# Library Carpentry Week One: Some Basics

1730-2030, Monday 9 November, City University London

## Lesson Plan

### Data collection

Conducted as attendees enter the room

Thinking about the programme as a whole, please rate your skill level. Would you say that you know:

1. Nothing
2. A Little
3. Lots
4. Lots and Lots

### Overview

#### Introduction

Welcome to Library Carpentry! This series of introductory workshops on software skills for librarians is an exploratory programme funded by the Software Sustainability Institute and supported by [Software Carpentry](http://software-carpentry.org/) and City University London. My thanks also go to the British Library and the University of Sussex: the former where I worked when I was setting this up and the latter where I work currently as part of the Sussex Humanities Lab, a centrally funded, multi-disciplinary programme that brings 20 or so folks at Sussex together, many of which are new hires in new posts and aims to make the humanities fit for the digital age, and - pleasingly - includes a core member embedded in the Library. I want to emphasise the exploratory element of the programme. With my former BL colleague Nora McGregor, I attended a Software Carpentry workshop 2 years ago that gave me the confidence to really delve deeply into acquiring the software skills that could enhance my work, practice, and research. But Software Carpentry is aimed at scientists and it takes a little sideways thinking to parse the lessons for non-scientists. So having spoken to Greg Wilson from Software Carpentry and having observed that library folks are a consistent and not insignificant fringe of folks attending Software Carpentry events, I started thinking how Software Carpentry lessons, along with materials produced at the Programming Historian and the BL's internal Digital Scholarship Training Programme, could be made into a baseline for a public programme of software skills lessons aimed at library folks that library folks could reuse in their local contexts. So rather than Library Carpentry become a programme I coordinate each and every running of, the hope is that it becomes a set of tools the community to manage, support, enrich, and reuse as the community sees fit. Periodically during the sessions we will be collecting anonymous feedback from you. All this will go into building this future vision for a community driven and supported set of tools that the library community can use to kick start their exploration of software skills.

The rationale for Library Carpentry is twofold. First, as Andromeda Yelton argues in her excellent recent [ALA Library Technology Report](http://journals.ala.org/ltr/issue/view/506) 'Coding for Librarians: learning by example', code is a means for librarians to take control of practice and to empower themselves and their organisation to meet user needs in flexible ways. Second, librarians play a crucial role in cultivating world class research. And in most research areas today world class research relies on the use of software. Librarians with software skills are then well placed to continue that cultivation of world class research.

In order to kick start your exploration of software, this four week programme will look at the following: **SLIDE**

* week 1: Some Basics
* week 2: Controlling Data (with the Shell)
* week 3: Versioning Data (with Git)
* week 4: Cleaning Data (with Open Refine)

#### Where to go for help

**SLIDE** You are a big class. Thankfully each week there will be multiple ways of getting help. First, use your skill level stickers to identify people on your table who can help: you will all be following along from the same worksheets, so someone around you may have got past the point you are stuck at. Second, there are plenty of helpers in the room including me who are there to help if those around you can't. You should all have access to coloured sticky notes: a red sticky note on your laptop indicates to one of us that you need help (it might also alert the attention of someone around you!). So, please use them. Third, before each of the three following classes you will be required to install software: all issues doing this should be reported to the Github issues page for the week, posting something there will alert someone to give you help in advance **bring up Github pages**. Fourth, and finally, as much of the programme is self-directed we encourage you to finish up or repeat tasks after class time: if you run into issues, again report them to the relevant Github issues page.

#### Final points of admin

**SLIDE** We will be in the same place each week, convening at the same time. I will be here each week, though those here to help may change.

Most of the sessions will involve following along from a worksheet. Much of the time you will be encouraged to go along at your own pace. For some of you this will feel like a lot of material, for others it might not feel like enough. Remember that the session is introductory. If you finish early our end time is not a hard stop, so you may of course leave. Alternatively, you might want to use the time to search online for more information or advanced skills guides. You may even wish to deepen your own skills by staying around and helping someone else out: there is nothing better for really getting to know something than teaching someone else! If you don't finish, don't worry, there are no prerequisites between classes and if you have time, you can always carry on at home or on the train.

