an entire directory with all of its contents, you need to add the recursive `-R` option to `cp` or `mv`. We do not currently have an directories to copy/move, so here is a non-functional example.

```
cp -R example/ new/target/directory/

# or

mv -R example/ new/target/directory/
```

File paths

Most of the above commands assume your terminal is pointing to your Desktop. This is why we can simply use `example/` to access that directory. However, you can provide full or partial file paths in any command. This improves clarity and prevents you from getting lost in the terminal.

For example, reconsider `cp`. Since our terminal is on the Desktop,

```
pwd
```

```
/Users/kim/Desktop
```

our original command

```
cp Saanich_Data.csv example/
```

is equivalent to

```
cp ~/Desktop/Saanich_Data.csv ~/Desktop/example/
```

which if you recall from our 'Unix navigation tutorial', uses the `~` shortcut and is equivalent to

```
cp /Users/kim/Desktop/Saanich_Data.csv /Users/kim/Desktop/example/
```

Now, this command is getting quite long but it provides exact paths that may be helpful if you share your code (or future you wants to revisit these files).

In the end, it is up to you to decide how exact you want your file paths to be and this decision amy differ depending on the project, computer, server, etc.

Deletion

rm

`rm` removes files or directories. It should be used with **extreme caution** because the removed files **do not** go to your computer's Trash/Recycle Bin. They disappear all together, and it is extremely difficult (if not impossible) to recover them.

This is an example of when full file paths are best in order to avoid errors.

Let's remove the data file first.

```
ls example/
```

```
Saanich_Data.csv
```

```
rm ~/Desktop/example/Saanich_Data.csv
```

And we see that it is gone.

```
ls example/
```

We can also remove the entire example directory using the recursive `-R` function.

```
ls -F
```

```
Notes.docx      To do.docx      example/        temp/
```

```
rm -R example/
```

```
ls
```

```
Notes.docx      To do.docx      temp/
```

Naming conventions

Working with command line forces you to interact with file and directory names directly since you have to type them out instead of clicking. This can reveal some issues with names that you may have never had to content with before.

In this tutorial, we have minimized these issues but when working on your own, here are some best practices to follow.

Spaces & special characters

Terminals denote spaces in a number of different ways. This means that if we have a file named `this is my file.txt`, it would need to be called in the terminal with `this is my file.txt` or `this\is\my\file.txt` or `this\ is\ my\ file.txt` and only one would work for a given system.

Specific formatting is also required for symbols like `()`, `!`, `@`, etc. This is not only inefficient but also causes issues when switching from one terminal system to another.

Solution: Use `_`

The simple solution is to avoid using spaces and special characters. Instead, it is best to use `_` as a separator. If we use `this_is_my_file.txt`, we can easily call the file in the command line without any issues no matter what system we are working in.

Other options you may have seen include `-` or `.`. The dash is not ideal because many command line programs use `-` to indicate options such as when we use `ls -F`. Similarly, the period is not ideal because file extensions like `.txt` follow a period. This does not cause issues per say, but if you choose to use `_` instead, you will more easily be able to determine file types as it will be the last and only text after a period. You can also save the `.` for dates or version numbers (more on this below).

'Final' files

You may want to designate the final version of a file you've been working on. You're first instinct may be to label it `this_is_my_file_FINAL.txt`. Resist this instinct! Don't be this guy! Inevitably, you will need to make some change in the future and will curse past you for calling the file 'final'.

Solution: Use 'submitted'

Instead, consider terms like `submitted` or `turned_in` if you want to specify that a certain version was complete in some way. And don't rename a file in this way until you've completed the submission task. This way the `submitted` version is truly what you submitted.

Dated files

It is unlikely that you will create a file, work on it once, and save it as a final, perfect version - never to be edited again. So, you may want to keep track of versions. This as important, if not more so, for yourself as it is for someone else using your files, because "Your closest collaborator is you 6 months ago. And past you doesn't answer emails" Karl Broman, U. of Wisconsin-Madison.

A common attempt at version control is to label each new version with a date. However, this leaves you with no reference for which changes occurred on which dates. How do you remember which date is which version, and more to the point, how would someone else know?

Solution: Version numbering

Instead of using dates, consider using a version numbering system. A common one in software development is MAJOR.MINOR.PATCH, which denote a:

- MAJOR version when you make incompatible changes
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes

So if you wanted to use this system for a manuscript, for example, you could denote a new

- MAJOR version when you add a new section like starting to write the results
- MINOR version when you add to or significantly change current sections like incorporating co-author comments or rewriting a paragraph
- PATCH version when you make small changes like correcting typos, word-smithing, etc.

Solution: Full version control

The next level of version control is to use a distributed version control system like Git. Think of these systems like the "track changes" feature of MS Word, except there is a record of the changes after you "accept" them.

The advantages of distributed version control over file naming methods are that:

- it is easy to see what changed when (time-stamped), why (commented), and by whom (author identified)
- you can jump back to any point in time since the file's creation, not just versions you deemed important enough to save at the time
- you have the freedom to experiment because you can always go back to the last known good version
- you can work on 2+ different versions in parallel
- you are able manage contributions from multiple people

If you want to learn more about Git and version control, checkout our 'Git tutorial'
