Data Structures & Models - Data Scripting Bootcamp

0: Introduction

0.1 Bootcamp Schedule

1. (Meta)Data Basics - including touching modeling, representations, and structures (1 hour)
2. Using the command line (in the *nix Shell and Bash) to interact with (meta)data (1 hour 30 minutes)
3. Versioning & collaborating on (meta)data using Git & GitHub (1 hour 15 minutes)
4. Querying & updating (meta)data contained in traditional MySQL databases (a popular database selection at CUL & elsewhere) (1 hour)
5. Wrap-up & Requests for future workshops (15 minutes)

0.2 Workshop Logistics

- CUL Metadata Working Group 2016-2017 Blurb
- Bootcamp Curriculum & Resources Location
  - Online materials for use in exercises
  - Parts of this will be more presentation-oriented, other parts will be self-directed
- This is a Bootcamp, so it won't go in depth. We just want to help you get acclimated to these technologies.
  - Use your Google-fu skills wisely.
  - Please be patient with yourself + others.
  - Help your neighbors as you are able.
- Follow the Hacker School Rules (also known as the Recurse Center Social Rules), please.
- What will we cover today? What can we cover in a future MWG meeting?
  - Add any new or unknown terms to a shared CUL MWG Glossary?
  - This will lead into our next two workshops.

1: Data Modeling

1.1: What is "Data Modeling"?

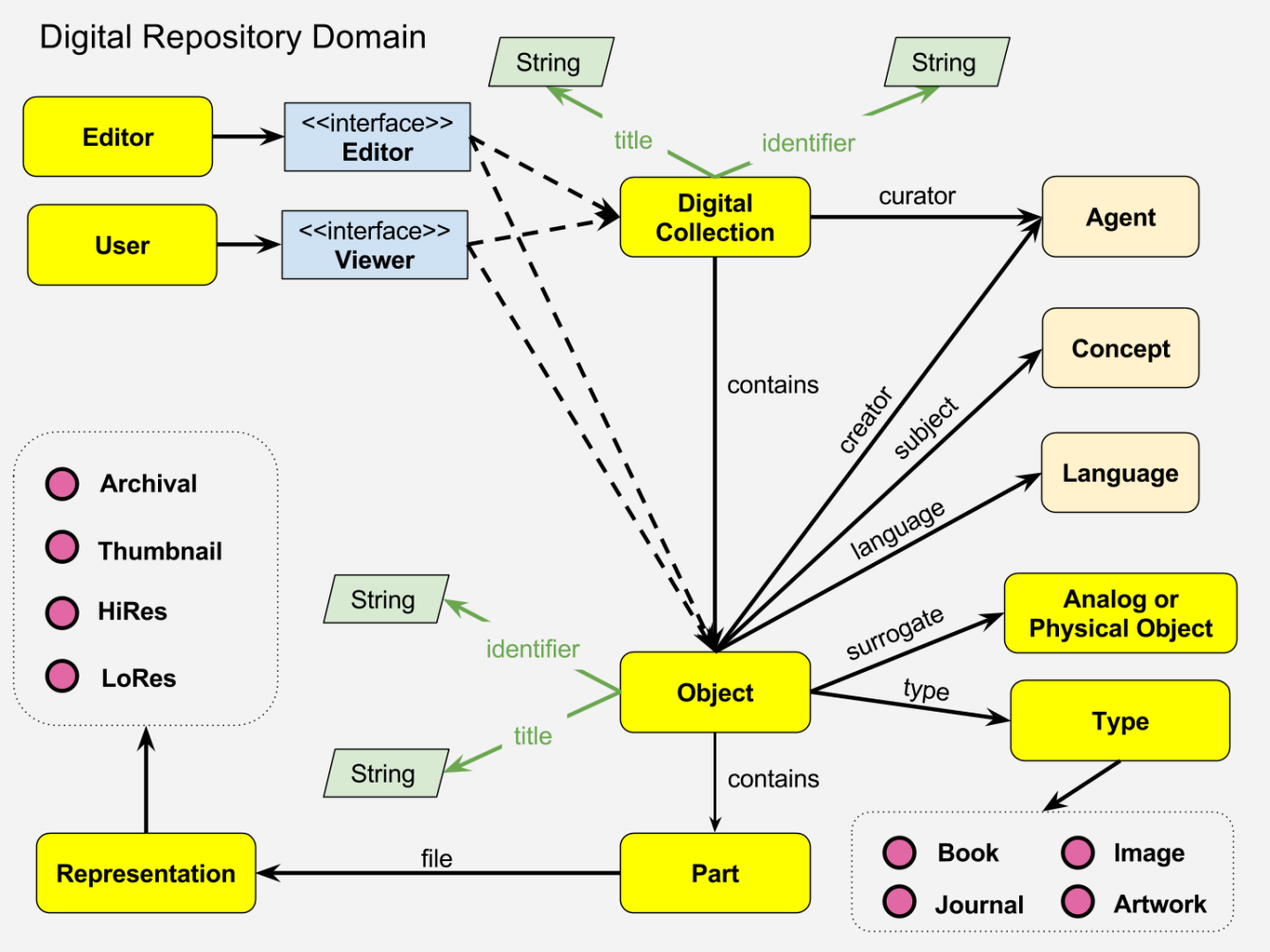
Data model: Represents the fundamental concepts that are relevant to a system. Concepts are often represent by classes that can have attributes. The combination of these create a data model, as they encapsulate the data that will be used by your system.

