Command line tools for archivists

There are numerous ways archivists can benefit from using command line tools. In particular, the exercises below will cover conversion of file formats and character encodings, batch renaming of files, searching for files and text within them, and making quick edits to text files. Parts of the cataloger and web developer exercises will likely be of interest, too; be sure to check those out after you're finished here!

Also, a reminder: the workshop leaders are here to help. Put up a sticky note if you need help, or feel free to ask others who are working on the same exercise! We wrote all of this out so you'd have it to refer to later, but we *want* to help while you've got us! If anything's unclear or confusing, let us know!

OK, here we go!

Setup

Navigate (`cd`) to `/home/nitrous/cli-workshop/archiving-exercise/`

Look in the directory (`ls`), just to see what's there.

Run the following command (don't type the >):

```
> ./makefiles.py
```

(When it's finished, it will output one word, "Complete.")

Congratulations, you just ran a Python script from the command line!

Digression: "Why," you might ask, "did we stick `./` in front of the name of the script?" Good question! When you want to run a script, you need to specify the full name of the script, which includes the path. In this case, you could also run it using`/home/nitrous/cli-workshop/archive-exercise/makefiles.py`, but that's a lot to type. And, as we mentioned earlier, when you want to refer to the current working directory, `.` is great shorthand, much easier to type.

Now, look in the directory again, and notice there's a new subdirectory, `c4lfiles`. You'll use this during the "Batch Renaming Files" exercise, later. For now, we're going to do fun things with another file in the directory, `lorem.txt`.

File formatting issues

Let's say you've got some files that aren't in the right character set. They're in ASCII, but you need them to be in UTF-8 Unicode. Or they're in UTF-8 but they need to be in UTF-16.

You can look at what type and format your file is with the `file` command.

Make sure you're in `/home/nitrous/cli-workshop/archiving-exercise/`, and type the following:

```
> file -bi lorem.txt
```

This tells you the *type* of the file `lorem.txt`. The `-b` flag tells it to be brief, which just means it won't put the filename at the beginning of the output.

Including `-i` changes the format of its output to include its (MIME type) [https://en.wikipedia.org/wiki/Media_type] along with, if applicable, the character encoding. To see the difference, go ahead and also type

`> file -b lorem.txt` `> file lorem.txt`

Often having the more precise output is helpful, so getting in the habit of using `-bi` is not a bad choice.

It would be valid, if you preferred, to separate the flags:

```
> file -b -i lorem.txt
```

So now we know our file is UTF-8.

Look inside the file, if you'd like. There are a number of ways to do this (`cat`, `more`, `less`).

```
> more lorem.txt
```

You can scroll slowly by hitting Enter and the up and down arrows; or page down by hitting the spacebar. If you are tired of scrolling, just type the letter `q` to get out.

Digression: the command `clear` will remove all of the old commands and text from your field of view, giving you a nice, clean command line interface. Feel free to use it at any point. You can still hit the up arrow to get to your last command, and then use up and down to scroll back through earlier commands. Nothing's lost; it just makes your screen cleaner.

You can also *change* the formatting of a file with `iconv`.

```
> iconv -f UTF-8 -t UTF-16 lorem.txt > lorem2.txt
```

This makes a copy of `lorem.txt`, in UTF-16 format instead of UTF-8, and saves it as `lorem2.txt`.

Confirm that it changed using the `file` command on `lorem2.txt`.

Another cool usage of iconv is on strings of text, like so:

```
> echo "Êàêèå-òî êðàêîçÿáðû" | iconv -t latin1 | iconv -f cp1251 -t utf-8
```

This should output `Какие-то кракозябры`

What that command is doing is using pipes (`|`) in order to write (`echo`) a string (`Êàêèå-òî êðàêîçÿáðû`) to the command`iconv`, first to convert it to encoding `latin1`, then taking the output of that and feeding it to `iconv` again, to convert it from the encoding `cp1251` to `utf-8`.

