# NYU Library Carpentry: Introduction to Working with Data

Vicky Steeves

November 5, 2020

# Contents

# Welcome

Welcome! This book is an adapation of the Library Carpentry lesson Introduction to Working with Data. It was made by Vicky Steeves for a Library Carpentry workshop at NYU.

## Setup

This lesson does not require any special setup!

# Jargon busting

Before we start any technical work, let's start by establishing some terms as a group. The first barrier to getting involved with computational work is often the language barrier – there are unknown terms and intimidating commands that can be hard to overcome.

So we are going to start with an exercise to get us thinking about data and code from a terminology level! Let's address head-on the terms that give us anxiety and the terms that we are interested in knowing more about, and bust that jargon before the rest of the sessions.

So for this first 30-40min of the day:

1. We will put you into breakout rooms of 4-6.
2. Brainstorm for 5 minutes about any terms, phrases, or ideas around data or code in libraries that you've come across and wish to know better. Just rapid-fire list out terms. Some examples might be, "the terminal," "Git," or "regular expressions."
3. Add your list of all the terms, phrases, and ideas that your group came up with to our class jamboard: https://jamboard.google.com/d/1-F6PKusN8esa9zoPBM4RqAfzsGxk0ADVRLc8QFKP7gg/edit?usp=sharing. If multiple people in your group said the same thing, note that!
   - Use the "board" with the same number as your breakout group, so breakout room 2 goes to board 2/5, breakout room 3 goes to board 3/5, using the arrows at the top of the jamboard.
4. After your brainstorming session, spend 10 minutes working together to try to explain what the terms, phrases, or ideas on your list mean. Write your potential definitions with your terms on your jamboard. Note: use both each other and the Internet as a resource.
5. Then we'll come back together as a group and have folks report out the terms you were able to explain as well as those you are still struggling with
6. For those that folks weren't able to explain, we'll add those to our mega class list and define them in-class together!

# A computational approach

To start this lesson, let's first ask: why take an automated or computational approach?

This is otherwise known as the 'why not do it manually?' question. There are plenty of times when manual work is the easiest, fastest, and most efficient approach. Here are two conditions that should make you consider using automation:

1. You know how to automate the task.
2. You think this is a task you will do over and over again.

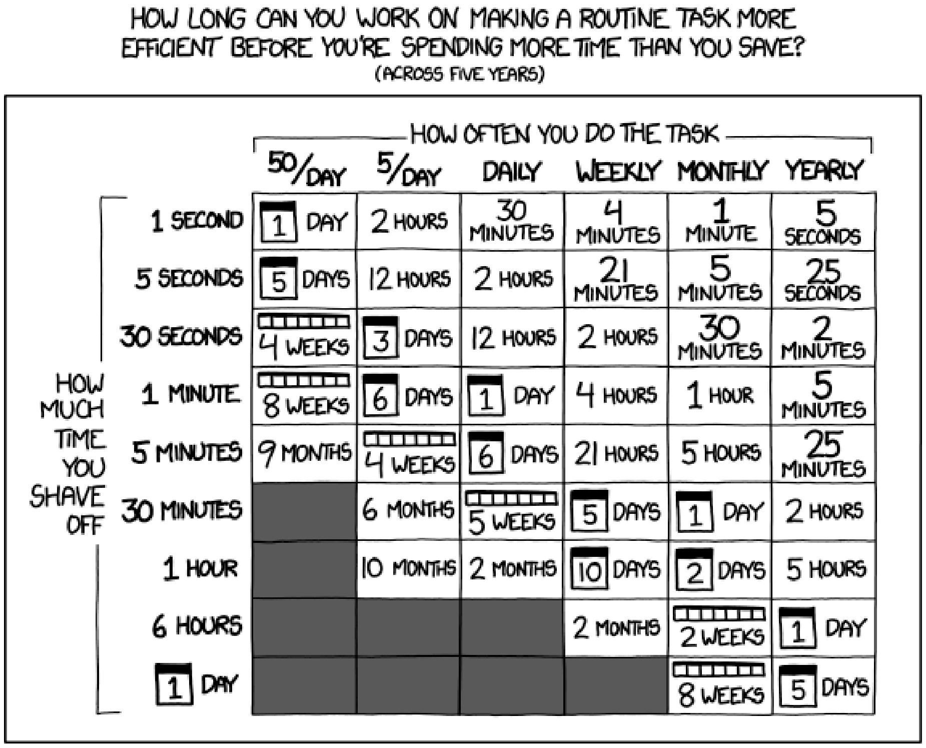
Automation refers to a process or procedure that runs with little to no supervision or action. Libraries have been automating workflows for years and many the Library Carpentry lessons can help people in libraries consider and implement automation approaches to further those efforts. You may often receive reports that you have to manually format or bibliographies where you have to clean metadata.