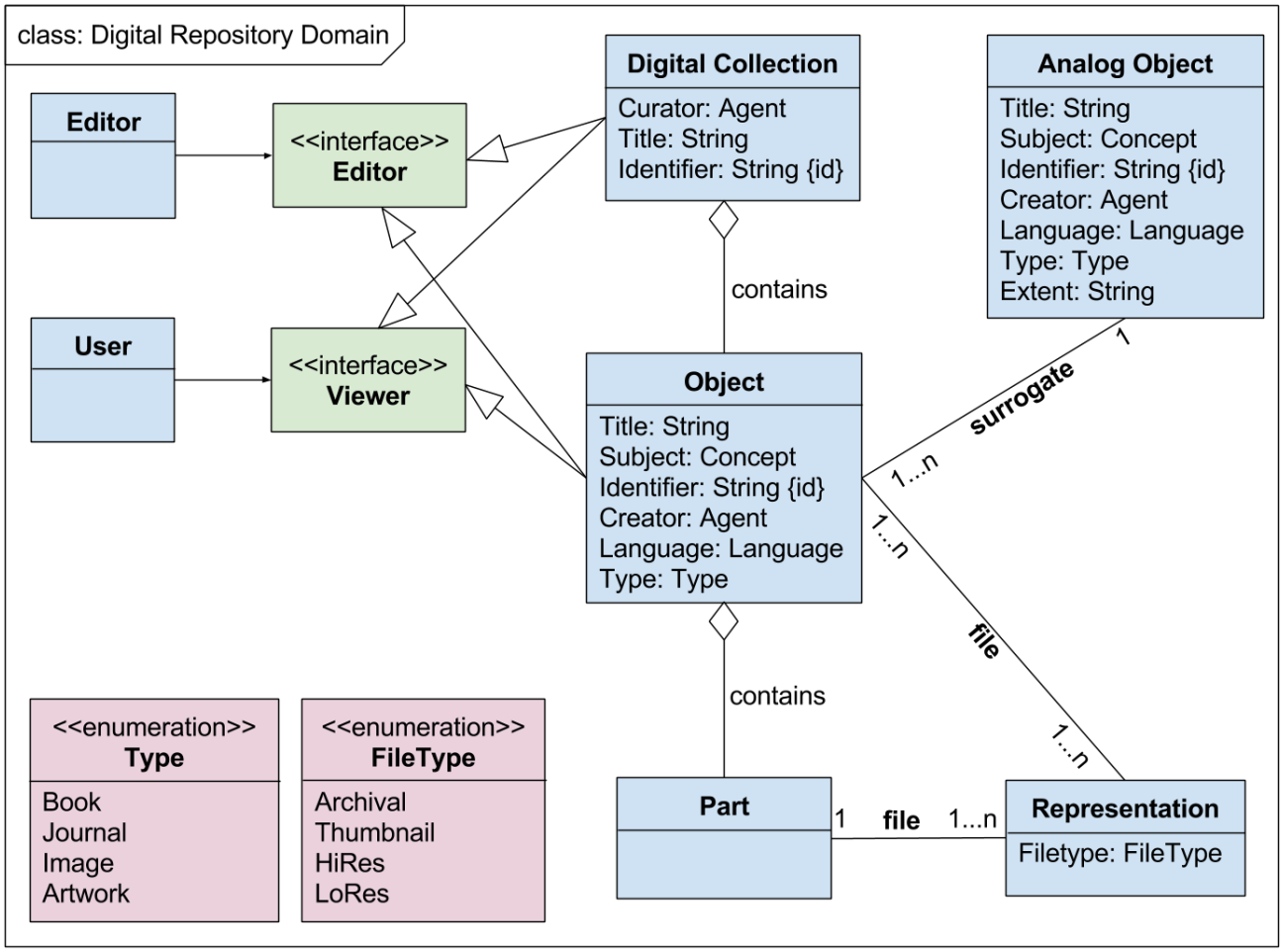
- Data Model is sometimes an Application Artifact
- Data Models act as a specification of a System.
- Model documents serve as documentation for design team, developers, and users.
- Data Models are ultimately a way to communicate understandings that bridge the conceptual and the functional.

