The Programming Historian

http://programminghistorian.org

Printed Edition.

Edited by

Adam Crymble, Fred Gibbs, Allison Hegel, Caleb McDaniel, Ian Milligan, Miriam Posner, Evan Taparata, Jeri Wieringa, Jeremy Boggs, and William J. Turkel.

Print Edition collated by

Adam Crymble.

2012-2015

The project is published by the *Editorial Board of the Programming Historian.*

Editorial Board of the *Programming Historian*

Albuquerque, United States
Houston, United States
Fairfax, United States
London, United Kingdom
Los Angeles, United States
Minneapolis, United States
Waterloo, Canada.

The Editorial Board of the *Programming Historian* is a non-legal entity, controlled entirely by its members. It exists solely to bring high quality technical tutorials to humanities scholars looking to improve their research skills.

http://programminghistorian.org

First printed on paper 2016.
Printed in the United Kingdom

ISSN 2397-2068 Online

The Editorial Board of the *Programming Historian* has no responsibility for the persistence or accuracy of URLs for external or third-party Internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

This book is dedicated to our
authors, reviewers, editors, and readers.

And to anyone who wants to learn.

Contents

Introduction

Part One: Setting Up

Part Two: Acquiring Data

Part Three: Transforming Data

Preface

Humanists need somewhere to learn digital skills.

In 2007, William J. Turkel and Alan MacEachern provided the solution. The pair published *The Programming Historian*, a series of tutorials designed to teach historians programming skills.[1] In 2011, their research assistant at the Network in Canadian History & Environment (NiCHE), Adam Crymble, put forth the following 'supersecret' proposal, which became the *Programming Historian* that you're reading today:

NiCHE

[SUPERSECRET]

10 February 2011

The Programming Historian 2: A Sustainable, Collaborative Open Access Model for Academic Publishing

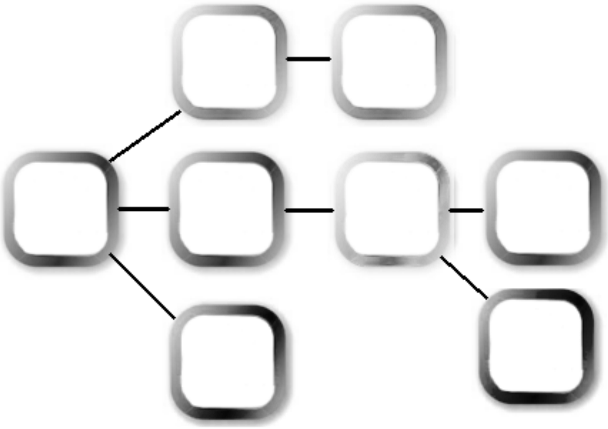
- A Proposal -

Academic publishing, meet the Internet. Most academics are suspicious of online work. The criticisms are justified; online publishing lacks peer review, the writing tends to be unpolished, and works are impermanent or frequently changed, frustrating attempts to cite. Those who support open access publishing – of whom there are many amongst digital humanists – dismiss the cost, sluggishness and rigidity of the academic publishing model. And yet relatively little work has been done to bring the two sides together. Instead, the conversations are dismissive and divisive. Digital humanists publish to their blogs or Twitter feeds, and traditional academics ignore them, while opting for more respectable places to publish. Rather than perpetuate the divide, a positive solution-based model is needed, one that addresses the concerns of those who support academic publishing and leads by example. This solution is a collaborative, open access version of *The Programming Historian,* which maintains rigorous standards, peer review, and relative permanence, while taking advantage of the flexibility of Web 2.0, Drupal and an engaged community.

This web-based project will provide an infrastructure to allow members of the community to contribute modules (chapters) for review under a

[1] William J. Turkel & Alan MacEachern, *The Programming Historian* (2007): http://niche-canada.org/wp-content/uploads/2013/09/programming-historian-1.pdf.

Creative Commons "by-sa" license. The editor(s) will try each module to ensure it is accurate and work with the author to ensure the lesson is comprehensive for our target user: an educated but non-specialist researcher interested in learning new techniques. Approved modules will then added to the collection. Quality modules will be sought over quantity of content.



To ensure the site does not become a code repository, mechanisms will be put in place to promote a linear learning model similar to a book, but with a bit of *choose your own adventure* thrown in. Authors seeking to make submissions will be required to build directly upon an existing module, or to ensure that their module is entirely self-sufficient. In this way readers can always find the beginning of a set of lessons, and contributors are encouraged to work within the existing structure. The image to the left shows how this collaborative building might look. The entire project may have several starting branches, depending on what type of project the authors seek to contribute: physical computing, text mining, website building etc.

That 'super secret' plan became our open project.

We have not created exactly the project first envisioned by Crymble, but the spirit is still there. Fred Gibbs, who joined as an editor in 2012, was instrumental in adding extra flexibility, dropping the requirement to build directly upon an existing lesson and thus bringing in a much more diverse set of tutorials. Subsequently, we opted for CC-BY licenses instead of CC-BY-SA. And we never used Drupal for our website, though we could have done.

Most importantly, the project, and **this book, is not designed to be read in order**. Instead, read the lessons that are useful for your own needs. Jump around sections, and stop reading a tutorial when you've taken from it what you need to know. This is not a linear reading experience – though you are of course welcome to take that approach as well.

Since we first went public with the new site in 2012, we have published 48 tutorials with the help of dozens of authors, reviewers, workshop attendees, and students. More than 200,000 people have logged in to learn a new skill. And while we received tremendous support from the Network in Canadian

History and Environment in our early days, we're proud to say, we've been able to build the project without an active budget. Digital Humanities projects have a reputation for being expensive. The *Programming Historian* was built with passion, pitched on a whim, and has grown to become one of the field's most important publications. It's our attempt to share skills, build a field, and break some moulds.

