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-master`, `/home/oski/Desktop/programming-fundamentals-master`,
and use `ls -F` to see what it contains:

```
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals
```

```
$ ls -F
```

```
0-0_Introduction.md 2-1_R-Python.md     mac_file_system.jpg
0-1_howtoprog.md    Install.md      pdfs
0-2_OS.md       LICENSE         regen_pdfs.sh
1-0_shell.md        README.md       resource.md
1-1_fildir.md       answers         test
1-2_create.md       data
```

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

```
$ mkdir scripts
```

As you might (or might not) guess from its name, `mkdir` means "make directory". Since `scripts` 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 2-1_R-Python.md     mac_file_system.jpg
0-1_howtoprog.md    Install.md      pdfs/
0-2_OS.md       LICENSE         regen_pdfs.sh
1-0_shell.md        README.md       resource.md
1-1_fildir.md       answers/        scripts/
1-2_create.md       data/           test/
```

However, there's nothing in it yet:

```
$ ls -F scripts
```

Text Editors

Let's change our working directory to `scripts` using `cd`, then use the command `touch` to create a file called `python_script.py`, and run a text editor called `vim` to edit the file:

```
$ cd scripts
$ touch python_script.py
$ vim python_script.py
```

Which Editor?

When we say, "`vim` 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. 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, or Sublime Text (https://www.sublimetext.com).
On Windows, you may wish to use Notepad++ (http://notepad-plus-plus.org/).

Text editors are not limited to .txt files. Code is also text - so any file with an extension like .py (for python), .R (for R), .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 simple line of Python code, first type `i` to enter `insert` mode in vim, then enter the following:

```
print('Wow! I'm programming!')
```

To `write`, or save, the file, we first type Esc to get out of insert mode, then:

```
:w
```

Then press `Enter`, then:

```
:q
```

and Enter to quit vim. This can also be combined into one command:

```
:wq
```

When we ask for a listing, we can see our script is still there:

```
$ ls
```

```
python_script.py
```

We can also run the program from the shell by calling the python interpreter:

```
python python_script.py
```

```
Wow! I'm programming!
```

Awesome!

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 python_script.py
```

(Mac)

```
$ open -a TextEdit python_script.py
```

This should open the file `python_script.py 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 python_script.py`:

```
$ rm python_script.py
```

This command removes files ("rm" is short for "remove"). If we run `ls` 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 a file and then move up one directory to `/home/oski/Desktop/programming-fundamentals-master` using `cd ..`:

```
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals-master/scripts
```

```
$ touch r_script.R
$ ls
```

```
r_script.R
```

```
$ cd ..
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals-master
```

```
$ ls scripts
```

```
scripts
```

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

```
$ rm scripts
```

```
rm: cannot remove `scripts': 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 scripts
```

```
rmdir: failed to remove `scripts': 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 `scripts` we must first delete the file `r_script.R`:

```
$ rm scripts/r_script.R
```

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

```
$ rmdir scripts
```

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 scripts
```

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.

```
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals-master
```

```
$ mkdir scripts
```

Let's try a different way to add text to a file, first we need to learn the bash command echo:

```
$ echo "Hello world!"
```

```
Hello world!
```

So the echo command will simply return whatever text we give it between quotes. Let's try redirecting this echo into a file with the redirect symbol >:

```
$ echo "print('This is printing in Python.')" > scripts/python_script.py
$ echo "print('This is printing in R.')" > scripts/r_script.R
```

The > tells the shell to **redirect** the command's output, which as we saw above is whatever is between quotes, to a file instead of printing it to the screen. The shell will create the file if it doesn't exist, or overwrite the contents of that file if it does.

```
$ ls scripts
```

```
python_script.py    r_script.R
```

We can run both too:

```
$ python scripts/python_script.py
```

```
This is printing in Python.
```

```
$ Rscript scripts/r_script.R
```

```
[1] "This is printing in R."
```

Coincidentally, Python and R both print with the same syntax (though the return looks slightly different), and many other languages print with a very similar syntax. But you'll soon learn that R and Python each have their own different syntax for most other tasks.

We can also send the content of `python_script.py` to the screen as output using `cat python_script.py`. `cat` stands for "concatenate": it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

```
$ cat scripts/python_script.py
```

```
print('This is printing in Python.')
```

Let's change the file's name using `mv`, which is short for "move":

```
$ mv scripts/python_script.py scripts/python_printing.py
```

The first parameter tells `mv` what we're "moving", while the second is where it's to go. In this case, we're moving `scripts/python_script.py` to `scripts/python_printing.py`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `scripts` now contains one file called `python_printing.py`:

```
$ ls scripts
```

```
python_printing.py  r_script.R
```

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

```
$ cat scripts/python_printing.py
```

```
print('This is printing in Python.')
```

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

Let's move `python_printing.py` 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 scripts/python_printing.py .
```

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

```
$ ls scripts
```

```
r_script.R
```

Further, `ls` with a filename or directory name as a parameter only lists that file or directory. We can use this to see that `python_printing.py` is now in our current directory:

```
$ ls python_printing.py
```

```
python_printing.py
```

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

```
$ cp python_printing.py scripts/python_script.py
$ ls python_printing.py scripts/python_script.py
```

```
python_printing.py scripts/python_script.py
```

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

```
$ rm python_printing.py
$ ls python_printing.py scripts/python_script.py
```

```
ls: cannot access python_printing.py: No such file or directory
python_printing.py
```

Exercises

Challenge 1

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

Challenge 2

Within that directory, create a file called my_script.R.

Challenge 3

Create a directory my_files/backup/ and copy my_script.R into my_files/backup/.

Challenge 4

Have the R script in my_files print "I love programming!" to the command line.

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

```
. 1-2_test.sh
```

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: [Software Carpentry (http://software-carpentry.org/v5/novice/shell/02-create.html)](http://software-carpentry.org/v5/novice/shell/02-create.html)