1.2: Why do we "Data Model"?

- Modeling for Design Feedback
- Provides common language and point of discussion for programmers, designers & domain experts.

1.3: What Data Modeling Can Look Like?




class: Digital Repository Domain
Editor
<<interface>>
Editor
Digital Collection
Curator: Agent
Title: String
Identifier: String {id}
Analog Object
Title: String
Subject: Concept
Identifier: String {id}
Creator: Agent
Language: Language
Type: Type
Extent: String
User
<<interface>>
Viewer
contains
Object
Title: String
Subject: Concept
Identifier: String {id}
Creator: Agent
Language: Language
Type: Type
surrogate
1
1...n
1...n
file
1...n
contains
<<enumeration>>
Type
Book
Journal
Image
Artwork
<<enumeration>>
FileType
Archival
Thumbnail
HiRes
LoRes
Part
1
file
1...n
Representation
Filetype: FileType

```ruby
module DPLA::MAP
  class Aggregation < ActiveTriples::Resource
    configure :type => RDF::ORE.Aggregation

    validates_presence_of :sourceResource, :originalRecord, :isShownAt, :object, :provider

    property :sourceResource, :predicate => RDF::EDM.aggregatedCHO, :class_name => 'DPLA::MAP::SourceResource'
    property :dataProvider, :predicate => RDF::EDM.dataProvider, :class_name => 'DPLA::MAP::Agent'
    property :originalRecord, :predicate => RDF::DPLA.originalRecord
    property :hasView, :predicate => RDF::EDM.hasView, :class_name => 'DPLA::MAP::WebResource'
    property :intermediateProvider, :predicate => RDF::DPLA.intermediateProvider, :class_name => 'DPLA::MAP::Agent'
    property :isShownAt, :predicate => RDF::EDM.isShownAt, :class_name => 'DPLA::MAP::WebResource'
    property :object, :predicate => RDF::EDM.object, :class_name => 'DPLA::MAP::WebResource'
    property :preview, :predicate => RDF::EDM.preview, :class_name => 'DPLA::MAP::WebResource'
    property :provider, :predicate => RDF::EDM.provider, :class_name => 'DPLA::MAP::Agent'
    property :rightsStatement, :predicate => RDF::EDM.rights, :class_name => 'DPLA::MAP::RightsStatement'

    def jsonld_context
      DPLA::MAP::CONTEXT['@context']
    end

    def to_jsonld
      JSON::LD::API.frame(JSON.parse(dump(:jsonld)), DPLA::MAP::FRAME)
    end
  end
end
```