We hope you will find a new skill within these pages. If we have missed what you're looking for, please get in touch and contribute a tutorial so that those who come after you can benefit from what you have learned. We're building this together, for each other.

Acknowledgments

Thank you to everyone who made this project possible. To William J. Turkel and Alan MacEachern for writing the first rendition of the *Programming Historian* and supporting its subsequent re-envisioning.

Thank you to our former editors, Jeremy Boggs and Carrie Sanders. To our current editors, Adam Crymble, Fred Gibbs, Allison Hegel, Caleb McDaniel, Ian Milligan, Miriam Posner, Evan Taparata, and Jeri Wieringa. Also to their friends and families and they who provide support while we volunteer our energy to this cause.

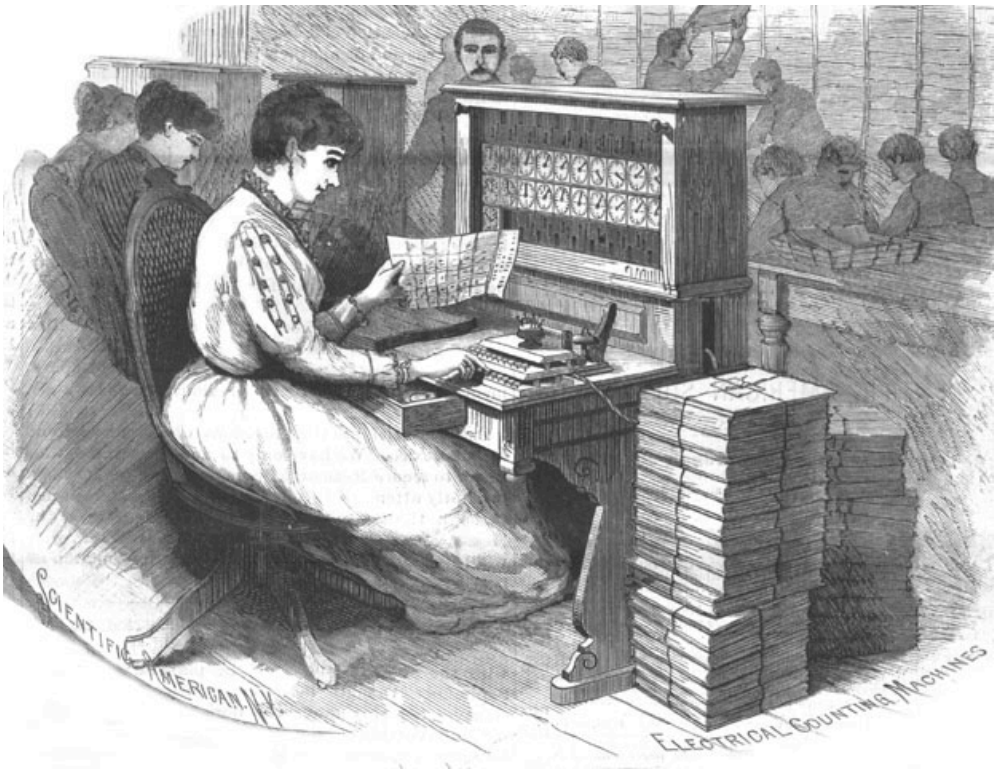
Thank you to our authors Kellen Kurschinski, Seth Bernstein, James Baker, Amanda Morton, Dennis Tenen, Grant Wythoff, Shawn Graham, Scott Weingart, Jim Clifford, Daniel Macfarlane, Josh MacFadyen, Jon Crump, Vilja Hulden, Marten Düring, Max De Wilde, Ruben Verborgh, Seth van Hooland, Jonathan Reeve, Spencer Roberts, Doug Knox, Heather Froehlich, Laura Turner O'Hara, Matthew Lincoln, Sarah Simpkin, Adam Crymble, Fred Gibbs, Miriam Posner, Jeri Wieringa, Ian Milligan, and Caleb McDaniel.

Thank you to our reviewers Matthew Lincoln, Daniel van Strien, Andrew Dunning, Robert Blades, Mila Oiva, Amanda Morton, Taryn Dewar, James Baker, Jonathan Reeve, Jim Clifford, Jacob Greene, Luke Bergmann, Kim Pham, John Fink, Thomas J. Duck, Patrick Burns, M. H. Beals, William J Turkel, Eduardo Sanchez, John Laudun, Tim Hitchcock, Konrad Lawson, Nora McGregor, Dennis Tenen, Mary Beth Start, Finn Arne Jørgensen, Jeff Bain-Conkin, Nick Pearce, Eli Pousson, Sarah Simpkin, Aurélien Berra, Chris Forster, Nick Ruest, Dave Shepard, Silvia Gutiérrez, Spencer Roberts, Sheila Brennan, Sara Palmer, John Levin, Amanda Visconti, Eli Pousson, Andrew Dunning, Tod Robbins, Jon Crump, Mauro Bieg, and Benjamin Schmidt.

Thank you to the many people who contributed openly and anonymously to our discussions about improving the accessibility of the project, including Heather Froehlich, Mia Ridge, Molly Desjardin, Scott Weingart, Fiona Tweedie, Matthew Lincoln, James Baker, Rebecca Sutton Koeser, Shawn Graham, Nora McGregor, M.H. Beals, Andromeda Yelton, Erin Bush, Daniel van Strein, Patrick Murray-John, Amanda Visconti, Eric Phetteplace, and Jessica Lord.

And thank you to the 200,000 readers of the *Programming Historian* around the world.

Introduction



The Programming Historian offers novice-friendly, peer-reviewed tutorials that help humanists learn a wide range of digital tools, techniques, and workflows to facilitate their research.