Tea, coffee, and snacks will be available at every session. Given the timing of the session you are welcome to bring some dinner with you (though you'll need to bring your own cutlery!)

Finally computers are stupid, can frustrate, and as you all have different machines it can be tricky to resolve problems. Please be patient, particularly if your issue is local. Stepping outside and taking a gulp of fresh air always helps.

Wifi: speak to Ernesto

#### This week

**SLIDE**

* Jargon Busting (1800-1845)
* Foundations (1845-1930)
* Regular Expressions (1930-2015)

### Jargon Busting (Group Task)

#### Requirements

* boards/pads
* pens
* sticky notes\*

#### Purpose

* icebreaker
* finding confidence level
* expectation management

#### Task

**SLIDE**

This group task is an opportunity for you to get help understanding terms, phrases, or ideas around code or software development that you've come across and perhaps feel you should know better.

* Start by getting into groups of 5 or 6.
* Next make a big list of all the problem terms, phrases, and ideas that come up. Retain duplicates. Then (taking common words as a starting point) work together to try and explain what a term, phrase, or idea means (note: use both each other and the internet as a resource!). Make a note of those your group resolves and those you are still struggling with. **15 minutes**
* {Trainer}: alongside this make a note of common problem terms, phrases, and ideas that both do and do not map to what Library Carpentry will cover (paying special attention to common problems that will not be covered)}
* Report back on an issue resolved by your group **1 minute each**
* {Trainer}: Report on mapping exercise: what is and isn't going to be covered.

### Foundations

**SLIDE** Before we crack on with using the computational tools at our disposal, I want to spend some time on some foundation level stuff - a combination of best practice and generic skills that frame what we'll be doing for the next month.

#### The Computer is Stupid

**SLIDE** This does not mean that the computer isn't useful. Given a repetitive task, a enumerative task, or a task that relies on memory it can produce results faster, more accurately, and less grudgingly than you or I. Rather when I say that you should keep in mind that the computer is stupid, I mean to say that computer only does what you tell it to. If it throws up an error it is often not your fault, rather in most cases the computer has failed to interpret what you mean because it can only work with what it knows (ergo, it is bad at interpreting). This is not to say that the people who told the computer what to tell you when it doesn't know what to do couldn't have done a better job with error messages, for they could. So keep in mind as we go along that if you find an error message frustrating it isn't the computer's fault that it is giving you an archaic and incomprehensible error message, it is person's.

#### Why take an automated or computational approach

**SLIDE** Otherwise known as the 'why not do it manually?' question. To start with, I'm not anti-manual. I do plenty of things manually that a machine could do in an automated way because either a) I don't know how to automate the task or b) I'm unlikely to repeat the task and estimate that automating it would take longer. However once you know you'll need to repeat a task you have a compelling reason to consider automating it. This is one of the main areas in which programmatic ways of doing outside of IT service environments are changing library practice. Andromeda Yelton, a US based librarian closely involved in the Code4Lib movement, recently put together an excellent American Library Association Library Technology Report called "Coding for Librarians: Learning by Example." The report is pitched at a real world relevance level, and in it Andromeda describes scenarios library professionals told her about where learning a little programming, usually learning ad-hoc, had made a difference to their work, to the work of their colleagues, and to the work of their library.

Main lessons:

* **Borrow, Borrow, and Borrow again**. This is a mainstay of programming and a practice common to all skill levels, from professional programmers to people like us hacking around in libraries;
* **The correct language to learn is the one that works in your local context**. There truly isn't a best language, just languages with different strengths and weaknesses, all of which incorporate the same fundamental principles;
* **Consider the role of programming in professional development**. That is both yours and of those you manage;
* **Knowing (even a little) code helps you evaluate projects that use code**. Programming can seem alien. Knowing some code makes you better at judging the quality of software development or planning activity that include software development
* **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...)

**SLIDE: Cost/benefit slide**

#### Keyboard shortcuts are your friend

**SLIDE** Though we will get more computational over the next 3 weeks, we can start our adventure into programming - as many of you will have already - with very simple things like keyboard shortcuts. We all have our favourites. Labour saving but also exploiting this stupid machine in best possible way. Alongside the very basic ones (ctrl+s for save; ctrl+c for copy; ctrl+x for cut; ctrl+v for paste) my favourite (in a Windows or Linux machines) is alt+tab, a keyboard shortcut that switches between programmes {Trainer: ask other helpers what their favourites are}. You can do all the lessons in Library Carpentry without keyboard shortcuts, but note that they'll likely come up a lot.