https://github.com/dpla/dpla_map/blob/map-4.0/lib/dpla/map/aggregation.rb

```xml
  <rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Organization" rdfs:label="Organization"
    rdfs:comment="An organization." vs:term_status="stable">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
<!--    <rdfs:subClassOf><owl:Class rdf:about="http://xmlns.com/wordnet/1.6/Organization"/></rdfs:subClassOf> -->
    <rdfs:subClassOf rdf:resource="http://xmlns.com/foaf/0.1/Agent"/>
    <rdfs:isDefinedBy rdf:resource="http://xmlns.com/foaf/0.1/"/>
    <owl:disjointWith rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <owl:disjointWith rdf:resource="http://xmlns.com/foaf/0.1/Document"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Group" vs:term_status="stable" rdfs:label="Group"
    rdfs:comment="A class of Agents.">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <rdfs:subClassOf rdf:resource="http://xmlns.com/foaf/0.1/Agent"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Agent" vs:term_status="stable" rdfs:label="Agent"
    rdfs:comment="An agent (eg. person, group, software or physical artifact).">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <owl:equivalentClass rdf:resource="http://purl.org/dc/terms/Agent"/>
<!--    <rdfs:subClassOf><owl:Class rdf:about="http://xmlns.com/wordnet/1.6/Agent-3"/></rdfs:subClassOf> -->
  </rdfs:Class>
```

http://xmlns.com/foaf/spec/

2: Data Structures & Scripting

Data structures: A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.

Data structures intersect with Data Models. Data models represent a concept to humans interacting with systems and computers. At a lower level than models, any type of data must be represented internally to the computer, and this is by means of data structures. For example, an array is a data structure (a number of elements in a specific order, typically all of the same type), as well as dictionaries (collection of (key, value) pairs, such that each possible key appears at most once in the collection). Also, the classes that form your data model, are data structures too, any representation of a specific data object has to be in form of a data structure.

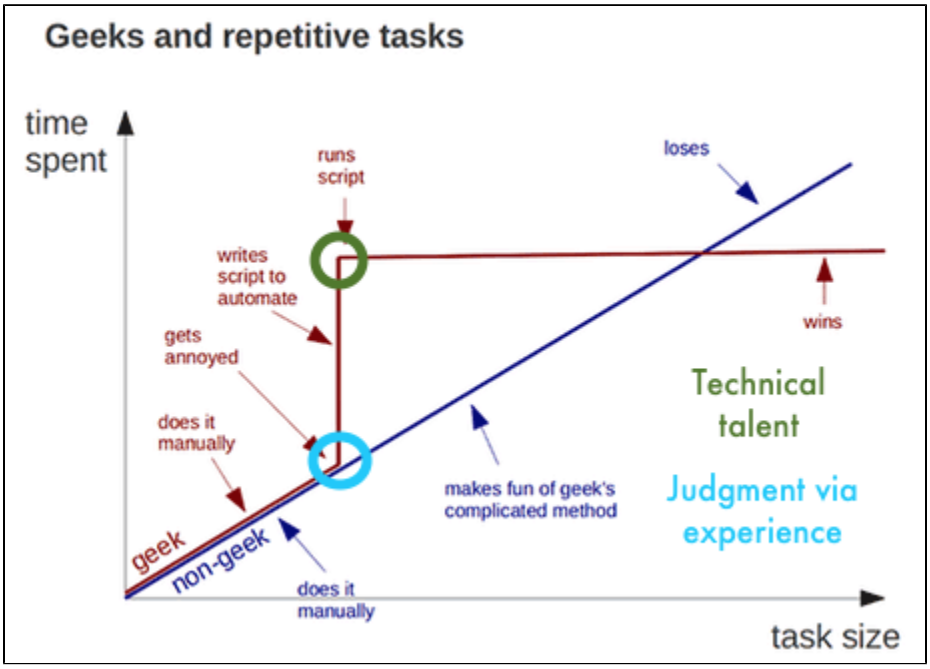
Thus, data structures are where we start for data scripting. First, some prerequisites on scripting generally.

2.1: Scripting Foundations

The following is taken mostly from the work of Library Carpentry's Data Intro for Librarians module. It is "some foundation level stuff - a combination of best practice and generic skills"

Some Points on Scripting

- The Computer is Stupid. It can only do (whether process errors or explain errors of processes) what humans tell it to do.
- Borrow, Borrow, and Borrow again; this is a mainstay of programming and a practice common to all skill levels. Borrow other people's code. Reuse and adapt existing scripts.
- The correct (programming) language to learn is the one that works in your local context.
- Consider the role of programming in professional development.
- Knowing (even a little) code helps you evaluate projects that use code.
- Automate some of your metadata work to make the time to do something else!



