Cataloging on the Command Line

Catalogers can profit from many uses of the command line, primarily through batch processing metadata files to make many small changes or convert from one format to another. In this exercise, we'll learn how to run scripts on the command line, pass information to those scripts (such as what file we want them to process), and "redirect" output to a file. Ready? Let's go!

Setup

We'll start with the worst part: sometimes we need to install software on the command line to Get Things Done. And sometimes installation can be quite annoying, involving errors, changing permissions, and editing our shell's configuration files. Hopefully this installation will go smoothly for you, but if not please ask your fellow attendees or the workshop instructors for help.

What are we doing, precisely? We want to use the provided "MARCgrep.pl" script, an excellent script written in the Perl programming language. Only to do so, we need the "MARC::Batch" module which the script utilizes. MARC::Batch provides the script with handy MARC processing tools so that it doesn't have to be a million lines long and filled with complex MARC parsing code. To install it, we use `cpan` which a sort of package manager for Perl that allows us to install modules, bits of reusable code. Run the following command:

```
> cpan MARC::Batch
```

An inordinate number of prompts will occur as `cpan` asks you for your entire personal history. You should be able to simply press Return through each question. Then, when you're finished, the script *still won't work*! Arrgh! We need to restart our shell so it recognizes the changes that `cpan` has made and knows where to find the newly added module. We can do so by running:

```
> exec -l zsh
```

Next, we need to install a python module named pymarc which is used in some of the later scripts. Pymarc's actually quite similar to the Perl module we just installed; it provides programs with the means of parsing and editing MARC records. We install it using `pip`, python's package manager, but we also precede the command with `sudo`. `sudo` stands for "superuser do" and it's a way of running a command with elevated privileges. Often, when installing software or editing your system's configuration, you'll need to prefix a command with `sudo`. So let's run

```
> sudo pip install pymarc
```

to get started. You can confirm that this worked by running the `python` command to enter the python REPL, then writing`import pymarc` at the prompt. If no errors or messages appear, then you've successfully installed pymarc. You can exit the python REPL with the `exit()` function.

Whew, hopefully all is well. We should be seeing a typical command line prompt now. Let's get to scripting.

What Exactly Is a Script?

"Script" is probably a term you've heard, whether you've run one or not. A script simply collects some lines of code or commands in one text file and executes them. Actually, all the commands we've typed thus far could be put in a file and run in sequence. This is one of the tremendous powers of the command line; anything we can do, no matter how lengthy or complex, can be written into a single script and executed over and over again without much trouble.

For instance, take these three commands:

```
> echo 'hello!' >> example.txt
> echo "example.txt is $(wc -l example.txt | sed -e 's/[a-zA-Z. ]//g') lines long
> echo 'and the last line reads:'
> tail -n1 example.txt
```

Rather than type them all out time and time again, we can write them into a script where their repetition is trivial. There are some other subtleties to scripts (like telling the operating system what program to use to run the script) but let's move on for now.

Running a Script

Running a script is similar to using any command or executing a program, you simply type the full path to the script file and press Enter. But this becomes incredibly tedious; we don't want to have to type out a long path to our current directory every time we run a script. Instead, we use the period (".") shorthand which refers to our current location. So we can run the "script.sh" in this folder like so:

```
> ./script.sh
```

Hey hey, we did it! Since "." refers to the current directory, the above is equivalent to "/path/to/where/you/downloaded/cataloging-exercise/script.sh".

That's most of what you need to know about running scripts, but there are some nuances. The "non-exe-script.sh" has the exact same text as "script.sh", but what happens when we try to run it? We get some kind of "permission denied" error. Why is that? Well, let's check permissions:

```
> ls -l
```

We're using the familiar `ls` command but have added an option to it that make its output more verbose. There's quite a bit of information here, but the most important part is at the beginning of each line where the permissions string "-rw-r--r--" precedes "non-exe-script.sh" while "script.sh" has the different permissions "-rwxr-xr-x". We won't go into exactly what these mean, but the "x" in that string refers to "eXecutable" and our "non-exe-script" isn't allowed to be run by the shell. We can change its permissions with the `chmod` (CHange MODe) command:

```
> chmod +x non-exe-script.sh
```

Now what happens when you try to run "non-exe.script.sh"? Does it work? What do you think you'd have to do to make it*not* executable any longer?