Some motivations and takeaways:

- *Automate to make the time to do something else*! Taking the time to gather together even the most simple programming skills can save time to do more interesting stuff! (even if often that more interesting stuff is learning more programming skills…)
- Consider the role of programming in professional development. *Computational skills improve your efficiency and effectiveness.* Stay alert to skills you want to learn, and be aware of what skills you can make sure your staff learn, as well.
- *Knowing (even a little) code helps you* evaluate projects that use code. Programming can seem unknowable. Getting to run some code can empower you to dive in, start small, and ramp up!

**Discussion break**

Is there something you would like to automate in your work?



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

| HOW MUCH TIME YOU SHAVE OFF | HOW OFTEN YOU DO THE TASK | | | | | |
|---|---|---|---|---|---|---|
| | 50/DAY | 5/DAY | DAILY | WEEKLY | MONTHLY | YEARLY |
| 1 SECOND | 1 DAY | 2 HOURS | 30 MINUTES | 4 MINUTES | 1 MINUTE | 5 SECONDS |
| 5 SECONDS | 5 DAYS | 12 HOURS | 2 HOURS | 21 MINUTES | 5 MINUTES | 25 SECONDS |
| 30 SECONDS | 4 WEEKS | 3 DAYS | 12 HOURS | 2 HOURS | 30 MINUTES | 2 MINUTES |
| 1 MINUTE | 8 WEEKS | 6 DAYS | 1 DAY | 4 HOURS | 1 HOUR | 5 MINUTES |
| 5 MINUTES | 9 MONTHS | 4 WEEKS | 6 DAYS | 21 HOURS | 5 HOURS | 25 MINUTES |
| 30 MINUTES | ▨ | 6 MONTHS | 5 WEEKS | 5 DAYS | 1 DAY | 2 HOURS |
| 1 HOUR | ▨ | 10 MONTHS | 2 MONTHS | 10 DAYS | 2 DAYS | 5 HOURS |
| 6 HOURS | ▨ | ▨ | ▨ | 2 MONTHS | 2 WEEKS | 1 DAY |
| 1 DAY | ▨ | ▨ | ▨ | ▨ | 8 WEEKS | 5 DAYS |

Figure 1: ‘Is it worth the time?’ by Randall Munroe available at https://xkcd.com/1205/ under a Creative Commons Attribution-NonCommercial 2.5 License.

## Names Matter

So let's set ourselves up well to be able to automate our tasks. We are all guilty of naming our files in such a way that sometimes we have a hard time finding them, and that means that it'll be harder for us to work with them in an automated fashion. The following XKCD comic may be all too familiar:

There are a number of other less than ideal file naming examples cataloged by Twenty Pixels. Do any of these examples look familiar?

Some best practices include:

- Prefix your files with the date created using a `YYYY-MM-DD` format
- Avoid special characters like `&`, `%`, `$`, `#`, `@`, and `*`. Just use letters and numbers.
- Do not make file identity dependent on capitalization unless implementing camel case (e.g. `fileName.xml`).
- Never use spaces in file names – many systems and software will not recognize them or will give errors unless such filenames are treated specially. Use an underscore `_` or a dash `-` instead of a space.
- Use short file names. For your sake and the sake of systems that'll fail if you give it like a 50 character file name.

### Naming files sensible things is good for you and for your computers

Working with data is made easier by structuring your files in a consistent and predictable manner. Without structured information, our lives would be much poorer. As library and archive people, we know this. But let's linger on this a little longer because for working with data it is especially important.

Examining URLs is a good way of thinking about why structuring data in a consistent and predictable manner might be useful in your work. Good URLs represent with clarity the content of the page they identify, either by containing semantic elements or by using a single data element found across a set or majority of pages.