We regularly publish new lessons, and we always welcome proposals for new lessons on any topic. Our editorial mentors will be happy to work with you throughout the lesson writing process. If you'd like to be a reviewer or if you have suggestions to make Programming Historian a more useful resource, please see the Contribute page on our website.

Our editors and peer reviewers work collaboratively with authors to craft tutorials that illustrate fundamental digital and programming principles and techniques. We have lessons on Acquiring Data, Transforming Data, Analyzing Data, Presenting Data, Sustaining Data, in addition to a suite of lessons introducing the Python programming language.

If you can't find what you're looking for, we welcome your online feedback. Better yet, get in touch via our website and contribute a lesson! *The Programming Historian* (ISSN 2397-2068) aims to set a new standard for openness and collaboration in scholarly publishing, and you can help!

Open Source

The Programming Historian is committed to open source and open access principles. All contributed lessons must make use of open source programming languages and open source software whenever possible. This policy is meant to minimize costs for all parties, and to allow the greatest possible level of participation. We believe everyone should be able to benefit from these tutorials, not just those with large research budgets for expensive proprietary software.

Gold Open Access

All submissions to *The Programming Historian* are published under a Creative Commons 'BY' license. This adheres to a 'Gold' open access model of publishing, which is fully compliant with RCUK funding and HEFCE publishing requirements for scholars in the UK,[2] as well as the Canadian Tri-Agency Open Access Policy.[3] 'Gold' open access means that the version of record is made freely available without subscription fee or restrictions on access. Authors are permitted to republish their tutorials anywhere. And so can anyone, as long as they cite the original author and respect his or her moral rights.

We do not charge Article Processing Charges (APCs), nor do we charge library subscriptions.

Peer Review

All tutorials that appear on *The Programming Historian* have been rigorously peer reviewed and copy edited. Each lesson is guided through the review process by one of our editors who are assigned to the piece. Review involves a thorough exchange with the editor to ensure the lesson works as intended and that all concepts are explained fully for a non-specialist reader, before the tutorial is sent to external reviewers to test it and provide further comments. We aim to return reviewed material to authors quickly, but our first priority is always to ensure a quality product.

Our peer review process is a bit different from what might be considered the 'traditional' peer review process. We do not solicit reviews to judge whether a tutorial is 'good enough' to be published. Rather, we consider the review process an integral component of a collaborative, productive, and sustainable effort for scholars to create useful technical resources for each other. Once a tutorial slips into our editorial workflow, our goal is to do

[2] 'Open Access', *Research Councils UK* (2014): http://www.rcuk.ac.uk/research/openaccess/; 'Open Access Research', *Higher Education Funding Council for England*: http://www.hefce.ac.uk/rsrch/oa/

[3] 'Tri-Agency Open Access Policy on Publications', *Science.gc.ca* (2015): http://www.science.gc.ca/default.asp?lang=En&n=F6765465-1.

everything we can to make sure the tutorial becomes as useful as possible and published in a reasonable amount of time.

Once the peer review has commenced, the role of the editor is to mediate between reviewers and authors, and keep the process on track in a timely manner. We strive to ensure all tutorials are functional on their date of publication. From time to time technology changes and tutorials cease to function as intended. If this happens, please report it using the ‘Give Feedback’ link in the website footer, and we will assign an editor to fix the problem.

Funding & Ownership

The Programming Historian is a volunteer-led initiative, controlled entirely by the 'Editorial Board of the Programming Historian’ with the help of community contributors. It is not a legal entity, and does not currently receive direct funding from any source.

The project is grateful for past support by the Network in Canadian History & Environment (NiCHE), and for hosting support from the Roy Rosenzweig Centre for New Media (RRCHNM). If you would like to provide financial support to help the project grow, please contact one of the Editorial Board members listed on the website.

This project is our attempt to demonstrate what open access academic publishing can and should be. Please tell your librarian to include the project in your library catalogue.

Part One: Setting Up

The lessons in this first section are about setting up your computer for digital history work. You do not need to install these programs, but some of the lessons in the rest of the volume do require the software described herein. You can either start off by getting completely set up, or you can do it as and when you need to.

1. Python Introduction and Installation

William J. Turkel and Adam Crymble – 2012

Lesson Goals

This first lesson in our section on dealing with Online Sources is designed to get you and your computer set up to start programming. We will focus on installing the relevant software – all free and reputable – and finally we will help you to get your toes wet with some simple programming that provides immediate results.

In this opening module you will install the Python programming language,[4] the Beautiful Soup HTML/XML parser,[5] and a text editor. Screencaps provided here come from Komodo Edit,[6] but you can use any text editor capable of working with Python. Here's a list of other options: Python Editors.[7] Once everything is installed, you will write your first programs, "Hello World" in Python and HTML.

the Python Programming Language

The first programming language we will introduce in the Programming Historian is Python, a free, open source language. Unless otherwise noted, we will be using **Python v.2** throughout. Version 3 is available but we have elected to stick with version 2 for now because it's the most widely used version and it is the one that ships preinstalled on new Macs. Python 3 has different syntax (think grammar rules) and if you are trying to use Python 3 with the Programming Historian, you may run into difficulties. We welcome version 3 translations of any of our lessons.

Backup Your Work!