(The specifics of why latin1 and cp1251 are used seemingly interchangeably like that are left for those interested enough to dig in. The finer points of character encodings are somewhat beyond the scope of this workshop.)

'iconv' is really useful if you have garbled text (or a whole file of garbled text), and you know, or have a good guess, what encoding it *should* be in, to un-garble it.

Batch renaming files

Let's pretend you have volunteered to archive all of the data from all the Code4Lib conferences up to now. Thanks! Unfortunately, the data were put together by lots of really enthusiastic volunteers, who were not given a complete file naming convention ahead of time. (Let’s pretend there aren’t a ton of metadata experts in the Code4Lib community who would prevent that from happening.) Everything is in CSV (comma-separated values) format.

Navigate to the `c4lfiles` directory (under `/home/nitrous/cli-workshop/archiving-`

`exercise/`), and look at what's in there (`ls`).

As you can see, there are about 100 files there, all from different years. They're kind of a mess. If you were to try renaming all of these files by hand, it would be tedious and take quite a while to do, but we're going to clean this whole thing up with less than five minutes' worth of work.

Now, there are a number of ways to do bulk edits of filenames on a CLI, including writing a script in Python or bash, or doing clever things with UNIX's `find` command. These are all totally valid. We’ll be using the `rename` command today.

If you're working through this on a Mac, rather than on Nitrous, you'll want to follow these directions first. And there'll be one other little snag, later, but we'll get there when we get there.

Rename

Rename uses something called "regular expressions," or as they are commonly referred to, "regex," to match on and change filenames. We are not going into regex in depth today, but we will use some fairly simple regular expressions, which will be explained along the way, to do the work we need to do. `rename` uses the Perl flavor of regular expressions.

```
rename -[flags] [regular expression] files
```

There's one super important flag you should know about, `-n`. That allows you to look at what the command will do, without actually running the command, which can save you from a LOT of pain. It is SO GOOD.

Regex

This is by no means a complete tutorial on regular expressions. You won't walk away, today, as a regex witch or wizard (unless you walked *in* that way), and that's OK. I'm not one, either.

The cool thing about regular expressions is that they're used a whole lot, so if you're trying to do something, you can Google the thing you're trying to do, and someone probably already wrote the regex you need; today's workshop will hopefully at least give you enough of a background to look at someone else's regex and make an educated assessment of whether it might work for you.

It's beyond of the scope of what we're doing today, but, if you'd like to play with regular expressions on your own, RegExr is a really nice interface for doing so. If you're already comfortable with regular expressions, you might enjoy Regex Golf orRegex Crossword.

Here's a list of the regular expression syntax we'll use today (and maybe one we won't, but it's good to know), so you can reference them as we're doing examples:

- `s/x/y/` - You may have seen this one around various tech chats, used as a shorthand for "I meant y, not x." This takes out x and replaces it with y ("s" is for "substitute," but the s/x/y/ pattern is referred to as a "switch statement")
- `.` - this is a special character in regex and means "any character"
- `\` - this is the escape character, allowing you to use a special character as a literal (if you want to match on a period, use `\.`)
- `$` - means "the end of the string" (if it's in the first half of a switch statement)
- `[]` - means "character class," or "match one thing out of what's inside the brackets"; it changes the behavior of special characters, too
- `()` - means "group" and allows a match to be referred to later
- `$` - refers to a previously matched group (e.g. `$1`) (if it's in the second half of a switch statement)
- `*` - means "match the preceding thing any number of times"
- `?` - means "the preceding thing is optional" (match 0 or 1 times)
- `+` - means "match one or more times"

This doesn't have to make a lot of sense to you right now (or ever); it's just a reference, to help you think about the examples below.

Make the file extensions consistent

Some of the volunteers who made all of these files didn't know CSV was a thing, so they saved their files full of comma-separated values as text files, ending in .txt. It's cool, we can fix that right up.