A typical example of the former are the URLs used by news websites or blogging services. WordPress URLs follow the format:

`ROOT/YYYY/MM/DD/words-of-title-separated-by-hyphens`

https://cradledincaricature.com/2015/07/24/code-control-and-making-the-argument-in-the-humanities/

A similar style is used by news agencies such as a The Guardian newspaper:

`ROOT/SUB_ROOT/YYYY/MMM/DD/words-describing-content-separated-by-hyphens`

https://www.theguardian.com/uk-news/2014/feb/20/rebekah-brooks-rupert-murdoch-phone-hacking-trial

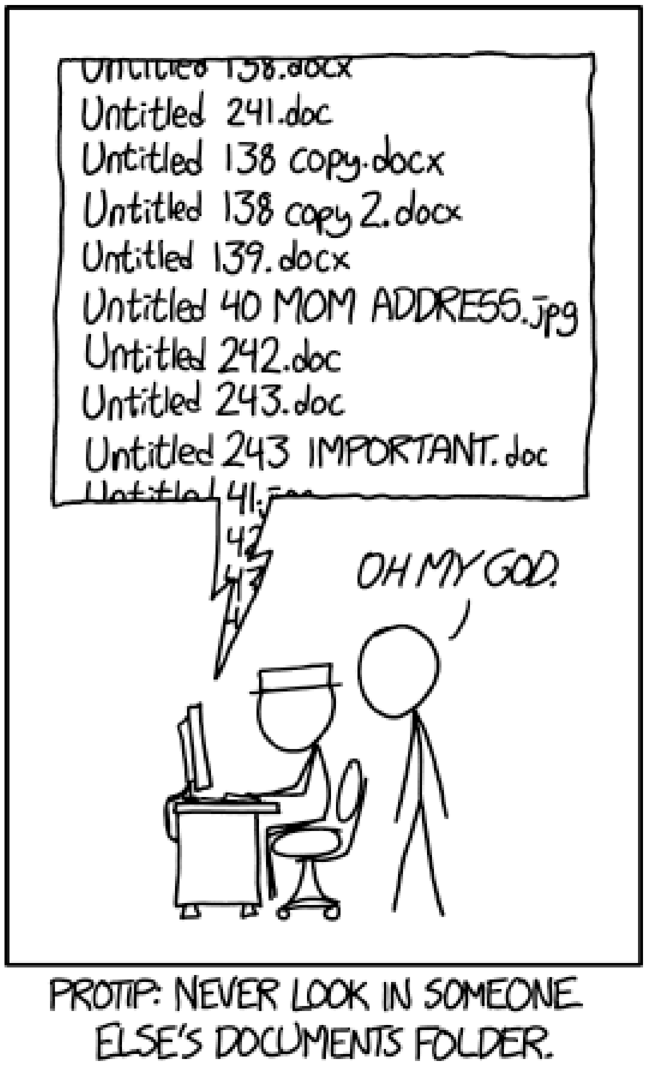


Figure 2: 'Protip: Never look in someone else's documents folder.' by Randall Munroe available at https://xkcd.com/1459/ under a Creative Commons Attribution-NonCommercial 2.5 License.

In data repositories, URLs structured by a single data element are often used. The National Library of Australia's TROVE uses this format:

`ROOT/record-type/REF`

https://trove.nla.gov.au/work/6315568

The Old Bailey Online uses the format:

`ROOT/browse.jsp?ref=REF`

https://www.oldbaileyonline.org/browse.jsp?ref=OA16780417

What we learn from these examples is that a combination of semantic description and data elements make for consistent and predictable naming structures that are readable both by humans and machines. Transferring this kind of pattern to your own files makes it easier to browse, to search, and to query using both the standard tools provided by operating systems and by the more advanced tools that we'll cover in these next days together. When we know what things look like, we can match on patterns and be able to automate quite a lot. We'll look at pattern matching next!

In practice, it's also useful then to keep a standard way of organizing your projects alongside your file naming conventions, to help us avoid falling down the nested folder rabbit hole. This way of organizing projects I've found is one of the most helpful:

- Put each `project` in its own directory, which is named after the project and perhaps prepended with that `YYYY-MM-DD` of when the project started.
  - if this is a more general file structure, I tend to have a `CurrentWork` directory and a `Documents` directory to separate out the files I am keeping because I'll refer to them again and the files I'm working on the most.
- A series of sub-directories such as `docs`, `data`, `metadata`, `events`, etc. depending on what you are working on.
- Within those directories, keep your naming convention!

The name of a file is important to ensuring it and its contents are easy to identify. `Data.csv` doesn't fulfill this purpose. A title that describes the data does. And adding dating convention to the file name, associating derived data with base data through file names, and using directory structures to aid comprehension strengthens those connections.

## Picking open formats

Otherwise known as, **plain text formats are your friend**. Why? Because computers can process them!

