Working With Files and Directories

Overview

Teaching: 30 min
Exercises: 20 min
Questions

- How can I create, copy, and delete files and directories?
- How can I edit files?

Objectives

- Create a directory hierarchy that matches a given diagram.
- Create files in that hierarchy using an editor or by copying and renaming existing files.
- Delete, copy and move specified files and/or directories.

Creating directories

We now know how to explore files and directories, but how do we create them in the first place?

Step one: see where we are and what we already have

Let's go back to our `data-shell` directory on the Desktop and use `ls -F` to see what it contains:

Bash

```
$ pwd
```

Output

```
/Users/nelle/Desktop/data-shell
```

Bash

```
$ ls -F
```

Output

```
creatures/  data/  molecules/  north-pacific-gyre/  notes.txt  pizza.cfg  solar.pdf  writing/
```

Create a directory

Let's create a new directory called `thesis` using the command `mkdir thesis` (which has no output):

Bash

```
$ mkdir thesis
```

As you might guess from its name, `mkdir` means 'make directory'. Since `thesis` is a relative path (i.e., does not have a leading slash, like `/what/ever/thesis`), the new directory is created in the current working directory:

Bash

```
$ ls -F
```

Output

```
creatures/  data/  molecules/  north-pacific-gyre/  notes.txt  pizza.cfg  solar.pdf  thesis/  writing/
```

Note that `mkdir` is not limited to creating single directories one at a time. The `-p` option allows `mkdir` to create a directory with any number of nested subdirectories in a single operation:

Bash

```
$ mkdir -p thesis/chapter_1/section_1/subsection_1
```

The `-R` option to the `ls` command will list all nested subdirectories wtihin a directory. Let's use `ls -FR` to recursively list the new directory hierarchy we just created beneath the `thesis` directory:

Bash

```
$ ls -FR thesis
chapter_1/

thesis/chapter_1:
section_1/

thesis/chapter_1/section_1:
subsection_1/

thesis/chapter_1/section_1/subsection_1:
```

📌 Two ways of doing the same thing

Using the shell to create a directory is no different than using a file explorer. If you open the current directory using your operating system's graphical file explorer, the `thesis` directory will appear there too. While the shell and the file explorer are two different ways of interacting with the files, the files and directories themselves are the same.

📌 Good names for files and directories

Complicated names of files and directories can make your life painful when working on the command line. Here we provide a few useful tips for the names of your files.

1. Don't use spaces.

   Spaces can make a name more meaningful, but since spaces are used to separate arguments on the command line it is better to avoid them in names of files and directories. You can use `-` or `_` instead (e.g. `north-pacific-gyre/` rather than `north pacific gyre/`).
2. Don't begin the name with `-` (dash).

   Commands treat names starting with `-` as options.
3. Stick with letters, numbers, `.` (period or 'full stop'), `-` (dash) and `_` (underscore).

   Many other characters have special meanings on the command line. We will learn about some of these during this lesson. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have spaces or other special characters, you should surround the name in quotes (`""`).

Since we've just created the `thesis` directory, there's nothing in it yet:

Bash

```
$ ls -F thesis
```

Create a text file

Let's change our working directory to `thesis` using `cd`, then run a text editor called Nano to create a file called `draft.txt`:

Bash

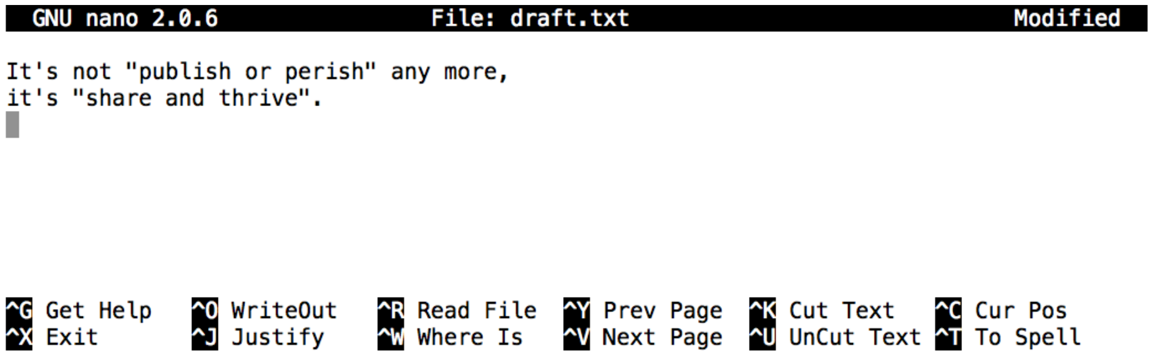
```
$ cd thesis
$ nano draft.txt
```

Which Editor?

When we say, ‘`nano` is a text editor’ we really do mean ‘text’: it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because it is one of the least complex text editors. However, because of this trait, it may not be powerful enough or flexible enough for the work you need to do after this workshop. On Unix systems (such as Linux and macOS), many programmers use Emacs (http://www.gnu.org/software/emacs/) or Vim (http://www.vim.org/) (both of which require more time to learn), or a graphical editor such as Gedit (http://projects.gnome.org/gedit/). On Windows, you may wish to use Notepad++ (http://notepad-plus-plus.org/). Windows also has a built-in editor called `notepad` that can be run from the command line in the same way as `nano` for the purposes of this lesson.

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer's start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you ‘Save As…’

Let's type in a few lines of text. Once we're happy with our text, we can press Ctrl+O (press the Ctrl or Control key and, while holding it down, press the O key) to write our data to disk (we'll be asked what file we want to save this to: press Return to accept the suggested default of `draft.txt`).



```
  GNU nano 2.0.6              File: draft.txt                      Modified

It's not "publish or perish" any more,
it's "share and thrive".




^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Once our file is saved, we can use Ctrl+X to quit the editor and return to the shell.

Control, Ctrl, or ^ Key

The Control key is also called the 'Ctrl' key. There are various ways in which using the Control key may be described. For example, you may see an instruction to press the `Control` key and, while holding it down, press the `X` key, described as any of:

- `Control-X`
- `Control+X`
- `Ctrl-X`
- `Ctrl+X`
- `^X`
- `C-x`

In nano, along the bottom of the screen you'll see `^G Get Help ^O WriteOut`. This means that you can use `Control-G` to get help and `Control-O` to save your file.

`nano` doesn't leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt`:

Bash

```
$ ls
```

Output

```
draft.txt
```

Creating Files a Different Way

We have seen how to create text files using the `nano` editor. Now, try the following command:

Bash

```
$ touch my_file.txt
```

1. What did the `touch` command do? When you look at your current directory using the GUI file explorer, does the file show up?
2. Use `ls -l` to inspect the files. How large is `my_file.txt`?
3. When might you want to create a file this way?

Solution

What's In A Name?

You may have noticed that all of Nelle's files are named 'something dot something', and in this part of the lesson, we always used the extension `.txt`. This is just a convention: we can call a file `mythesis` or almost anything else we want. However, most people use two-part names most of the time to help them (and their programs) tell different kinds of files apart. The second part of such a name is called the **filename extension**, and indicates what type of data the file holds: `.txt` signals a plain text file, `.pdf` indicates a PDF document, `.cfg` is a configuration file full of parameters for some program or other, `.png` is a PNG image, and so on.

This is just a convention, albeit an important one. Files contain bytes: it's up to us and our programs to interpret those bytes according to the rules for plain text files, PDF documents, configuration files, images, and so on.

Naming a PNG image of a whale as `whale.mp3` doesn't somehow magically turn it into a recording of whalesong, though it *might* cause the operating system to try to open it with a music player when someone double-clicks it.

Moving files and directories

Returning to the `data-shell` directory,

Bash

```
cd ~/Desktop/data-shell/
```

In our `thesis` directory we have a file `draft.txt` which isn't a particularly informative name, so let's change the file's name using `mv`, which is short for 'move':

Bash