What we want to do is to find every file that ends in ".txt" and change it so it ends in ".csv" instead. **Don't type it in yet**, but the command for that is

```
> rename 's/\.txt$/.csv/' *.txt
```

You have to look at it out of order, to understand what it's doing:

`rename [some regex] *.txt` tells `rename` to operate only on files that end in .txt. This is important and possibly confusing: the `*.txt` at the end of the command is **not** a regular expression. This is the usage of `*` that you're more used to from library databases: it's a wildcard. We're telling `rename` to match on every filename that ends in .txt, no matter what characters come before the period.

As you can no doubt imagine, wildcards are really useful on the command line.

We could still put qualifiers into the regular expression (the middle part of the command that we're ignoring right now) that would make the `rename` command skip certain filenames, and we will; but it's important to understand what that third argument in the command is doing.

Now let's look at the middle argument: `'s/\.txt$/.csv/'` - the single-quotes around the regular expression tell the command line interpreter that it's all one statement.

The `s/x/y/` pattern is a switch statement, meaning that we want to swap one thing for another (take out x, put in y).

`.txt` is what we're looking for, and we specify that it needs to be at the end of the string (filename) with `$`. But `.` has a specific meaning ("any character") if it's not escaped, so we have to add the `\` in front of it.

`.csv` is refreshingly straightforward, right? It puts a literal ".csv" in where the .txt was removed from.

Look at what the command would do if you executed it by typing

```
> rename -n 's/\.txt$/.csv/' *.txt
```

The `-n` flag prevented it from actually running; it just showed you what it *would* do. So, now you should **type the same command without the flag (`-n`) to execute it**.

Look at your list of files (`ls`), and you should see that they all end in ".csv"

Other places this might come in handy:

- JPEGs mixed in with your JPGS (`rename 's/\.jpeg$/.jpg/' *.jpeg` -- though keep in mind that this will only swap out lowercase ".jpeg", just as `rename 's/\.JPEG$/.jpg/' *.jpeg` will only swap out uppercase)
- HTM files mixed in with your HTML files (`rename 's/\.htm$/.html/' *.htm`)

Change the various delimiters to underscores

Our volunteers' delimiters (the things that split "code4lib," the year, and what kind of thing is in the file) are all over the place. There are plus signs, dashes, spaces, and underscores.

I like underscores, so let's standardize on that.

This can be done with three commands (but wait before you enter them):

```
> rename 's/-/_/g' *
```

```
> rename 's/\+/_/g' *
```

```
> rename 's/\ /_/g' *
```

Let's break these down. This pattern will start to become very familiar. First off, notice we're matching on every file, this time (the last argument in each command is a wildcard, `*`).

The first command is a super straightforward switch statement (say that five times fast!): take out – and replace with _. The `g` on the end of the switch statement says to do it "globally," which means if we have the file "code4lib-2001-thing.csv", it will replace both dashes. If you leave off the `g`, it will only replace the first dash. (Try it! Remember to use the`-n` flag to see what the output would be, without actually running the command.)

The second command is very similar, but because + is a special character for regular expressions, we have to escape it. Again, we want to do the swap more than once if we have a filename with multiple plusses, so we include the `g` flag.

A slight digression: remember, **spaces in filenames are problematic**. We can work around them, but we have to remember to escape every space, or else use quotes (single or double quotes, doesn't matter). So, to view the contents of `that file.txt`, any of these commands would do:

```
less that\ file.txt
```

```
less 'that file.txt'
```

```
less "that file.txt"
```

Spaces have to be escaped both in normal command line usage *and* in regular expressions (*usually*, more on that below).

So, the third command, above: it makes sense, now, right? That's an escaped space in the first half of the switch statement, and it's replaced with an underscore, globally.