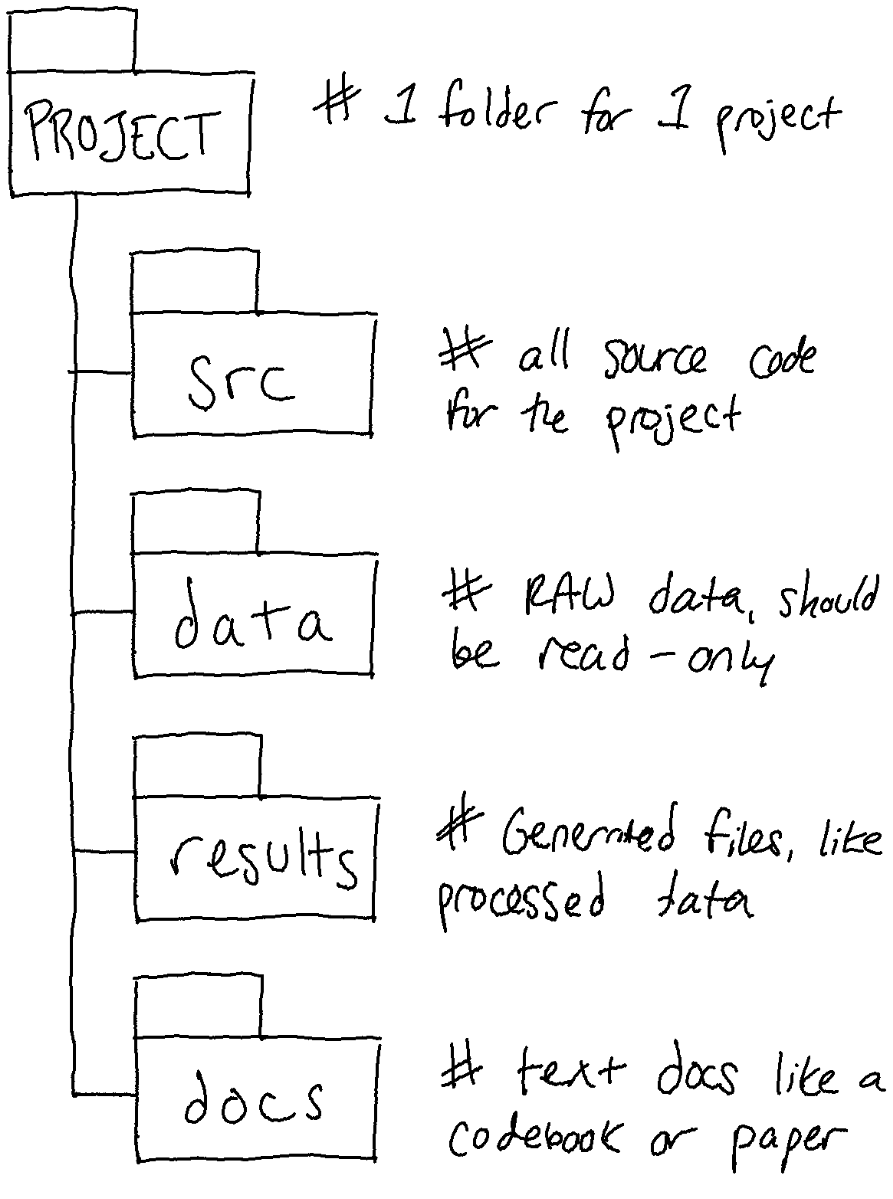

PROJECT
# 1 folder for 1 project
src
# all source code for the project
data
# RAW data, should be read-only
results
# Generated files, like processed data
docs
# text docs like a codebook or paper

Figure 3: Here's my own personal file structure I use for my research projects!

If you want computers to be able to process your stuff automatically, try to get in the habit where possible of using software-agnostic formats such as `.csv` (comma-separated values) or `.tsv` (tab-separated values) for tabular data/spreadsheets.

These plain text formats are preferable to the proprietary formats (e.g., Microsoft Word or Excel) because they can be opened by many software packages – so they don't make you rely on/have to pay for software to view your work! This is much more sustainable nad increases the chances of keeping the files viewable and editable in the future. Most standard office suites include the option to save files in `.txt`, `.csv`, and `.tsv` formats, meaning you can continue to work with familiar software and save your files in the more perennial formats. Compared to `.docx` or `.xlsx`, these formats have the additional benefit of containing only **machine-readable elements**.

For instance, when we license data for text mining, we collect it in plain formats like `.txt` or `.xml` so that researchers who come to us to use this data can use it with their analysis scripts which might do things like automate looking for entities or keywords in large amounts of text. In Library Carpentry: The UNIX Shell lesson that we'll do this afternoon with Andrew, we all see how the command line can be a powerful tool for working with text files.