Before you download or install any new software, it is crucial that you make backups of your work. Each day before you do any programming, make sure to back up your work. At the end of a day's work, make another backup of any programs that you've written that day. You should back up your whole computer at least weekly, and preferably more frequently. It is also a good idea to make off-site backups of your work, so that you don't lose everything if something happens to your computer or to your home or

[4] 'Python': https://www.python.org/

[5] 'Beautiful Soup': http://www.crummy.com/software/BeautifulSoup/

[6] 'Komodo Edit': http://komodoide.com/komodo-edit/

[7] 'Python Editors', *Python*: https://wiki.python.org/moin/PythonEditors/

office. Sites like Jungle Disk and Dropbox provide easy-to-use and relatively inexpensive online backup options.[8]

Choose Your Operating System

Step 1 – Install and Set Up Software

In order to work through the techniques in this website, you will need to download and install some freely available software. We have provided instructions for Mac, Windows and Linux in the following three lessons. Once you have installed the software for your operating system, move on to 'Understanding Web Pages and HTML'.[9] If you run into trouble with our instructions or find something that doesn't work on your platform, please let us know.

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[8] 'Jungle Disk': https://www.jungledisk.com/; 'Dropbox': http://dropbox.com/.

[9] William J. Turkel and Adam Crymble, 'Understanding Web Pages and HTML' *The Programming Historian* (2012).

2. Setting Up an Integrated Development Environment for Python (Mac)

William J. Turkel and Adam Crymble – 2012

Back up your computer

Mac users can take advantage of the Time Machine for this.[10]

Install Python v.2

As of May 2012, Mac OS X comes preinstalled with Python 2. You can check to see if you have Python installed by launching the Terminal in the `'Applications/Utilities'` directory and entering `which python` followed by the Enter key. Pushing the Enter key sends the command to the computer when using the terminal. If you see `'/usr/bin/python'` or something similar containing the word 'python' and a bunch of slashes, then you are all set. If not, close the Terminal, download the latest stable release of the Python programming language (Version 2.7.3 as of May 2012) and install it by following the instructions on the Python website.[11]

Create a Directory

To stay organized, it's best to have a dedicated directory (folder) on your computer where you will keep your Python programs (e.g., `programming-historian`) and save it anywhere you like on your hard drive.

Beautiful Soup

Download the latest version of Beautiful Soup and copy it to the directory where you are going to put your own programs.[12] Beautiful Soup is a library (a collection of prewritten code) that makes it easy for Python programs to break web pages down into meaningful chunks that can be further processed.

Install Komodo Edit

Komodo Edit is a free and open source code editor, but as we said in the introduction, you have many other text editing options.[13] Some of our testers prefer a program called TextWrangler.[14] Which you use is up to you,

[10] 'Use Time Machine to back up or restore your Mac' *Apple Support*: https://support.apple.com/en-gb/HT201250

[11] 'Python': https://www.python.org/

[12] 'Beautiful Soup': http://www.crummy.com/software/BeautifulSoup/

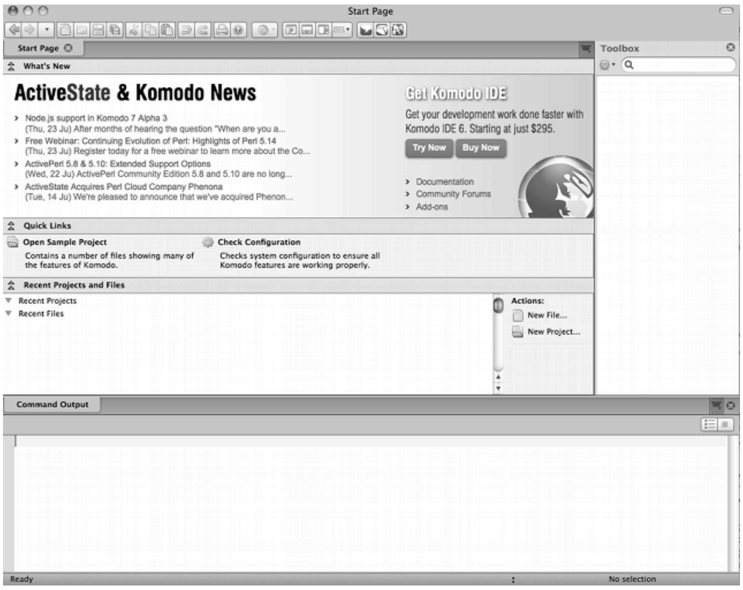
[13] 'Python Editors' *Python*: https://wiki.python.org/moin/PythonEditors/

[14] 'TextWrangler': http://www.barebones.com/products/textwrangler/

but for the sake of consistency in our lessons, we will be using Komodo Edit. You can download a copy of Komodo Edit from the Komodo Edit website.[15] Install it from the `.DMG` file

Start Komodo Edit

It should look something like this:



- Node.js support in Komodo 7 Alpha 3 (Thu, 23 Ju) After months of hearing the question "When are you a...
- Free Webinar: Continuing Evolution of Perl: Highlights of Perl 5.14 (Thu, 23 Ju) Register today for a free webinar to learn more about the Co...
- ActivePerl 5.8 & 5.10: Extended Support Options (Wed, 22 Ju) ActivePerl Community Edition 5.8 and 5.10 are no long...
- ActiveState Acquires Perl Cloud Company Phenona (Tue, 14 Ju) We're pleased to announce that we've acquired Phenon...

Get your development work done faster with Komodo IDE 6. Starting at just $295.

Contains a number of files showing many of the features of Komodo.

Checks system configuration to ensure all Komodo features are working properly.

screenshot of Komodo Exit on OS X

If you don't see the Toolbox pane on the right hand side, choose `View->Tabs & Sidebars ->Toolbox`. It doesn't matter if the Project pane is open or not. Take some time to familiarize yourself with the layout of the Komodo editor. The Help file is quite good

Configure Komodo Edit

Now you need to set up the editor so that you can run Python programs. In the Toolbox window, click on the gear icon and select "`New Command…`". This will open a new dialog window. Rename your command to "`Run Python`" and feel free to change the icon if you like. In the "`Command`" box, type

```
%(python) %f
```

and under "Start in," enter

```
%D
```

Click OK. Your new Run Python command should appear in the Toolbox pane.