What's kind of cool, though, is that this whole character swapping deal can also be done with a single command:

```
> rename 's/[ +-]/_/g' *
```

The brackets are a game changer. They make what's inside them into what's known as a "character class," meaning that the regular expression will match one time, on any one thing inside them (*just* one time, unless you use a modifier like `*`). They also change the rules, because, notice: the space and plus sign aren't escaped. I know, believe me, I *know*. But just roll with it, believe me, try each example for yourself (trust but verify!), and see that it works.

Now, **run either the three separate commands or the combined command** (using `-n` to see the output before you commit to it).

Make everything lowercase

The volunteers all capitalized Code4Lib differently, just as they likely were split on how it's pronounced. Capitalization can affect how things are alphabetized in some systems, and it just looks messy; so let's make everything lowercase, shall we?

Note: We've reached the other snag for Mac users. This (apparently) won't work on a Mac, because Macs seem to have case-insensitive file naming. (I'm not bitter. This didn't waste *hours* of my time. Nope.) It works fine on Nitrous and many other systems with command line interfaces, so it's definitely worth your time to find out about. Just know that, on some systems, it'll fail with a message along the lines of "'Code4Lib_programs_2014.csv' not renamed: 'code4lib_programs_2014.csv' already exists."

Anyway, the command (wait to run it):

```
> rename 's/([A-Z])/\L$1/g' *
```

You see that we're calling `rename` on every file in the directory. You see the switch statement and the "global" flag (`g`).

Now, "A-Z" is maybe the most intuitive statement you'll see in a regex. It means just what you'd think, and when it's put into a character class like that, it means "any single character between A and Z." It won't match on a lowercase character; that would be `[a-z]`.

Now we end up in a bit of a bind. We can't replace every character we find with the same thing, like we've done above. (Every .txt became a .csv. Every space, plus sign, or dash became an underscore.) If we find a capital "C", we want to replace it with a lowercase "c", but then we don't want a capital "L" to *also* be replaced with a lowercase "c", right?

Notice that our character class is inside parentheses. That allows us to grab the thing that is matched (a bit like *this* in some programming languages, if that helps you), in something called a "group" (yes, in this case, it is one character, but it wouldn't have to be; it could be any number of characters). Now look at the second half of the switch statement; that $1 is a variable saying "the first group we grabbed."

And `\L` says "the lowercase of," so that `\L$1` is "the lowercase of that group we grabbed."

Run the command, and see the beauty of all-lowercase filenames!

Make the year appear in the same order in every filename

The volunteers seemed to split halfway on whether they named their files "code4lib-year-type.csv" or "code4lib-type-year.csv", but worry not; we can fix it with a simple(?) `rename` command.

This is the only example that relies on the others all having been run first; the others could have been done in any order. This one assumes everything is lowercase and all of the delimiters are underscores. So make sure you have run all the others first.

```
> rename 's/([a-z]*)_([a-z]*)_([0-9]*)/$1_$3_$2/' *
```

Nothing in this command is new, except for using multiple groups. It's just more complex-*looking*. But if you break it down, it makes sense, right? The underscores are literals, and the `*`s mean "match the preceding thing any number of times."

Grep

Grep is an incredibly useful tool for finding a particular word or phrase or pattern within a text file.

Let's say you have a copy of the Declaration of Human Rights handy on your machine (you *do*, actually:`human_rights.txt`), and you want to find the part where it talks about freedom of expression, without having to read the whole thing.

Go ahead and run this:

```
> grep -in "freedom of opinion and expression" human_rights.txt
```

The `-i` flag is *super* useful, because it makes the search case-insensitive.

The `-n` flag is *also* super useful, because it tells grep to output the line number.

The part in quotation marks is a regular expression (in quotes because of the spaces). You can use the same tricks you learned above while grepping files!

And then `human_rights.txt` is the file you're searching within. But you can also search multiple files at once.

```
> grep -in "grep" *
```

It shows you this section of these directions (we did mention you had a copy of these directions in this folder, right? because you do). Funny, right?