Word processors like Microsoft Word, LibreOffice Writer, and Google Docs will explicitly not work for this purpose – those tools are meant to optimize how documents appear to humans, not to computers. They add many hidden characters and generally are unsuitable for working with plain text files. The category of tool you'll want to use to get around that is called `a text editor`. Text editors save only the text that you type – there is no hidden formatting or metadata. What you see in a text editor is what a computer will see when it tries to process that data.

This is what it looks like when you open a Word doc in a plain text editor:

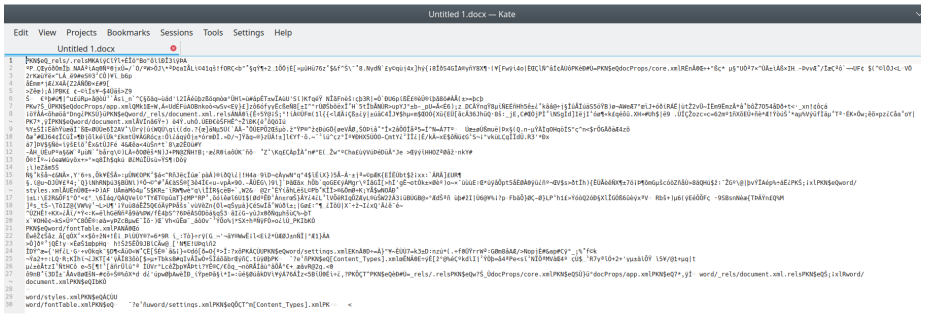
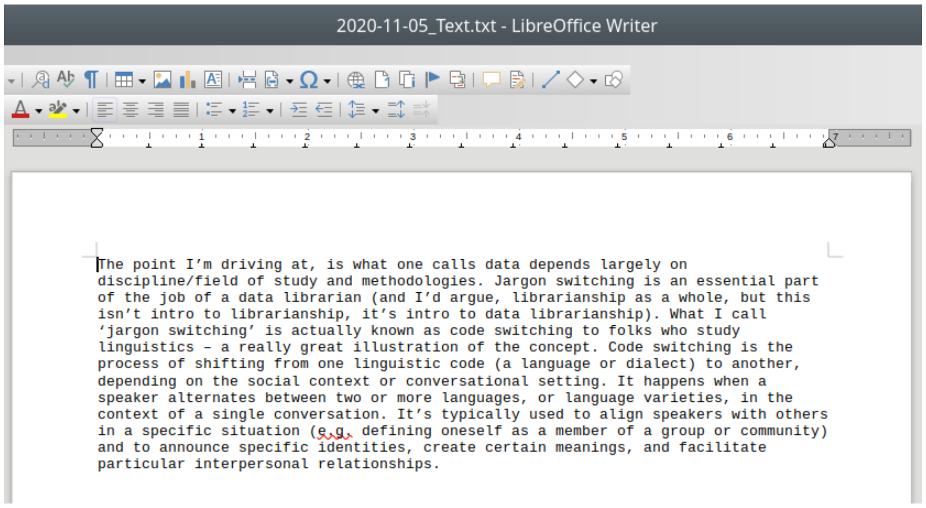


Figure 4: A `.docx` file opened in Kate, a plain text editor

But this is what it looks like when you open a plain text file in a Word processor (in this case, LibreOffice Writer):



```
The point I'm driving at, is what one calls data depends largely on
discipline/field of study and methodologies. Jargon switching is an essential part
of the job of a data librarian (and I'd argue, librarianship as a whole, but this
isn't intro to librarianship, it's intro to data librarianship). What I call
'jargon switching' is actually known as code switching to folks who study
linguistics – a really great illustration of the concept. Code switching is the
process of shifting from one linguistic code (a language or dialect) to another,
depending on the social context or conversational setting. It happens when a
speaker alternates between two or more languages, or language varieties, in the
context of a single conversation. It's typically used to align speakers with others
in a specific situation (e.g. defining oneself as a member of a group or community)
and to announce specific identities, create certain meanings, and facilitate
particular interpersonal relationships.
```

Figure 5: A `.txt` file opened in LibreOffice Writer, a word proccesor

So when working with files for automation or computational purposes, it is more important to focus on meaningful transmission of data as opposed to formatting. However, there are plain files that can be rendered very fancifully while also staying plain-text! One example of this is `Markdown`. These materials are written in Markdown, in fact! A special kind, called RMarkdown, but still just plain Markdown :)

Markdown files, which use the file extension `.md`, are machine and human readable. Markdown applications can be disparate, from simple to-do lists, to extensive manual pages. For example, GitHub renders text via Markdown. For instance, you can inspect the underlying Markdown for this lesson! You can even do things like convert a `.csv` table to a `.md` table, using a helpful resources such as Convert CSV to Markdown.