[15] 'Komodo Edit': http://komodoide.com/komodo-edit/

Step 2 – "Hello World" in Python

It is traditional to begin programming in a new language by trying to create a program that says 'hello world' and terminates. We will show you how to do this in Python and HTML.

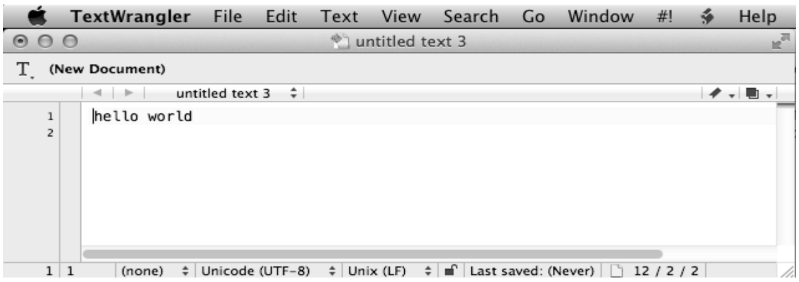
Python is a good programming language for beginners because it is very high-level. It is possible, in other words, to write short programs that accomplish a lot. The shorter the program, the more likely it is for the whole thing to fit on one screen, and the easier it is to keep track of all of it in your mind.

The languages that we will be using are all interpreted. This means that there is a special computer program (known as an interpreter) that knows how to follow instructions written in that language. One way to use the interpreter is to store all of your instructions in a file, and then run the interpreter on the file. A file that contains programming language instructions is known as a program. The interpreter will execute each of the instructions that you gave it in your program and then stop. Let's try this.

In your text editor, create a new file, enter the following two-line program and save it to your `programming-historian` directory as `hello-world.py`

```
# hello-world.py
print 'hello world'
```

Your chosen text editor should have a "`Run`" button that will allow you to execute your program. If you are using TextWrangler, click on the "#!" button and Run. If all went well, it should look something like this:



TextWrangler File Edit Text View Search Go Window #! Help
untitled text 3
(New Document)
untitled text 3
1 1 (none) Unicode (UTF-8) Unix (LF) Last saved: (Never) 12 / 2 / 2

```
hello world
```

TextWrangler-hello-world

Interacting with a Python shell

Another way to interact with an interpreter is to use what is known as a shell. You can type in a statement and press the Enter key, and the shell will respond to your command. Using a shell is a great way to test statements to make sure that they do what you think they should. This is done slightly differently on Mac, Linux and Windows.

You can run a Python shell by launching the "terminal". On the Mac, open the Finder and double-click on `Applications -> Utilities -> Terminal` then typing “python” into the window that opens on your screen. At the Python shell prompt, type

```
print 'hello world'
```

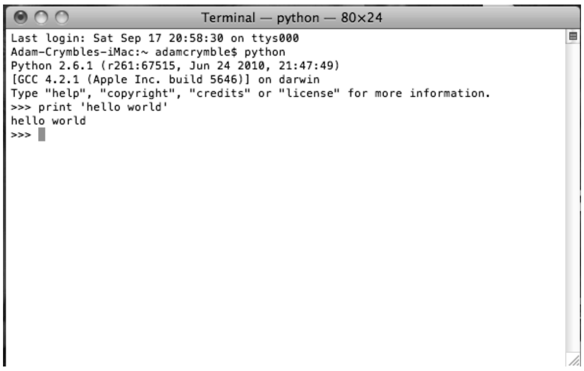
and press Enter. The computer will respond with

```
hello world
```

When we want to represent an interaction with the shell, we will use `->` to indicate the shell’s response to your command, as shown below:

```
print 'hello world'
-> hello world
```

On your screen, it will look more like this:



```
Last login: Sat Sep 17 20:58:30 on ttys000
Adam-Crymbles-iMac:~ adamcrymble$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>> 
```

hello world terminal on a Mac

Now that you and your computer are up and running, we can move onto some more interesting tasks. If you are working through the Python lessons in order, we suggest you next try ‘Understanding Webb Pages and HTML’.[16]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[16] William J. Turkel and Adam Crymble, ‘Understanding Web Pages and HTML’ *The Programming Historian* (2012).

3. Setting up an Integrated Development Environment for Python (Linux)

William J. Turkel and Adam Crymble – 2012

Thanks to John Fink for providing the basis of this section. These instructions are for Ubuntu 12.04 LTS, but should work for any apt based system such as Debian, or Linux Mint, provided you have sudo installed.

Back up your computer

It is always important to make sure you have regular and recent backups of your computer. This is just good advice for life, and is not limited to times when you are engaged in programming.

Install Python v. 2 and Python "Beautiful Soup" module

Open a terminal (`Dash Home`, then type `Terminal`, then click on the Terminal icon).

Now type: `sudo apt-get install python2.7 python-beautifulsoup`

Enter your password, and then type `Y` to finish the install. Note that you probably have Python 2.7 installed already, so don't be alarmed if Ubuntu tells you that.

Create a directory

You will keep your Python programs in this directory. It can be anywhere you like, but it is probably best to put it in your home folder. Something like this in your open terminal window should do the trick:

```
cd ~
mkdir programming-historian
```

Install Komodo Edit

Komodo Edit is a free and open source code editor, but as we said in the introduction, you have many other text editing options.[17] You can download Komodo Edit at the Komoto Edit Website.[18] Once you've downloaded it, open it with Ubuntu's package manager, extract it to your home directory, and follow the installation instructions. If you are following along with these instructions and have installed Komodo Edit, open the home folder, go to the `Komodo-Edit-7/bin` directory, and click on komodo. You can also

[17] 'PythonEditors' *Python:* https://wiki.python.org/moin/PythonEditors/

[18] 'Komodo Edit': http://komodoide.com/komodo-edit/

right click on the Komodo icon in your launcher and click “`Lock to Launcher`” to have Komodo saved permanently to your launcher bar.