By the way, if you want to *stop* a script you can use Ctrl + C to do so. This sends a "keyboard interrupt" and will stop whatever your shell is currently doing. It can be *very* handy when you realize you've accidentally run a script with the wrong parameters. Parsing our example MARC file will take a long time so remember Ctrl + C is vital.

Passing Information to a Script & "Flags"

OK, we know to run a script we need to a) refer to its location, and b) make sure it's executable. So far, so good. But what if we want to pass information to a script? For instance, what if a script could run on different files or operate differently depending on the options we specify, much like how `ls -l` is different from plain ol' `ls`?

One common way to pass information on the command line is by "flags". We've already seen several examples, because these flags are merely the hyphenated letters we've used previously. But let's dive into a cataloging example using a wonderful script in this folder: MARCgrep.pl. MARCgrep is a tool for searching over MARC files and counting up how many records match a provided pattern, or have a certain field or subfield. It's syntax is like so:

- The "-e" flag is followed by a string like "{field},{indicator1},{indicator2},{subfield},{value}" such as "245,,,,fox" (matches any record with "fox" in its 245 title field)
- The "-c" flag, if present, means that the script only counts how many records match the pattern, otherwise the script spits out the full record
- The "-f" flag can be followed by a comma-separated list of fields to print instead of the full record, e.g. we use "245,100" to print just the title and author fields.

There's a full example below; what do you expect it will produce? Give it a try!

```
> ./MARCgrep.pl -e '245,,,,fox' -f '245' example.mrc
```

Note that we've wrapped the information we're passing to the script in quotation marks. This isn't strictly necessary, but is a good habit to get into because (as discussed earlier) spaces can cause trouble on the command line. Just to practice passing different information to the script, try to answer the following questions using the provided "example.mrc" set of records:

- How many records have "turkey" in their 245 title field?
- What's the title of the record where the author's (100 field) last name is Rodriguez?
- How many books have 651 fields?
- Is Shakespeare more commonly the author (100) or the subject (600) in these records?
- What are the other subjects of the book where Gentrification - California - San Francisco is a subject (650)?

Try a few more queries on your own, with different combinations of "-e", "-f", and "-c" flags.

There's also a second way to pass information, by *positional* parameters. Remember when we ran `cmod +x non-exe-script.sh` up above? How did the script know that our first parameter "+x" was an option and our second one was a file? Commands might expect certain types of information in certain positions, in the case of the `chmod` command the syntax is always: chmod, followed by a (required) option defining the permissions to change, followed by a (required) file. Try reversing the two arguments:

```
> chmod non-exe-script.sh +x
```

What happened? Did your computer explode? Hopefully not. Most likely, the chmod command complained that "non-exe-script.sh" was an "invalid mode" because it was expecting the *mode* (translation: permissions) first and the file second, not the other way around. We just have to remember the order of positional arguments. For that reason, they're not as convenient or explicit as flags, which tend to make it obvious what's going on and don't care about order at all.

Output Redirection

Typically, when a script runs, it prints its output to "standard out", i.e. right in our terminal. But output can be redirected into a file, or become the input of another script or program. Below, we'll learn how to send the output of MARCgrep to a file instead of our screen. This can be immensely helpful in a lot of situations; when we print out thousands of subject headings, we probably want to put them in a file to review or analyze later, not to read them by scrolling a thousand lines in our terminal.