The Markdown Guide is a helpful resource for learning Markdown but you can also try:

- CommonMark's guide
- Mastering Markdown
- Markdown Here Cheatsheet

## Summary

- File directory structures should be consistent and predictable.

- Apply naming conventions to directories and file names to identify them, to create associations between data elements, and to assist with the long term readability and comprehension.
- If you're interested in automation, then plain-text files are the best to use because they were made for computer-reading.

## Further Reading

Burton, M., Lyon, L., Erdmann, C., & Tijerina, B. (2018). "Shifting to data savvy: the future of data science in libraries."

de la Cruz, J., & Hogan, J. (2016). "'Hello, World!': Starting a Coding Group for Librarians." Public Services Quarterly, 12(3), 249-256.

Ovadia, Steven. "Markdown For Librarians And Academics." Behavioral & Social Sciences Librarian 33.2 (2014): 120-124.

Rosati, D. A. (2016). "Librarians and Computer Programming: understanding the role of programming within the profession of librarianship". Dalhousie Journal of Interdisciplinary Management, 12(1).

Yelton, A. (2015). "Coding for Librarians: Learning by Example." American Library Association.

# Regular Expressions

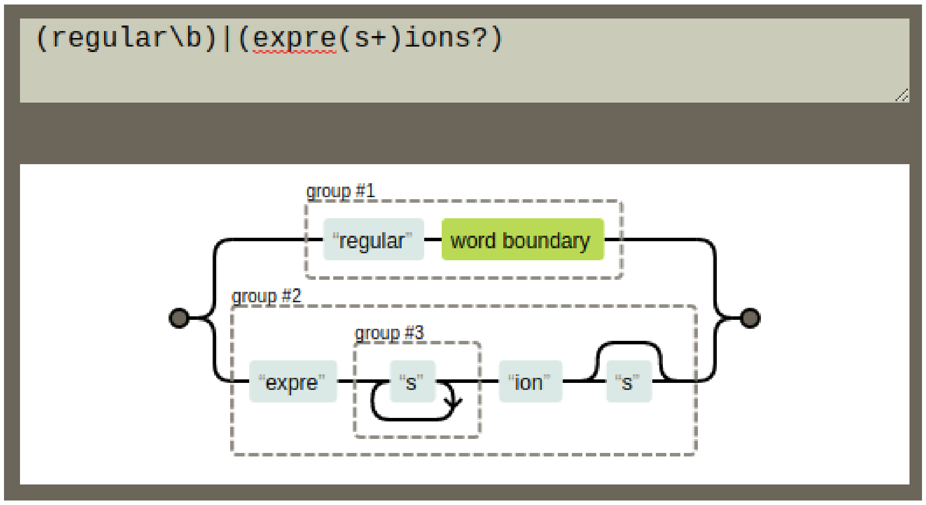
Regular expressions (called `regex` for short) are a way to use patterns to find text. They are an incredibly powerful tool that can amplify your capacity to find, manage, and transform files. You can think of regex as a more complex "find and replac" operation. In a computational setting, we'll refer to the text as a `string`, which is a contiguous sequence of symbols or values. For example, a word, a date, a set of numbers (e.g., a phone number), or an alphanumeric value (e.g., an identifier). A string could be any length, ranging from empty (zero characters) to one that spans many lines of text (including line break characters). You can use regex with pretty much every programming language, and even from within applications like OpenRefine and Google Sheets!

In library searches, we are most familiar with a small part of regular expressions known as the "wild card character" (`*`), but there are many more features to the complete regular expressions syntax. Regular expressions will let you:

- Match on types of characters (e.g. 'upper case letters', 'digits', 'spaces', etc.).
- Match patterns that repeat any number of times (e.g. phone numbers, or addresses).
- Capture the parts of the original string that match your pattern (e.g. get the year from a birthday).

Regular expressions rely on the use of literal characters (like the word "hello") and metacharacters. A `metacharacter` is any American Standard Code for Information Interchange (ASCII) character that has a special meaning. By using metacharacters and possibly literal characters, you can construct a regex for finding strings or files that match a pattern rather than a specific string.