Why Automate? Andy Kirk @visualisingdata

2.2: Simple Place to Start for Enabling Data Scripting

Keyboard shortcuts are your friend

Starting with relatively simple things like keyboard shortcuts can both help with automation and introduce programming concepts. This also extends to macros and spreadsheet formulas - these are scripting and mean you already have your start in programming. Honestly, many library technologists and programmers often started with these sort of tasks.

Plain text formats are your friend

For all computers to be able to process your stuff, use platform-agnostic formats such as .txt for notes and .csv or .tsv for tabulated data as much as possible. These plain text formats are preferable to the proprietary formats used as defaults by Microsoft Office 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. Machine-readable elements usually mean not including textual formats like bold, italics, and coloring. This type of formatting can be helpful 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).

Though it is likely that notation schemes will emerge from existing individual practice, existing schema are available to represent headers, breaks, et al. if you need them. 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 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. Combined with pandoc, a markdown file can be exported to PDF, LaTeX or other formats, so it is a great way to create machine-readable, easily searchable documents that can be repurposed in many ways. It is a universal document converter.

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

Working with data is made easier by structuring your files and directories in a consistent and predictable manner.

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.

A typical example of the former (containing semantic elements) 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 among other repository websites, URLs structured by a single data element are often used. The NLA's TROVE structures its online archive using the format:

- ROOT/record-type/REF
- http://trove.nla.gov.au/work/6315568

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.

In practice, the structure of a good data directory 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 or need to work on a large number of files at once.

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.

3: Data Queries & Regular Expressions

Filenaming conventions become important when you want to use some simple automation and scripting tools to interact with your files and data.

Consistent and semantic filenaming and directory structure means, 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.

3.1: Regular Expressions

A powerful means of doing this selecting & querying based on file or data 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. Regular expressions are typically surrounded by `/` characters, though we will (mostly) ignore those for ease of comprehension. Regular expressions 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 to start for data scripting here.

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`, `reorganize`, `organises`, `organizes`, `organised`, `organized`, et cetera, because we've not specified the beginning or end of our string. So there are a bunch of special syntax that help us be more precise.

3.2: Regular Expressions Syntax (Starter)

The first we've seen: 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-z0-9]` matches any upper or lower case letter or any digit (note: this is case-sensitive)

Then there are:

- `.` 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
- `\` NB: this is also 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.
- `^` asserts the position at the start of the line. So what you put after it will only match the first characters of a line or contents of a cell.
- `$` asserts the position at the end of the line. So what you put after it will only match the last character of a line of contents of a cell.
- `\b` adds a word boundary. Putting this either side of a stops the regular expression matching longer variants of words. So:
    - the regular expression `foobar` will match `foobar` and find `666foobar`, `foobar777`, `8thfoobar8th` et cetera
    - the regular expression `\bfoobar` will match `foobar` and find `foobar777`
    - the regular expression `foobar\b` will match `foobar` and find `666foobar`
    - the regular expression `\bfoobar\b` will find `foobar`

Other useful special characters are:

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

3.3: Regular Expression Exercises

Below are some examples to get a feeling for using Regular Expressions. We will go through the first few together to get a feeling for their logic - which will be extremely helpful for running matches for certain filenames, call values in a table, or matches in the data itself.

If there is time, run through the following examples by yourself or in small groups. You do not need to get through all of these, and we will be referring back to these in the next session.

If you want to check your Regular Expression logic in real-time, use regex101, regexr, myregexp, regex pal, or regexper.com: the first four help you see what text your regular expression will match, the latter shows the workflow of a regular expression.

3.3.1: Using special characters in regular expression matches

```
organiseorganize
Organise
Organize
organife
Organike
```

Or, any other string that starts a line, begins with a letter o in lower or capital case, proceeds with rgani, has any character in the 7th position, and ends with the letter e.

3.3.2: Using special characters in regular expression matches: \w*

```
organise

Organize
organifer
Organi2ed111
```

Or, any other string that starts a line, begins with a letter o in lower or capital case, proceeds with rgani, has any character in the 7th position, follows with letter e and zero or more characters from the range [A-Za-z0-9].[Oo]rgani.e\w+$

3.3.3: Using special characters in regular expression matches: \w+$

```
organiser
Organized
organifer
Organi2ed111
```

Or, any other string that ends a line, begins with a letter o in lower or capital case, proceeds with rgani, has any character in the 7th position, follows with letter e and one or more characters from the range [A-Za-z0-9].