```
$ mv thesis/draft.txt thesis/quotes.txt
```

The first argument tells `mv` what we're 'moving', while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `thesis` now contains one file called `quotes.txt`:

Bash

```
$ ls thesis
```

Output

```
quotes.txt
```

One has to be careful when specifying the target file name, since `mv` will silently overwrite any existing file with the same name, which could lead to data loss. An additional option, `mv -i` (or `mv --interactive`), can be used to make `mv` ask you for confirmation before overwriting.

Note that `mv` also works on directories.

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll use just the name of a directory as the second argument to tell `mv` that we want to keep the filename, but put the file somewhere new. (This is why the command is called 'move'.) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

Bash

```
$ mv thesis/quotes.txt .
```

The effect is to move the file from the directory it was in to the current working directory. `ls` now shows us that `thesis` is empty:

Bash

```
$ ls thesis
```

Further, `ls` with a filename or directory name as an argument only lists that file or directory. We can use this to see that `quotes.txt` is still in our current directory:

Bash

```
$ ls quotes.txt
```

Output

```
quotes.txt
```

Moving Files to a new folder

After running the following commands, Jamie realizes that she put the files `sucrose.dat` and `maltose.dat` into the wrong folder. The files should have been placed in the `raw` folder.

Bash

```
$ ls -F
 analyzed/ raw/
$ ls -F analyzed
fructose.dat glucose.dat maltose.dat sucrose.dat
$ cd analyzed
```

Fill in the blanks to move these files to the `raw/` folder (i.e. the one she forgot to put them in)

Bash

```
$ mv sucrose.dat maltose.dat ____/____
```

Solution

Copying files and directories

The `cp` command works very much like `mv`, except it copies a file instead of moving it. We can check that it did the right thing using `ls` with two paths as arguments — like most Unix commands, `ls` can be given multiple paths at once:

Bash

```
$ cp quotes.txt thesis/quotations.txt
$ ls quotes.txt thesis/quotations.txt
```

Output

```
quotes.txt   thesis/quotations.txt
```

We can also copy a directory and all its contents by using the recursive (https://en.wikipedia.org/wiki/Recursion) option `-r`, e.g. to back up a directory:

Bash

```
$ cp -r thesis thesis_backup
```

We can check the result by listing the contents of both the `thesis` and `thesis_backup` directory:

Bash

```
$ ls thesis thesis_backup
```

Output

```
thesis:
quotations.txt

thesis_backup:
quotations.txt
```

Renaming Files

Suppose that you created a plain-text file in your current directory to contain a list of the statistical tests you will need to do to analyze your data, and named it: `statstics.txt`

After creating and saving this file you realize you misspelled the filename! You want to correct the mistake, which of the following commands could you use to do so?

1. `cp statstics.txt statistics.txt`
2. `mv statstics.txt statistics.txt`
3. `mv statstics.txt .`
4. `cp statstics.txt .`

Solution

Moving and Copying

What is the output of the closing `ls` command in the sequence shown below?

Bash

```
$ pwd
```

Output

```
/Users/jamie/data
```

Bash

```
$ ls
```

Output

```
proteins.dat
```

Bash

```
$ mkdir recombined
$ mv proteins.dat recombined/
$ cp recombined/proteins.dat ../proteins-saved.dat
$ ls
```

1. `proteins-saved.dat recombined`
2. `recombined`
3. `proteins.dat recombined`
4. `proteins-saved.dat`

Solution

Removing files and directories

Returning to the `data-shell` directory, let's tidy up this directory by removing the `quotes.txt` file we created. The Unix command we'll use for this is `rm` (short for 'remove'):

Bash

```
$ rm quotes.txt
```

We can confirm the file has gone using `ls` :

Bash

```
$ ls quotes.txt
```

Output

```
ls: cannot access 'quotes.txt': No such file or directory
```

Deleting Is Forever

The Unix shell doesn't have a trash bin that we can recover deleted files from (though most graphical interfaces to Unix do). Instead, when we delete files, they are unlinked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away.

Using `rm` Safely

What happens when we execute `rm -i thesis_backup/quotations.txt` ? Why would we want this protection when using `rm` ?

Solution

If we try to remove the `thesis` directory using `rm thesis` , we get an error message:

Bash