Make a “Run Python” Command in Komodo Edit

In Komodo Edit, click the gear icon under `Toolbox` and select `New Command`.

In the top field type “`Run Python File`“

In the Command field, type: `%(python) %F` Then hit the OK button at the bottom of the Add Command window.

Step 2 – “Hello World” in Python

It is traditional to begin programming in a new language by trying to create a program that says “hello world” and terminates. We will show you how to do this in Python and HTML.

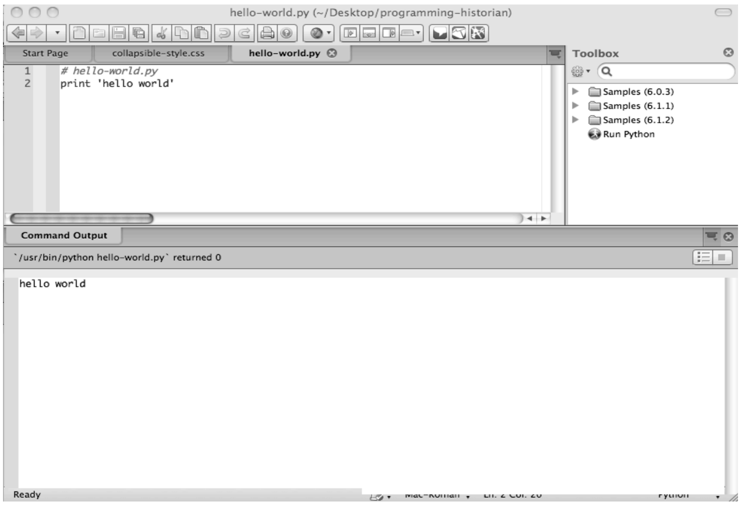
Python is a good programming language for beginners because it is very high-level. It is possible, in other words, to write short programs that accomplish a lot. The shorter the program, the more likely it is for the whole thing to fit on one screen, and the easier it is to keep track of all of it in your mind.

The languages that we will be using are all interpreted. This means that there is a special computer program (known as an interpreter) that knows how to follow instructions written in that language. One way to use the interpreter is to store all of your instructions in a file, and then run the interpreter on the file. A file that contains programming language instructions is known as a program. The interpreter will execute each of the instructions that you gave it in your program and then stop. Let’s try this.

In your text editor, create a new file, enter the following two-line program and save it to your `programming-historian` directory as `hello-world.py`

```
# hello-world.py
print 'hello world'
```

Your chosen text editor should have a “`Run`” button that will allow you to execute your program. If all went well, it should look something like this (Example as seen in Komodo Edit. Click on the image to see a full-size copy):



```
# hello-world.py
print 'hello world'
```

```
`/usr/bin/python hello-world.py` returned 0
```

```
hello world
```

hello world in Komodo Edit on a Mac

Interacting with a Python shell

Another way to interact with an interpreter is to use what is known as a shell. You can type in a statement and press the Enter key, and the shell will respond to your command. Using a shell is a great way to test statements to make sure that they do what you think they should.

You can run a Python shell by launching the “terminal”. For Linux, go to `Applications-> Accessories -> Terminal`and do the same. At the Python shell prompt, type

```
python
```

This will open up the Python prompt, meaning that you can now use Python commands in the shell. Now type

```
print 'hello world'
```

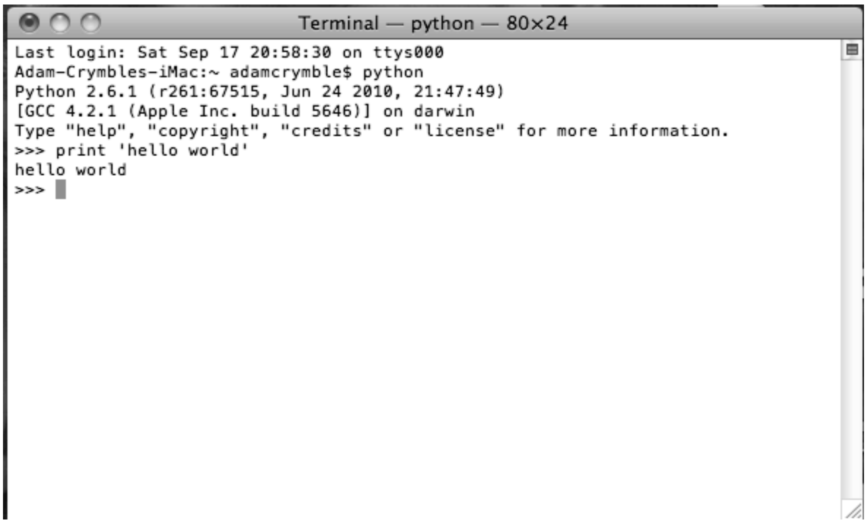
and press Enter. The computer will respond with

```
hello world
```

When we want to represent an interaction with the shell, we will use `->` to indicate the shell’s response to your command, as shown below:

```
print 'hello world'
-> hello world
```

On your screen, it will look more like this:



```
Last login: Sat Sep 17 20:58:30 on ttys000
Adam-Crymbles-iMac:~ adamcrymble$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>> 
```

hello world terminal on a Mac

Now that you and your computer are up and running, we can move onto some more interesting tasks. If you are working through the Python lessons in order, we suggest you next try 'Understanding Web Pages and HTML'.[19]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[19] William J. Turkel and Adam Crymble, 'Understanding Web Pages and HTML' *The Programming Historian* (2012).

