the command line

4 November 2024

What is the command line?

It's a text interface to your computer. It can do almost anything that you can do with the more common graphical programs: open files, edit files, move, rename, delete files. You can also do things like surf the web or read email, if you really want to.

It's also called

Useful to know in doing web searches, and in interpreting the answers.

- *Terminal* (particularly on the Mac)
- *Shell*
- *CLI* (command line interface)
- *Bash* (actually a particular program, but often a synecdoche)

Why should we care?

The command line enables you to do common many computational tasks without writing any code. In this workshop we will do some of the things we did in Python in previous weeks.

The command line is particularly suitable for working with very big files, because the whole file is not read into memory at once but line by line, or to working with lots of files.

Often programs *can* be run from the command line, such as `.py` Python files. Sometimes programs can *only* be run from the command line, such as Pandoc.

where am I? The simplest form is just to type a command and press enter:

```
pwd
```

This stands for *print working directory*. CLI commands tend to be very short and quick to type.

You can run CLI commands from Jupyter Notebook by prepending `!`.

You can also open a CLI window from the Jupyter Notebook.

But the commands will need to match the default terminal you're running. For example:

If you're on a Mac you can run this to list the files on the same directory as the Notebook:

```
[ ]: !ls
```

If you're on Windows you can run this to the list the files in the same folder as the Notebook:

```
[ ]: !dir
```

count words Very often you need or want to run the command *on* something. For example, to the count the words in a file you will need to specify the file. This is called an *argument*:

```
wc persuasion.txt
```

(If you're not in the same folder as the file you want to run `wc` on, you will need to specify the path to the file).

You should get three numbers back: the number of lines, the number of words and the number of characters. This is the default with `wc`.

But we can choose to display only some of the figures with a *flag*. A flag modifies the default behaviour of a command. With `wc` we can return only the number of words in *Persuasion* with the `-w` flag:

```
wc -w persuasion.txt
```

What about if we want to get the word counts of all of Jane Austen's novels? Instead of specifying a specific file as the argument we can specify a pattern to match. This is called *globbing*. The Austen files in our directory all end in `.txt` and no other files do, so this pattern is easy: anything ending in `.txt`

```
wc -w *.txt
```

A key feature of the command line is that it scales very well. We're getting the word counts for six novels here, but if we had 10,000 or 1000,000 novels in a directory, we can use exactly the same command. As long as they all end in `.txt`, or some other pattern that we can specify.

The good news is that many command line programs follow exactly the same form as `wc -w filename.txt`.

find strings Now let's look for strings in files using the tool `grep` (it stands for *global regular expression print*).

The basic usage of grep is:

```
grep "string-you-want-to-find" filename
```

So to look for the word *Anne* in `persuasion.txt`:

```
grep "Anne" persuasion.txt
```

To get a count of the *number of lines* that contain the string, `grep` has a `-c` flag:

```
grep -c "Anne" persuasion.txt
```

Note that flags vary from one CLI program to another, although they *sometimes* mean the same thing.

You can combine flags and the order usually doesn't matter. To count case-insensitive searches with `grep` we can use the `-c` flag and the `-i` flag together:

```
grep -ci "marriage" persuasion.txt
```

We can apply the same `grep` commands and, instead of searching just one text, we can search all the files in the directory with a `.txt` termination:

```
grep -ci "marriage" *.txt
```

Again, this is counting six Jane Austen novels but if there were 1,000 novels in the same directory we'd type exactly the same thing.

Of course we can use regular expressions with `grep` (it's in the name, after all). To do that we need the `-E` flag. Again, we would be returning lines that contain the specified regex but normally we'd like to see just the results of the regex matches. So we combine three flags:

- `-i`, case insensitive
- `-E`, use regex
- `-o`, only return the part of the line that matches

```
grep -Eio "\w+ marriage" *.txt
```

pipes

Individual command line programs are quite powerful, but using them by themselves is only a fraction of their power. The full utility of the command line lies in the way it is possible to chain commands together, sending the output from one simple command to the input of a different simple command, building up to the exact output required.

We've got the results for the word before marriage but we might like to know how many there are across all of Austen's novels. We can count with `grep` but it gives results per novel. OK in the case of six novels but inconvenient if we are working with hundreds or thousands.

Happily, we can combine CLI commands with other using the `|` character, known as pipe. This passes the output of the command to the left of the pipe to the input of the command to its right. So we can combine the two commands we've already seen, `wc` and `grep`:

```
grep -Eio "marriage[sd]?" *.txt | wc -l
```

Another example: when we counted the words in Jane Austen's novels the results were ordered alphabetically by novel. `wc` doesn't have a sort argument because there is a separate `sort` command that we can pipe to:

```
wc -w *.txt | sort
```

`sort` has arguments of its own, like `-r` for reverse:

```
wc -w *.txt | sort -r
```

Examining complex files We have two big XML files and two big HTML files of Adam Smith's *The Wealth of Nations*.

How can we find out what's in them?

We know that they are marked up with angle brackets so we can use `grep` and `regex` to find all the results, count and order them.

(We could also write a Python script to do this but this way is quicker).

The XML files are the only files in the directory with an XML termination so we don't need to specify the names:

```
grep -Eo "<[^>]+>" *.xml
```

But we don't want the filename; the flag for this is `-h`:

```
grep -Eoh "<[^>]+>" *.xml
```

Nor do we want the closing tag so we adjust the regex:

```
grep -Eo "<[^>/]+>" *.xml
```

```
grep -Eo "<[^>/]+>" *.xml | sort | uniq -c | sort -nr | head -n 20
```

We can just change `*.xml` to `*.html` to compare the HTML version:

```
grep -Eoh "<[^>/]+>" *.html | sort | uniq -c | sort -nr | head -n 20
```

Saving your results All these results aren't very useful if they just appear on screen. To write to a file instead, append `> filename` at the end of the commands (be aware that if a file of that name already exists it will be overwritten without warning you).

```
grep -Eio "\w+ marriage" *.txt | sed 's/ /,/' | sed 's/:/,/' > marriage.csv
```

Group work

- Search for the string *capital* in the XML version of *The Wealth of Nations*
- How many lines contain *capital*?
- Returning the whole line makes it hard to see the results. Search for *capital* but only return 10 characters around it (you can use the method we used when we did this in regex in week 2.
- Write the results of the above search to a file called `capital.txt`
- How can you find lines in *The Wealth of Nations* that contain *capital* and *gold*? How can you count how many lines contain both words? (you'll need to use pipes)
- We didn't cover this above, but the flag for lines which **do not** match is `-v`. Use this to find and count the lines that contain *capital* but which do not contain *gold*.
- Redo the marriage search in Jane Austen that we did above, but remove the file termination when writing the results to CSV.
- Delete all of the XML elements in *The Wealth of Nations* and write the result to a new file called `wealth-of-nations.txt`