```
$ rm thesis
```

Error

```
rm: cannot remove `thesis': Is a directory
```

This happens because `rm` by default only works on files, not directories.

`rm` can remove a directory *and all its contents* if we use the recursive option `-r` , and it will do so *without any confirmation prompts*:

Bash

```
$ rm -r thesis
```

Given that there is no way to retrieve files deleted using the shell, `rm -r` *should be used with great caution* (you might consider adding the interactive option `rm -r -i`).

Operations with multiple files and directories

Oftentimes one needs to copy or move several files at once. This can be done by providing a list of individual filenames, or specifying a naming pattern using wildcards.

Copy with Multiple Filenames

For this exercise, you can test the commands in the `data-shell/data` directory.

In the example below, what does `cp` do when given several filenames and a directory name?

Bash

```
$ mkdir backup
$ cp amino-acids.txt animals.txt backup/
```

In the example below, what does `cp` do when given three or more file names?

Bash

```
$ ls -F
```

Output

```
amino-acids.txt  animals.txt  backup/  elements/  morse.txt  pdb/  planets.txt  salmon.txt  sunspot.txt
```

Bash

```
$ cp amino-acids.txt animals.txt morse.txt
```

Solution

Using wildcards for accessing multiple files at once

Wildcards

`*` is a **wildcard**, which matches zero or more characters. Let's consider the `data-shell/molecules` directory: `*.pdb` matches `ethane.pdb` , `propane.pdb` , and every file that ends with '.pdb'. On the other hand, `p*.pdb` only matches `pentane.pdb` and `propane.pdb` , because the 'p' at the front only matches filenames that begin with the letter 'p'.

`?` is also a wildcard, but it matches exactly one character. So `?ethane.pdb` would match `methane.pdb` whereas `*ethane.pdb` matches both `ethane.pdb` , and `methane.pdb` .

Wildcards can be used in combination with each other e.g. `???ane.pdb` matches three characters followed by `ane.pdb` , giving `cubane.pdb ethane.pdb octane.pdb` .

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example typing `ls *.pdf` in the `molecules` directory (which contains only files with names ending with `.pdb`) results in an error message that there is no file called `*.pdf` . However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

List filenames matching a pattern

When run in the `molecules` directory, which `ls` command(s) will produce this output?

```
ethane.pdb methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`
3. `ls *t??ne.pdb`
4. `ls ethane.*`

Solution

More on Wildcards

Sam has a directory containing calibration data, datasets, and descriptions of the datasets:

Bash

```
.
├── 2015-10-23-calibration.txt
├── 2015-10-23-dataset1.txt
├── 2015-10-23-dataset2.txt
├── 2015-10-23-dataset_overview.txt
├── 2015-10-26-calibration.txt
├── 2015-10-26-dataset1.txt
├── 2015-10-26-dataset2.txt
├── 2015-10-26-dataset_overview.txt
├── 2015-11-23-calibration.txt
├── 2015-11-23-dataset1.txt
├── 2015-11-23-dataset2.txt
├── 2015-11-23-dataset_overview.txt
├── backup
│   ├── calibration
│   └── datasets
└── send_to_bob
    ├── all_datasets_created_on_a_23rd
    └── all_november_files
```

Before heading off to another field trip, she wants to back up her data and send some datasets to her colleague Bob. Sam uses the following commands to get the job done:

Bash

```
$ cp *dataset* backup/datasets
$ cp ____calibration____ backup/calibration
$ cp 2015-____-____ send_to_bob/all_november_files/
$ cp ____ send_to_bob/all_datasets_created_on_a_23rd/
```

Help Sam by filling in the blanks.