For example, say you wants to change the way that our office telephone numbers on are displayed on the website by removing the parentheses around the area code. Rather than search for each specific phone number (that could take forever and be prone to error) or searching for every open parenthesis character (could also take forever and return many false-positives), you could search for the pattern of a phone number and do bulk transformations! Save yourself all the time!!



```
(regular\b)|(expre(s+)ions?)
```

Figure 6: An example of a regular expression, visualized with REGEXPER

Since regular expressions defines some ASCII characters as "metacharacters" that have more than their literal meaning, it is also important to be able to "escape" these metacharacters to use them for their literal meaning. For example, the period `.` means "match any character", but if you want to match an actual period like at the end of a sentence then you will need to use a \ in front of it to signal that you want to use the period as a plain old period and not a metacharacter meaning "match any character."

A very simple use of a regular expression would be to locate the same word spelled two different ways. For example the regular expression `organi[sz]e` matches both `organise` and `organize`. But because it locates all matches for the pattern in the file, not just for that word, it would also match `reorganise`, `reorganize`, `organises`, `organizes`, `organised`, `organized`, etc.

## Learning common regex metacharacters

Regular expressions tend to be easier to write than they are to read. This is less of a problem if you are the only one who ever needs to maintain the program, but if several people need to watch over it, the syntax can turn into more of a hindrance than an aid. I always make sure to: a) keep a cheatsheet on hand if I ever need to look at a regex, and b) document what my regex is trying to do in a notes document or README file associated with the work I am using regex for. Here are the common regex metacharacters and some norms.

Square brackets can be used to define a list or range of characters to be found. So:

- `[ABC]` matches A or B or C.
- `[A-Z]` matches any upper case letter.
- `[A-Za-z]` matches any upper or lower case letter.
- `[A-Za-z0-9]` matches any upper or lower case letter or any digit.

Note that regex are case sensitive! `A` will not match `a`. There are other metacharacters that help us match:

- `.` matches any character.
- `\d` matches any single digit.
- `\w` matches any part of word character (equivalent to `[A-Za-z0-9]`).
- `\s` matches any space, tab, or newline.
- `\` used to escape the following character when that character is a special character. So, for example, a regular expression that found `.com` would be `\.com` because `.` is a special character that matches any character.
- `^` is an "anchor" which asserts the position at the start of the line. So what you put after the caret will only match if they are the first characters of a line. The caret is also known as a circumflex.
- `$` is an "anchor" which asserts the position at the end of the line. So what you put before it will only match if they are the last characters of a line.
- `\b` asserts that the pattern must match at a word boundary. Putting this either side of a word stops the regular expression matching longer variants of words. So:
  - the regular expression `mark` will match not only `mark` but also find `marking`, `market`, `unremarkable`, and so on.
  - the regular expression `\bword` will match `word`, `wordless`, and `wordlessly`.
  - the regular expression `comb\b` will match `comb` and `honeycomb` but not `combine`.
  - the regular expression `\brespect\b` will match `respect` but not `respectable` or `disrespectful`.

Other useful special characters are:

- `*` matches the preceding element zero or more times. For example, `ab*c` matches "ac", "abc", "abbbc", etc.
- `+` matches the preceding element one or more times. For example, `ab+c` matches "abc", "abbbc" but not "ac".
- `?` matches when the preceding character appears zero or one time.
- `{VALUE}` matches the preceding character the number of times defined by VALUE; ranges, say, 1-6, can be specified with the syntax `{VALUE,VALUE}`, e.g. `\d{1,9}` will match any number between one and nine digits in length.
- `|` means **or**.
- `/i` renders an expression case-insensitive (equivalent to `[A-Za-z]`).

So we can use these all in combination with each other to find text that matches our patterns. For example, when using a pattern `text?.*`, it will find files like:

- `textf.txt`
- `text1.asp`
- `text9.html`

## Group Challenges

To embed this knowledge we won't - however - be using computers. Instead we'll use pen and paper for now.

Let's try out these exercises as a larger group using the cheatsheet I've given you above to be create regular expressions that answer these questions. I'll call out the question, I'll give you all some time, and whoever wants to either chat out an answer or walk us through their answer on audio would be welcome to!

1. What will the regular expression `^[Oo]rgani.e\b` match?
2. What will the regular expression `^[Oo]rgani.e\w*` match?
3. What will the regular expression `[Oo]rgani.e\w+$` match?
4. What will the regular expression `^[Oo]rgani.e\w?\b` match?
5. What will the regular expression `^[Oo]rgani.e\w?$` match?
6. What will the regular expression `\b[Oo]rgani.e\w{2}\b` match?
7. What will the regular expression `\b[Oo]rgani.e\b|\b[Oo]rgani.e\w{1}\b` match?