3.3.4: Using special characters in regular expression matches: \w?\b

```
organise
Organized
organifer
Organi2ek
```

Or, any other string that starts a line, begins with a letter o in lower or capital case, proceeds with rgani, has any character in the 7th position, follows with letter e, and ends with zero or one characters from the range [A-Za-z0-9].

3.3.5: Using special characters in regular expression matches: \w?$

```
organise

Organized

organifer
Organi2ek
```

Or, any other string that starts and ends a line, begins with a letter o in lower or capital case, proceeds with rgani, has any character in the 7th position, follows with letter e and zero or one characters from the range [A-Za-z0-9].

3.3.6: Using special characters in regular expression matches: \w{2}\b

```
organisers

Organizers
organifers
Organi2ek1
```

Or, any other string that begins with a letter o in lower or capital case after a word boundary, proceeds with rgani, has any character in the 7th position, follows with letter e, and ends with two characters from the range [A-Za-z0-9].

3.3.7: Using special characters in regular expression matches: \w{1}\b

```
organise

Organi1e
Organizer
organifed
```

Or, any other string that begins with a letter o in lower or capital case after a word boundary, proceeds with rgani, has any character in the 7th position, and end with letter e, or any other string that begins with a letter o in lower or capital case after a word boundary, proceeds with rgani, has any character in the 7th position, follows with letter e, and ends with a single character from the range [A-Za-z0-9].

3.3.8: Using Square Brackets

```
French

France

Frence

Franch
```

This will also find words where there are characters either side of the solutions above, such as Francer, foobarFrench, and Franch911.

3.3.9: Using dollar signs

```
French

France

Frence

Franch
```

This will also find strings at the end of a line. It will find words where there were characters before these, for example foobarFrench.

3.3.10: Introducing options

```
^France|^French
```

This will also find words where there were characters after French such as Frenchness.

3.3.11: Case insensitivity

```
\b[Cc]olou?r\b|\bCOLOU?R\b /colou?r/i
```

In real life, you should only come across the case insensitive variations colour, color, Colour, Color, COLOUR, and COLOR (rather than, say, coLour). So based on what we know, the logical regular expression is \b[Cc]olou?r\b|\bCOLOU?R\b. An alternative more elegant option we've not discussed is to take advantage of the / delimiters and add an ignore case flag: so /colou?r/i will match all case insensitive variants of colour and color.

3.3.12: Word boundaries

```
\bhead ?rest\b
```

Note that although \bhead\s?rest\b does work, it will also match zero or one tabs or newline characters between head and rest. So again, although in most real world cases it will be fine, it isn't strictly correct.

3.3.13: Matching non-linguistic patterns (Write the Regex for the Query)

0+[a-z]{4}\b

3.3.14: Matching digits (Write the Regex for the Query)

```
\d{4}
```

Note this will also match 4 digit strings within longer strings of numbers and letters.

3.3.15: Matching dates (Write the Regex for the Query)

```
\b\d{2}-\d{2}-\d{4}\b
```

Depending on your data, you may chose to remove the word bounding.

3.3.16: Matching multiple date formats (Write the Regex for the Query)

```
\d{2}-\d{2}-\d{2,4}$
```

```
Note this will also find strings such as 31-01-198 at the end of a line, so you may wish to check your data and
revise the expression to exclude false positives. Depending on your data, you may chose to add word bounding at
the start of the expression.
```

3.3.17: Matching publication formats (Write the Regex for the Query)

```
.* : .*, \d{4}
```

Without word boundaries you will find that this matches any text you put before British or Manchester. Nevertheless, the regular expression does a good job on the first look up and may be need to be refined on a second depending on your data.

References & Further Reading

- HydraConnect 2016 Introduction to Data Modeling Workshop Slides (with speaker notes)

- Regular Expression Websites
  - regex101, regexr, myregexp, regex pal, regexper.com or regexcrossword.
- Library Carpentry: Data Introduction for Librarians & References from that site:
  - 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
  - Fiona Tweedie, “Why Code?”, *The Research Bazaar* (October 2014), http://melbourne.resbaz.edu.au/post/95320810834/why-code