The resulting directory structure should look like this

Bash

```
.
├── 2015-10-23-calibration.txt
├── 2015-10-23-dataset1.txt
├── 2015-10-23-dataset2.txt
├── 2015-10-23-dataset_overview.txt
├── 2015-10-26-calibration.txt
├── 2015-10-26-dataset1.txt
├── 2015-10-26-dataset2.txt
├── 2015-10-26-dataset_overview.txt
├── 2015-11-23-calibration.txt
├── 2015-11-23-dataset1.txt
├── 2015-11-23-dataset2.txt
├── 2015-11-23-dataset_overview.txt
├── backup
│   ├── calibration
│   │   ├── 2015-10-23-calibration.txt
│   │   ├── 2015-10-26-calibration.txt
│   │   └── 2015-11-23-calibration.txt
│   └── datasets
│       ├── 2015-10-23-dataset1.txt
│       ├── 2015-10-23-dataset2.txt
│       ├── 2015-10-23-dataset_overview.txt
│       ├── 2015-10-26-dataset1.txt
│       ├── 2015-10-26-dataset2.txt
│       ├── 2015-10-26-dataset_overview.txt
│       ├── 2015-11-23-dataset1.txt
│       ├── 2015-11-23-dataset2.txt
│       └── 2015-11-23-dataset_overview.txt
└── send_to_bob
    ├── all_datasets_created_on_a_23rd
    │   ├── 2015-10-23-dataset1.txt
    │   ├── 2015-10-23-dataset2.txt
    │   ├── 2015-10-23-dataset_overview.txt
    │   ├── 2015-11-23-dataset1.txt
    │   ├── 2015-11-23-dataset2.txt
    │   └── 2015-11-23-dataset_overview.txt
    └── all_november_files
        ├── 2015-11-23-calibration.txt
        ├── 2015-11-23-dataset1.txt
        ├── 2015-11-23-dataset2.txt
        └── 2015-11-23-dataset_overview.txt
```

Solution

Organizing Directories and Files

Jamie is working on a project and she sees that her files aren't very well organized:

Bash

```
$ ls -F
```

Output

```
analyzed/  fructose.dat    raw/   sucrose.dat
```

The `fructose.dat` and `sucrose.dat` files contain output from her data analysis. What command(s) covered in this lesson does she need to run so that the commands below will produce the output shown?

Bash

```
$ ls -F
```

Output

```
analyzed/   raw/
```

Bash

```
$ ls analyzed
```

Output

```
fructose.dat    sucrose.dat
```

Solution

Reproduce a folder structure

You're starting a new experiment, and would like to duplicate the directory structure from your previous experiment so you can add new data.

Assume that the previous experiment is in a folder called '2016-05-18', which contains a `data` folder that in turn contains folders named `raw` and `processed` that contain data files. The goal is to copy the folder structure of the `2016-05-18-data` folder into a folder called `2016-05-20` so that your final directory structure looks like this:

```
2016-05-20/
└── data
    ├── processed
    └── raw
```

Which of the following set of commands would achieve this objective? What would the other commands do?

Bash

```
$ mkdir 2016-05-20
$ mkdir 2016-05-20/data
$ mkdir 2016-05-20/data/processed
$ mkdir 2016-05-20/data/raw
```

Bash

```
$ mkdir 2016-05-20
$ cd 2016-05-20
$ mkdir data
$ cd data
$ mkdir raw processed
```

Bash

```
$ mkdir 2016-05-20/data/raw
$ mkdir 2016-05-20/data/processed
```

Bash

```
$ mkdir -p 2016-05-20/data/raw
$ mkdir -p 2016-05-20/data/processed
```

Bash

```
$ mkdir 2016-05-20
$ cd 2016-05-20
$ mkdir data
$ mkdir raw processed
```

Solution

Key Points

- `cp old new` copies a file.
- `mkdir path` creates a new directory.
- `mv old new` moves (renames) a file or directory.
- `rm path` removes (deletes) a file.
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt` .
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not `any.txt` .
- Use of the Control key may be described in many ways, including `Ctrl-X` , `Control-X` , and `^X` .
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Most files' names are `something.extension` . The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Depending on the type of work you do, you may need a more powerful text editor than Nano.

‹ (../02-filedir/index.html)

› (../04-pipefil