This logic is useful when you have lots of files in a directory, when those files have logical file names, and when you want to isolate a selection of files. It can be used for looking at cells in spreadsheets for certain values, or for extracting some data from a column of a spreadsheet to make new columns. There are many other contexts in which regex is useful when using a computer to search through a document, spreadsheet, or file structure. Some real-world use cases for regex are included on an ACRL Tech Connect blog.

## Breakout Challenge

You will be put in teams of four to six to get the answers for the exercises below. If you want to check your logic use regex101, myregexp, regex pal or regexper.com. The first three help you see what text your regular expression will match, the latter visualizes the workflow of a regular expression.

1. What will the regular expression `Fr[ea]nc[eh]` match?
2. What will the regular expression `Fr[ea]nc[eh]$` match?
3. What would match the strings `French` and `France` that appear at the beginning of a line?
4. How do you match the whole words `colour` and `color` (case insensitive)?
5. How would you find the whole word `headrest` and or `head rest` but not `head  rest` (that is, with two spaces between head and rest?
6. How would you find a string that ends with four letters preceded by at least one zero?
7. How do you match any four-digit string anywhere?
8. How would you match the date format `dd-MM-yyyy`?
9. How would you match the date format `dd-MM-yyyy` or `dd-MM-yy` at the end of a line only?
10. How would you match publication formats such as `British Library : London, 2015` and `Manchester University Press: Manchester, 1999`?

## Matching & Extracting Strings

We'll do this last set of exercises together as a large group again. For this exercise, open a browser and go to https://regex101.com. Regex101.com is a free regular expression debugger with real time explanation, error detection, and highlighting. It will help us go through the following exercises to do some more complex tasks with regular expressions.

Open the swcCoC.md file, copy the text, and paste that into the test string box.

For a quick test to see if it's working, type the string `community` into the regular expression box.

If you look in the box on the right of the screen, you see that the expression matches six instances of the string `community` (the instances are also highlighted within the text).

I'll call out the question, I'll give you all some time, and whoever wants to either chat out an answer or walk us through their answer on audio would be welcome to!

1. Type `community`. You get three matches. Why not six?

2. If you want to match `community-led` by adding other regex characters to the expression community, what would they be?
3. Change the expression to `communi` and you get 15 full matches of several words. Why?
4. Type the expression`[Cc]ommuni`. You get 16 matches. Why?
5. Type the expression `^[Cc]ommuni`. You get no matches. Why?
6. Find all of the words starting with `Comm` or `comm` that are plural.
7. Let's look at phone numbers: three digits, a dash, and four digits. How would we write a regex expression that matches this?
8. How would we expand the expression above to include an area code (three digits and a dash)?
9. How would we expand the expression above to include a phone number with an area code in parenthesis, separated from the phone number, with or without a space?
10. Country codes are preceded by a `+` and can have up to three digits. We also have to consider that there may or may not be a space between the country code and anything appearing next. How can we expand our phone number regex to include that?

One of the reasons we stress the value of consistent and predictable directory and filenaming conventions is that working in this way enables you to use the computer to select files based on the characteristics of their file names. For example, if you have a bunch of files where the first four digits are the year and you only want to do something with files from `2017`, then you can using regex! Or if you have `journal` somewhere in a filename when you have data about journals, you can use regex to select just those files. Equally, using plain text formats means that you can go further and select files or elements of files based on characteristics of the data within those files.

## Regex in Google Sheets

For our last exercise, I thought we would do one example of how to use regex in an application we probably all work in: Google Sheets! Google Sheets has a function called `REGEXEXTRACT` which takes as the first argument a string and a quoted regular expression in the second, like so: `=REGEXEXTRACT(G2, "[\w.-]+@")`

So let's make a new column based on an old column in Google Sheets, using regex! Do this individually and we'll come back and share strategies as a larger group.

1. Open this Google Sheet with your NYU account: https://docs.google.com/spreadsheets/d/19GNBt_-2EQUKpUnt1T6RbRrAn13XyIIV2Cm7w88W_M0/edit?usp=sharing
2. Click `File > Make a copy`, and make a copy of it in your NYU Google Drive.
3. Look in the `ADDRESS` column and notice that the values contain the latitude and longitude in parenthesis after the library address.
4. Construct a regular expression to match and extract the latitude and longitude into a new column named `latlong`.

## Summary

Please fill in this: https://forms.gle/kCA6zx4UY6zvB5EJ8.

- Regular expressions are used for pattern matching and can be used in many technologies, tools, and programming languages.
- Regular expressions are useful for searching and transforming data.

## Further Resources

Test yourself with RegexCrossword.com/ or via the extra quizzes provided by Library Carpentries: https://librarycarpentry.org/lc-data-intro/03-quiz/index.html.