4. Setting Up an Integrated Development Environment for Python (Windows)

William J. Turkel and Adam Crymble – 2012

Back up your computer

It is always important to make sure you have regular and recent backups of your computer. This is just good advice for life, and is not limited to times when you are engaged in programming.

Install Python v.2

Go to the Python website, download the latest stable release of the Python programming language (Version 2.7.3 as of May 2012) and install it by following the instructions on the Python website.[20]

Create a Directory

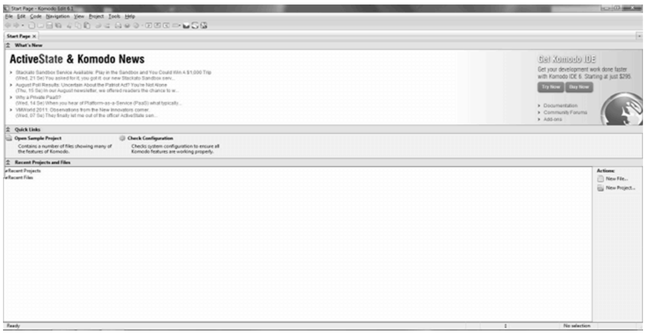
To stay organized, it's best to have a dedicated directory (folder) on your computer where you will keep your Python programs (e.g., `programming-historian`) and save it anywhere you like on your hard drive.

Install Komodo Edit

Komodo Edit is a free and open source code editor, but as we said in the introduction, you have many other text editing options.[21] You can download a copy from the Komodo Edit website.[22]

Start Komodo Edit

It should look something like this:



ActiveState & Komodo News

- Stackato Sandbox Service Available: Play in the Sandbox and You Could Win A $1,000 Trip (Wed, 21 Se) You asked for it, you got it: our new Stackato Sandbox serv...
- August Poll Results: Uncertain About the Patriot Act? You're Not Alone (Thu, 15 Se) In our August newsletter, we offered readers the chance to w...
- Why a Private PaaS? (Wed, 14 Se) When you hear of Platform-as-a-Service (PaaS) what typically...
- VMWorld 2011: Observations from the New Innovators corner (Wed, 07 Se) They finally let me out of the offical ActiveState san...

Get your development work done faster with Komodo IDE 6. Starting at just $295.

Open Sample Project: Contains a number of files showing many of the features of Komodo.

Check Configuration: Checks system configuration to ensure all Komodo features are working properly.

Komodo Edit on Windows

[20] 'Python': https://www.python.org/

[21] 'PythonEditors' *Python: https://wiki.python.org/moin/PythonEditors/*

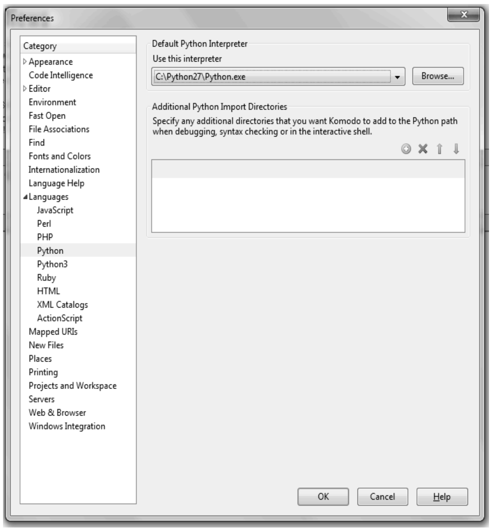
[22] 'Komodo Edit': http://komodoide.com/komodo-edit/

If you don’t see the Toolbox pane on the right hand side, choose `View -> Tabs -> Toolbox`. It doesn’t matter if the Project pane is open or not. Take some time to familiarize yourself with the layout of the Komodo editor. The Help file is quite good

Configure Komodo Edit

Now you need to set up the editor so that you can run Python programs.

1. Choose `Edit -> Preferences`. This will open a new dialog window. Select the Python category and set the “`Default Python Interpreter`” (it should be `C:\Python27\Python.exe`)
   If it looks like this, click OK:



Default Python Interpreter

Use this interpreter: C:\Python27\Python.exe

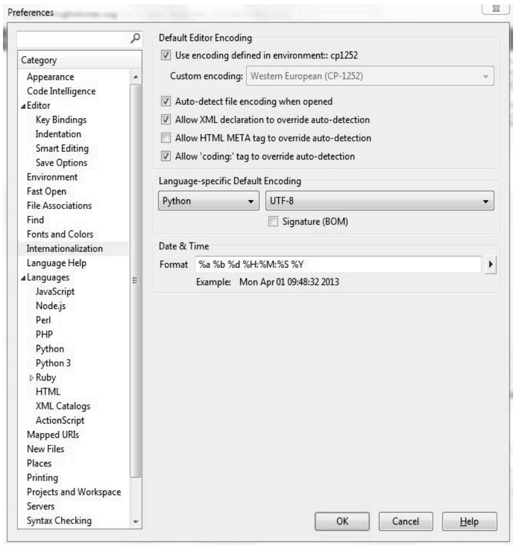
Additional Python Import Directories

Specify any additional directories that you want Komodo to add to the Python path when debugging, syntax checking or in the interactive shell.

