title: The Unix Shell subtitle: Creating Things

minutes: 20

The Unix Shell: Creating Things

Learning 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.
- Display the contents of a directory using the command line.
- Delete specified files and/or directories.

Creating Things

We now know how to explore files and directories, but how do we create them in the first place? Let's go back to programming-fundamentals, /home/oski/Desktop/programming-fundamentals,

and use 1s -F to see what it contains:

\$ pwd

/home/oski/Desktop/programming-fundamentals

\$ ls -F

0-0_Introduction.md	1-2_create.md	README.md
0-1_BCE.md	1-3_pipe.md	data/
0-2_help.md	1-4_loop.md	resource.md
1-0_shell.md	1-5_scripts.md	test/
1-1_fildir.md	LICENSE	

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

\$ mkdir thesis

As you might (or might not) guess from its name, mkdir means "make directory". Since thesis is a relative path (i.e., doesn't have a leading slash), the new directory is created in the current working directory:

\$ ls -F

0-0_Introduction.md	1-2_create.md	README.md	
0-1_BCE.md	1-3_pipe.md	data/	
0-2_help.md	1-4_loop.md	resource.md	
1-0_shell.md	1-5_scripts.md	test/	
1-1_fildir.md	LICENSE	thesis/	

However, there's nothing in it yet:

```
$ ls -F thesis
```

Text Editors

Let's change our working directory to thesis using cd, then use the command touch to create a file called draft.txt, and run a text editor called Nano to edit the file:

```
$ cd thesis
$ touch draft.txt
$ 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 almost anyone can drive it anywhere without training, but please use something more powerful for real work. On Unix systems (such as Linux and Mac OS X), many programmers use Emacs (http://www.gnu.org/software/emacs/) or Vim (http://www.vim.org/) (both of which are completely unintuitive, even by Unix standards), or a graphical editor such as Gedit (http://projects.gnome.org/gedit/), which is on BCE. On Windows, you may wish to use <u>Notepad++ (http://notepad-plus-plus.org/</u>).

Text editors are not limited to .txt files. Code is also text - so any file with an extension like .py (for python) .sh (for shell) can also be edited in a text editor. So can files containing markup, like .html (for HTML) or .md (for markdown). Markup is a way to format text (bold, lists, links, etc) using simple syntax.

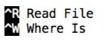
Let's type in a few lines of text, then use Control-O to write our data to disk:

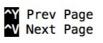
```
GNU nano 2.0.6
                              File: draft.txt
                                                                        Modified
```

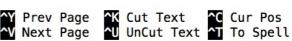
```
It's not "publish or perish" any more,
it's "share and thrive".
```













Once our file is saved, we can use Control-X to quit the editor and return to the shell. (Unix documentation often uses the shorthand ^A to mean "control-A".) 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:

\$ ls

draft.txt

Alternative: Notepad or TextEdit

You'll need to use a Unix text editor like nano, emacs or vim when you only have access to the shell (such as when you're logged in remotely to another computer through SSH, for cloud computing). However, if you're working with text files located on your laptop, you can also use the shell to open a stand-alone application to edit your text files. Try the following:

(Windows)

\$ Notepad draft.txt

(Mac)

\$ open -a TextEdit draft.txt

This should open the file draft.txt in a new Notepad or TextEdit window. Add some text, then click **Save** in the **File** menu. Now close that window and return to the shell.

If you open the same file in nano again, you'll see the text that you added in Notepad or TextEdit. These are just different ways of manipulating the same files on your computer. Notepad or TextEdit require you to be working on your Windows or Mac laptop, while nano or vim can be used when you just have a Unix shell.

Removing

Let's tidy up by running rm draft.txt:

\$ rm draft.txt

This command removes files ("rm" is short for "remove"). If we run 1s again, its output is empty once more, which tells us that our file is gone:

\$ ls

Deleting Is Forever

Unix doesn't have a trash bin: when we delete files, they are unhooked 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.

Let's re-create that file and then move up one directory to /home/oski/Desktop/programming-fundamentals using cd ..:

\$ pwd

/home/oski/Desktop/programming-fundamentals/thesis

\$ touch draft.txt

\$ nano draft.txt

\$ ls

draft.txt

\$ cd ..

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

\$ rm thesis

rm: cannot remove `thesis': Is a directory

This happens because rm only works on files, not directories. The right command is rmdir, which is short for "remove directory". It doesn't work yet either, though, because the directory we're trying to remove isn't empty:

\$ rmdir thesis

rmdir: failed to remove `thesis': Directory not empty

This little safety feature can save you a lot of grief, particularly if you are a bad typist. To really get rid of thesis we must first delete the file draft.txt:

\$ rm thesis/draft.txt

The directory is now empty, so rmdir can delete it:

\$ rmdir thesis

With Great Power Comes Great Responsibility

Removing the files in a directory just so that we can remove the directory quickly becomes tedious. Instead, we can use rm with the -r flag (which stands for "recursive"):

\$ rm -r thesis

This removes everything in the directory, then the directory itself. If the directory contains sub-directories, rm -r does the same thing to them, and so on. It's very handy, but can do a lot of damage if used without care.

Moving

Let's create that directory and file one more time. (Note that this time we're running nano with the path thesis/draft.txt, rather than going into the thesis directory and running nano on draft.txt there.)

\$ pwd

/home/oski/Desktop/programming-fundamentals

\$ mkdir thesis

\$ nano thesis/draft.txt

\$ ls thesis

draft.txt

To see the contents of a file, we can call the command cat, short for concatenate.

\$ cat thesis/draft.txt
hello!

draft.txt isn't a particularly informative name, so let's change the file's name using mv, which is short for "move":

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

The first parameter 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:

\$ ls thesis

quotes.txt

To check it has the same contents, we can use cat again:

\$ cat thesis/quotes.txt
hello!

Just for the sake of inconsistency, mv also works on directories — there is no separate mvdir command.

Let's move quotes.txt into the current working directory. We use mv once again, but this time we'll just use the name of a directory as the second parameter 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.

\$ mv thesis/quotes.txt .

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

\$ ls thesis

Further, 1s with a filename or directory name as a parameter only lists that file or directory. We can use this to see that quotes.txt is still in our current directory:

```
$ ls quotes.txt
```

quotes.txt

Copying

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 1s with two paths as parameters --- like most Unix commands, 1s can be given thousands of paths at once:

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

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

To prove that we made a copy, let's delete the quotes.txt file in the current directory and then run that same ls again. This time it tells us that it can't find quotes.txt in the current directory, but it does find the copy in thesis that we didn't delete:

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

ls: cannot access quotes.txt: No such file or directory thesis/quotations.txt

Rochelle's Pipeline: Organizing and Moving Files

Knowing just this much about files and directories, Rochelle is ready to organize the files for her text project. First, she cd's into the programming-fundamentals directory. From there, she creates a directory called new-york-times (to remind herself where the data came from) inside her data directory. Inside that, she creates a directory called 2015-01-01, which is the date she started processing the texts. She used to use names like conference-paper and revised-results, but she found them hard to understand after a couple of years. (The final straw was when she found herself creating a directory called revised-revised-results-3.)

Rochelle names her directories "year-month-day", with leading zeroes for months and days, because the shell displays file and directory names in alphabetical order. If she used month names, December would come before July; if she didn't use leading zeroes, November ('11') would come before July ('7').

```
$ cd ~/programming-fundamentals/data
$ mkdir new-york-times
$ mkdir new-york-times/2015-01-01
```

Now she's ready to add the text files that she downloaded from LexisNexis into the directory.

The text files that she downloaded are, unsurprisingly, in the directory downloads

```
$ cd downloads
$ ls
```

```
human-rights-2000.TXT human-rights-2004.TXT human-rights-2008.TXT
human-rights-2001.TXT human-rights-2005.TXT human-rights-2009.TXT
human-rights-2002.TXT human-rights-2006.TXT
human-rights-2003.TXT human-rights-2007.TXT
```

Rochelle wants to move them into the directory she just created.

```
$ cp human-rights-2000.TXT ../new-york-times/2015-01-01
$ ls ../new-york-times/2015-01-01
```

```
human-rights-2000.TXT
```

Huzzah! But does Rochelle really have to time in a command for each file she wants to move? No, there's an easier way! Instead of giving an input for each file, Rochelle can write cp *.TXT. The * in *.TXT matches zero or more characters, so the shell turns *.TXT into a complete list of .TXT files

```
$ cp *.TXT ../new-york-times/2015-01-01
$ ls ../new-york-times/2015-01-01
```

```
human-rights-2000.TXT human-rights-2004.TXT human-rights-2008.TXT
human-rights-2001.TXT human-rights-2005.TXT human-rights-2009.TXT
human-rights-2002.TXT human-rights-2006.TXT
human-rights-2003.TXT human-rights-2007.TXT
```

Wildcards {.callout}

* is a wildcard. It matches zero or more characters, so *.pdb matches ethane.pdb, propane.pdb, and so on. On the other hand, p*.pdb only matches pentane.pdb and propane.pdb, because the 'p' at the front only matches itself.

? is also a wildcard, but it only matches a single character. This means that p?.pdb matches pi.pdb or p5.pdb, but not propane.pdb. We can use any number of wildcards at a time: for example, p*.p?* matches anything that starts with a 'p' and ends with '.', 'p', and at least one more character (since the '?' has to match one character, and the final '*' can match any number of characters). Thus, p*.p?* would match preferred.practice, and even p.pi (since the first '*' can match no characters at all), but not quality.practice (doesn't start with 'p') or preferred.p (there isn't at least one character after the '.p').

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 a parameter to the command as it is. For example typing ls *.pdf in the new-york-times directory (which contains only files with names ending with .TXT) 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 another example of orthogonal design.

Exercises

Challenge 1

cd into programming-fundamentals. Create a directory called my_files.

Challenge 2

Within that directory, create a file called script.sh.

Challenge 3

Copy script.sh into my_files/backup/

Challenge 4

The command ls -t returns a listing arranged by time of last edit. Add this command to script.sh.

When you feel you have met these challenges successfully, cd into test/ and type

into the command line. If you were successful, the output will look like this:

```
Challenge 1
...passed
Challenge 2
...passed
Challenge 3
...passed
Challenge 4
...passed
```

Adapted from: <u>Software Carpentry (http://software-carpentry.org/v5/novice/shell/02-create.html)</u>