Writing to/over a file

We can write the output of a command to a file using the greater than sign and providing the name of the file. This command will write a short greeting to a text file:

```
> echo 'Hello, by jolly, hello!' > greetings.txt
```

What happens if we write a different salutation to the file?

```
> echo 'Good tidings, my friend!' > greetings.txt
```

Our new text overwrote the previous entry entirely. So the output redirect operator > has the following properties:

- if the file doesn't exist, it will be created
- if the file *does* exist, its content will be erased
- the file will be filled with the text output of the command preceding it

Knowing this, try printing a list of your favorite MARC field to a text file using MARCgrep and the sample file provided here. Then overwrite the list with *your least favorite* MARC field (it's the leader, isn't it? I knew it). Notice how, while we're running the command, we don't see any text printed to our terminal because it's all being *redirected* into the file.

Appending to a file

We don't have to write over the contents of a file, erasing our favorite MARC field with our least favorite. Instead, using a slightly different operator—two greater than signs in a row ">>"—will *append* our output to the end of a file.

```
> echo 'this is the first line' >> lines.txt
> echo 'this is the second line' >> lines.txt
> cat lines.txt
this is the first line
this is the second line
```

Let's try using MARCgrep to make a list of all 650 fields with Library in them *and* all 710 fields with "Library" in them. This involves running the script twice and redirecting the output to the same file, but *appending* to that file rather than overwriting it.

Now, let's write all 650 fields with "Archive" in them to the same file, overwriting the list of fields we compiled previously. We can use the `cat` command to print out the file's text afterwards, checking if we did it right. The `head` command is also useful here; it prints out just the beginning of a file, not its full contents.

What if we want to *prepend* our output to the beginning of a file? Can you think of a way, involving the `cat` command, to do that? Spoiler: it will not be straightforward.

"Piping" Output Through Multiple Commands

Perhaps the most powerful feature of the command line is that you can not only redirect a command's output to a file, you can pass the output of one command as the input to a following command. This seemingly simple feature allows us to create complex yet elegant operations by chaining together many smaller, simpler actions. We do so by putting a vertical bar "|" between two commands, which causes the *output* of the former to be used as the *input* of the latter. What is then printed out (or redirect to a file!) is the result of the last command.

Earlier, we used MARCgrep to print out MARC fields matching a given pattern. You may have noticed that the output included blank lines, which cluttered it and made it unnecessarily lengthy. What if we pipe the output of MARCgrep through another command to strip out blank lines? Here's an example utilizing our earlier search for the word "fox" in title fields:

```
> ./MARCgrep.pl -e '245,,,,fox' -f '245' example.mrc | sed -e '/$^/d'
```

This command is more arcane than anything we've seen previously; what's with all the silly punctuation at the end?!? It translates to "delete every line with no characters on it", since the `$^` bit is a Regular Expression which matches *empty*strings. Regular Expressions are complex, powerful tools...and well beyond the scope of this workshop. But if you already know them, the syntax of the handy `sed` command isn't tricky. `sed` stands "Stream EDitor" and it excels at manipulating text being piped around. You can even string together numerous `sed` commands like so:

```
> echo 'MARC is great!' | sed -e 's/is/must/' | sed -e 's/great/die/'
```

Try the above command on your system; what output does it produce? By now, we have figured out that the `sed` syntax is basically `sed -e {EDIT}` where {EDIT} is some type of pattern-based edit to perform on each line of the input. We've already seen two types of `sed` edits:

- `sed -e '/{PATTERN}/d'` *deletes* all lines matching the {PATTERN}
- `sed -e 's/{PATTERN}/{SUBSTITUTE}/'` *replaces* the first mactched instance of {PATTERN} with its {SUBSTITUTE} on each line

We can actually perform several edits with one call to `sed` by stringing together a series of `-e {EDIT}` flags, so the example above could be written more concisely as `sed -e 's/is/must/' -e 's/great/die/'` but I wanted to demonstrate that an arbitrary number of pipes can be used in one command. Knowing the `sed` syntax, let's try a few exercises:

- Print out all the 650 subject fields with "Pennsylvania" but delete 650s with the word "history" in them
- Do the above but also filter out blank lines
- Do the above but drive Pittsburgh natives mad by replacing "Pennsylvania" with "Philadelphia"
- Finally, try writing the results of these operations to a file

See how we can string together a number of simple operations to do something rather complex? The last exercise really drives home how the "stream" of text is simply being diverted; rather than printing it to our screen right after the first command, we sent it through a few pipes and finally into a file.