#### Plain text formats are your friend

**SLIDE** Why? Because computers can process them!

If you want computers to be able to process your stuff I'd encourage you to get in the habit where possible of using platform agnostic formats such as .txt for notes and .csv or .tsv for tabulated data (the latter pair are just spreadsheet formats, separated by commas and tabs respectively). These plain text formats are preferable to the proprietary formats used as defaults by Microsoft Office or iWork because they can be opened by many software packages and have a strong chance of remaining 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 still take appropriate action to make your work accessible. Compared to .doc or .xls these formats have the additional benefit of containing only machine-readable elements. Whilst using bold, italics, and colouring to signify headings or to make a visual connection between data elements is common practice, these display-orientated annotations are not (easily) machine-readable and hence can neither be queried and searched nor are appropriate for large quantities of information (the rule of thumb is if you can't find it by ctrl+f it isn't machine readable). Preferable are simple notation schemes such as using a double-asterisk or three hashes to represent a data feature: in my own notes, for example, three question marks indicate something I need to follow up on, chosen because ??? can easily be found with a CTRL+F search.

??? was also chosen by me because it doesn't clash with existing schemes. Though it is likely that notation schemes will emerge from existing individual practice, existing schema are available to represent headers, breaks, et al. One such scheme is Markdown, a lightweight markup language. Markdown files are as .md, are machine readable, human readable, and used in many contexts - GitHub for example, renders text via Markdown. An excellent [Markdown cheat sheet is available on GitHub](https://github.com/adam-p/markdown-here) for those who wish to follow – or adapt – this existing schema. Notepad++ http://notepad-plus-plus.org/ is recommended for Windows users as a tool to write Markdown text in, though it is by no means essential for working with .md files. Mac or Unix users may find Komodo Edit, Text Wrangler, Kate, or Atom helpful.

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

**SLIDE** Working with data is made easier by structuring your stuff in a consistent and predictable manner.

Why?

Without structured information, our lives would be much poorer. As library and archive people we know this. But I'll 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 research 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.

**look at webpages** 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
* <http://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
* <http://www.theguardian.com/uk-news/2014/feb/20/rebekah-brooks-rupert-murdoch-phone-hacking-trial>

In archival catalogues, URLs structured by a single data element are often used. The British Cartoon Archive structures its online archive using the format:

* ROOT/record/REF
* <http://www.cartoons.ac.uk/record/SBD0931>

And the Old Bailey Online uses the format:

* ROOT/browse.jsp?ref=REF`
* <http://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 data structures that are readable both by humans and machines. Transferring this to your stuff 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 Library Carpentry will cover.

In practice, the structure of a good archive might look something like this:

* A base or root directory, perhaps called 'work'.
* A series of sub-directories such as 'events', 'data', ' projects' et cetera
* Within these directories are series of directories for each event, dataset or project. Introducing a naming convention here that includes a date element keeps the information organised without the need for subdirectories by, say, year or month.

All this should help you remember something you were working on when you come back to it later (call it real world preservation).

The crucial bit for our purposes, however, is the file naming convention you choose. The name of a file is important to ensuring it and its contents are easy to identify. 'Data.xslx' doesn't fulfil 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 connection.

To recap, the key points about structuring your data are:

* Data structures should be consistent and predictable
* Consider using semantic elements or data identifiers to data directories
* Fit and adapt your data structure to your work
* 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 of your data structures

### Regular Expressions

**SLIDE** One of the reason why I have stressed 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 name. So, 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 '2014', then you can. Or if you have 'journal' somewhere in a filename when you have data about journals, you can use the computer to select just those files then do something with them. 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* files.

A powerful means of doing this selecting based on file characteristics is to use regular expressions, often abbreviated to regex. A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations. It will let you:

* Match on types of character (e.g. 'upper case letters', 'digits', 'spaces', etc.)
* Match patterns that repeat any number of times
* Capture the parts of the original string that match your pattern

As most computational software has regular expression functionality built in and as many computational tasks in libraries are built around complex matching, it is good place for Library Carpentry to start in earnest.