Find

Sometimes you forget where you put a file, or you need to look for files based on certain attributes (like when they were created or modified). That's when you use `find`.

Remember earlier today when we implied that all commands looked like

```
> command [-flags] [arguments]?
```

Wellll, that was mostly true. But some commands have flags that 1) are more than one character, and/or 2) have their own arguments. `find` is one of them. Here's its format:

```
> find (starting directory) (matching criteria and actions)
```

So if you want to make a list of files named, for instance, "instructions.md" in `/home/nitrous/cli-workshop/`, here's what that looks like:

```
> find /home/nitrous/cli-workshop/ -name instructions.md -print
```

(It should show you the names and full paths of three files.)

`/home/nitrous/cli-workshop` is where it starts looking, and it looks for files with the name "instructions.md," in all of the subdirectories under that directory. (There is a flag to prevent it from acting recursively, though.) That final flag, `-print`, tells it to show a list of those files and their paths.

You can do some really powerful stuff with `find`, including searching for files that were accessed on a certain date (not a super useful example to run, right now, on this machine) or files that were modified on a certain date or files that are a certain size.

Here's a command to look at everything in our workshop directory that was modified today:

```
> find /home/nitrous/cli-workshop/ -mtime -1 -print
```

But you saw that `-print` flag? That implies that there are other things you can do, besides printing the list, right? Right.

`-mtime` is "the time when the file was modified," and `-1` is, confusingly, not actually a flag; it means "less than one day ago"; its inverse is `+1`, meaning "more than a day ago." How do you think you'd get the files modified yesterday? (Try it!)

You can execute commands on all of the files you `find`, which is a lot of power to have in one command... It also explains why I said you could replace `rename`, above, with a cleverly-run `find` command. (That's an exercise for any of you who want to try it.) There's more information on `find` here.

Nano

Sometimes you need to edit text files, even though you're busy doing stuff on the command line. Not a problem.

There are a whole bunch of text editors available on the command line, and there are strong feelings about which one's best. They're all good, honestly; just, some require a bit more time commitment to learn.

The one that takes the least time to learn (in my opinion) and is also available on the largest number of systems is `nano`. It's super friendly and even gives you hints at its commands on the bottom of the screen!

Create a new text file named mytext.txt:

```
> nano mytext.txt
```

Type a little bit, whatever you want, it doesn't matter.

The "^" in front of each letter (in the menu options at the bottom of the screen) refers to the `ctrl` key. So **`ctrl-x`** will let you exit the text editor, at which point it asks if you want to save. Choose "Y" to save your file, and if you hit "Enter" it will save as "mytext.txt." (You can also change the filename at this point, if you prefer.) Once you've saved, feel free to list the files in the directory (`ls`) to convince yourself you made a text file.

Remember how `grep -n` was cool because it gave you a line number (99, in the case of what we searched)? Here's why we like that:

```
> nano +99 human_rights.txt
```

And now you're right there by the line you searched for, and you can make relevant edits.

If it's your first time in a text editor on the command line, please take a little time to play. It can be super disorienting at first, but it's useful to develop a little comfort with CLI text editors! (And don't worry about messing anything up; the Universal Declaration of Human Rights is still out there on the internet, no matter what you do to this copy.)

We don't need to go through all of the commands right now, but it's worth pointing out that `ctrl-k` cuts an entire line, and`ctrl-u` puts it back in **wherever your cursor is**. So that's how you cut and paste in a text file on the command line.

Also, if you want to save your work without exiting the file, you want to use `ctrl-o`.

Wrap up

So that's the exercise for archivists. Hopefully you've asked your questions as they've come up, but if not, please feel free to call over one of the workshop leaders or anyone else who seems to know what's going on.

If you're finished before the workshop time is over (or you're excited and want to keep going on this after the workshop's over), feel free to work on the cataloger or web developer exercise, too.

Thank you!