Knowing the find/replace and delete uses of `sed` provides a tremendous amount of utility on the command line. Let's learn one more command just to get further piping practice. Say we want to *count* the number of subject headings that match a pattern (ignoring that MARCgrep has this feature built in...). The `wc` command stands for "word count" and it can be used as a rudimentary analysis tool:

```
> MARCgrep.pl -e '245,,,,library' -f '245' | wc
```

This prints out three quantities: first the number of *lines* in our output, then the number of *words*, and lastly the number of*bytes*. If we're only interested in one of those figures, `wc` has flags that allow us to retrieve only the desired one. Use either the manual page (`man wc`) or the `--help` flag to figure out what the flags are, then complete these two exercises:

- How many titles in the example MARC file contain the words "library" or "archive"? (Remember that you might need to filter out blank lines to obtain an accurate total)
- How many words are in each of the title fields containing the word "fox"? (Don't cheat by counting yourself! This is probably easiest if you `echo` the titles one-by-one into `wc`)

Exercise One: Batch process a MRC file

Included in this folder is a Python script named "pm-script.py". It uses the incredibly popular "pymarc" Python module to batch process MARC records. The script's syntax is:

```
> ./pm-script.py --limit 100 input.mrc output.mrc
```

where the number after the optional "limit" flag is the maximum number of MARC records to process, the first first passed in the input, and the second file passed is where the output will be written to. The command also accepts an optional "--verbose" flag which causes it to print information about what it's doing. Try running the script a few times, at first with a small limit but then process the full file. What is the script doing? Can we write its output to the same file multiple times and if so what happens to the prior output? Can we run it on its own output and if so what does that accomplish?

You may be thinking "but my ILS client can already do this"! And that's true in a number of cases, the operations possible on the command line may be achievable elsewhere. Still, there's often greater flexibility in command line scripts, which can utilize complex logic that your ILS may not be able to, and these scripts can be combined to perform several operations in a row quickly. Try running the "batch-analysis.sh" script; what does it do? Remember you can run `cat batch-analysis.sh`to read the text of the script, or open it up in a text editor.

Bonus Problem: the "batch-analysis.sh" script writes out a plain text file. Try mimicking the script's use of the stream editor`sed` to delete URLs beginning with a certain pattern (put the pattern after the "`^`" in the original `sed` command and delete the dollar sign).

Exercise Two: Use "help" to figure it out

We learned earlier to pass information to a script using a `--limit` flag and positional parameters. That's all well and good, but what if we're given a script or program and we're not sure what flags it accepts or where the positional parameters go? We can often search the web for answers, and that's a fine approach, but one of the nicest things about the command line is how documentation is often right at our fingertips.

In this folder there's a "exercise-two.py" script. Find out what it does and run it successfully on our sample MRC file. Unsure where to start? There's a common convention whereby the flag –`-help` or its shorthand `-h` will provide information. Try getting help with and then using the script. Run the script multiple times with different parameters to make sure you understand its usage.

Let's return to MARCgrep.pl too; what if we want to find out how many records *do not* have a certain field? Let's use MARCgrep's help to find out!

Notes

- MARCgrep http://en.pusc.it/bib/MARCgrep
- pymarc https://github.com/edsu/pymarc
- Internet Archive Open Library Data https://archive.org/details/ol_data (source of example MARC)
- `sed` tutorail http://www.grymoire.com/Unix/Sed.html