**SLIDE** 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".

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 it would also match reorganise. So there are a bunch of special syntax that help us be more precise.

The first we've seen: square brackets can be used to define a list or range of characters to be found. So: **SLIDE**

* [ABC] matches A or B or C
* [A-Z] matches any upper case letter
* [A-Za-z0-9] matches any upper or lower case letter or any digit (note: this is case-sensitive)

Then there are: **SLIDE**

* . 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
* ^ defines the start of the string
* $ defines the end of the string

**SLIDE** {Question}: So, what is ^[Oo]rgani.e$ going to match.

Other useful special characters are **SLIDE**:

* \* matches when the proceeding character appears any number of times including zero
* + matches when the proceeding character appears any number of times excluding zero
* ? matches when the proceeding character appears one or zero times
* {VALUE} matches the proceeding character the number of times define by VALUE; ranges can be specified with the syntax {VALUE,VALUE}
* | means or.

{Questions}: So, what are these going to match?

* **SLIDE** ^[Oo]rgani.e\w\*$
* **SLIDE** ^[Oo]rgani.e\w+$
* **SLIDE** ^[Oo]rgani.e\w?$
* **SLIDE** ^[Oo]rgani.e\w{2}$
* **SLIDE** ^[Oo]rgani.e$|^[Oo]rgani.e\w{2}$

**SLIDE** This logic is super 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. Or for looking at cells in spreadsheets for certain values. Or for extracting some data from a columns of a spreadsheet to make a new columns. I could go on. The point is, it is super useful in many contexts. To embed this knowledge we won't - however - be using computers. Instead we'll use pen and paper. I want you to work in teams of 5 or 6 to work through the exercises in the handout. I have an answer sheet over here if you want to check where you've gone wrong. When you finish, I'd like you to split your team into two groups and write each other some tests. These should include a) strings you want the other team to write regex for and b) regular expressions you want the other team to work out what they would match. Then test each other on the answers. If you want to check your logic, use [regex101](https://regex101.com/).

#### Handout

What does Fr[ea]nc[eh] match?

What does Fr[ea]nc[eh]$ match?

What would match strings that begin with French and France only? {France|French}

How do you match the words 'colour' and 'color' (case insensitive)? {colou?r}

How would you find 'headrest' and 'head rest' but not 'head rest'? {head?rest}

How would you find a 4 letter word that begins a string and is preceded by at least one zero? {0+[a-z]{4}$}

How do you match any 4 digit string? {\d{4}}

How would you match the date format 'dd-MM-yyyy'? {\d{2}-\d{2}-\d{4}}

How would you match the date format 'dd-MM-yyyy' or 'dd-MM-yy' at the end of a string only? {\d{2}-\d{2}-\d{2,4}$}

How would you match a thirteen digital ISBN? {^$}

How would you match publication formats such as 'British Library : London, 2015' and 'Manchester University Press : Manchester, 1999'? {.\* : .\*, \d{4}}

## Next week

**SLIDE**

Shell.

Instructions of what to install per operating system on the Github page.

Anyone who wants help now, we are happy to help.

### References

James Baker , "Preserving Your Research Data," *Programming Historian* (30 April 2014), <http://programminghistorian.org/lessons/preserving-your-research-data.html>. The sub-sections 'Plain text formats are your friend' and 'Naming files sensible things is good for you and for your computers' are reworked from this lesson.

Owen Stephens, "Working with Data using OpenRefine", \*Overdue Ideas" (19 November 2014), <http://www.meanboyfriend.com/overdue_ideas/2014/11/working-with-data-using-openrefine/>. The section on 'Regular Expressions' is reworked from this lesson developed by Owen Stephens on behalf of the British Library

Andromeda Yelton, "Coding for Librarians: Learning by Example", *Library Technology Reports* 51:3 (April 2015), doi: [10.5860/ltr.51n3](http://dx.doi.org/10.5860/ltr.51n3)

Fiona Tweedie, "Why Code?", *The Research Bazaar* (October 2014), <http://melbourne.resbaz.edu.au/post/95320810834/why-code>