Set the Default Python Interpreter

2. Next, in the Preferences section select *Internationalization*. Select *Python* from the drop-down menu titled *Language-specific Default Encoding* and make sure that UTF-8 is selected as the default encoding method.[23]

[23] ‘UTF-8’, *Wikipedia:* https://en.wikipedia.org/wiki/UTF-8



Default Editor Encoding

☑ Use encoding defined in environment:: cp1252

Custom encoding: Western European (CP-1252)

☑ Auto-detect file encoding when opened

☑ Allow XML declaration to override auto-detection

□ Allow HTML META tag to override auto-detection

☑ Allow 'coding:' tag to override auto-detection

Language-specific Default Encoding

Python UTF-8

□ Signature (BOM)

Date & Time

Format %a %b %d %H:%M:%S %Y

Example: Mon Apr 01 09:48:32 2013

Set the Language to UTF-8

Next choose `Toolbox->Add->New Command`. This will open a new dialog window. Rename your command to `'Run Python'`. Under `'Command'`, type:

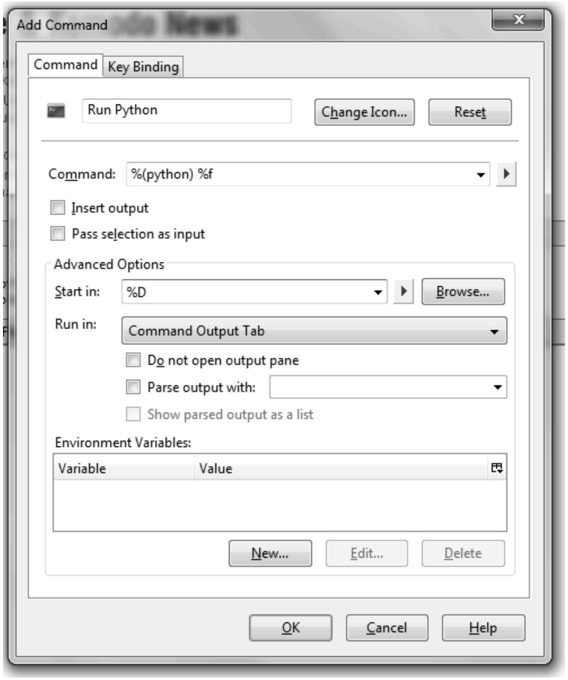
```
%(python) %f
```

If you forget this command, Python will hang mysteriously because it isn't receiving a program as input.

Under `'Start in'`, enter:

```
%D
```

If it looks like this, click OK:



Run Python

Command: %(python) %f

□ Insert output

□ Pass selection as input

Advanced Options

Start in: %D

Run in: Command Output Tab

□ Do not open output pane

□ Parse output with:

□ Show parsed output as a list

Environment Variables:

Variable	Value

Run Python Command Windows

“Run Python” Command

Your new command should appear in the Toolbox pane. You may need to restart your machine after completing this step before Python will work with Komodo Edit

Step 2 – “Hello World” in Python

It is traditional to begin programming in a new language by trying to create a program that says “hello world” and terminates. We will show you how to do this in Python and HTML.

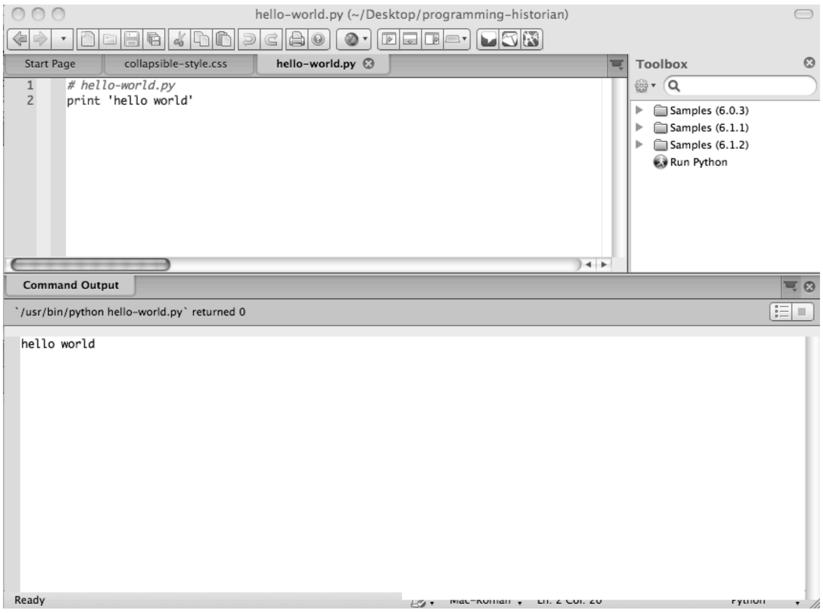
Python is a good programming language for beginners because it is very high-level. It is possible, in other words, to write short programs that accomplish a lot. The shorter the program, the more likely it is for the whole thing to fit on one screen, and the easier it is to keep track of all of it in your mind.

The languages that we will be using are all interpreted. This means that there is a special computer program (known as an interpreter) that knows how to follow instructions written in that language. One way to use the interpreter is to store all of your instructions in a file, and then run the interpreter on the file. A file that contains programming language instructions is known as a program. The interpreter will execute each of the instructions that you gave it in your program and then stop. Let’s try this.

In your text editor, create a new file, enter the following two-line program and save it to your `programming-historian` directory as `hello-world.py`

```
# hello-world.py
print 'hello world'
```

Your chosen text editor should have a “`Run`” button that will allow you to execute your program. If all went well, it should look something like this (Example as seen in Komodo Edit. Click on the image to see a full-size copy):



```
# hello-world.py
print 'hello world'
```

```
`/usr/bin/python hello-world.py` returned 0
```

```
hello world
```

hello world in Komodo Edit

Interacting with a Python shell

Another way to interact with an interpreter is to use what is known as a shell. You can type in a statement and press the Enter key, and the shell will respond to your command. Using a shell is a great way to test statements to make sure that they do what you think they should.

You can run a Python Shell by double-clicking on the python.exe file. If you installed version 2.7 (the most recent as of May 2012), then this file is probably located in the `C:\Python27\python.exe` directory. In the shell window that opens on your screen type:

```
print 'hello world'
```

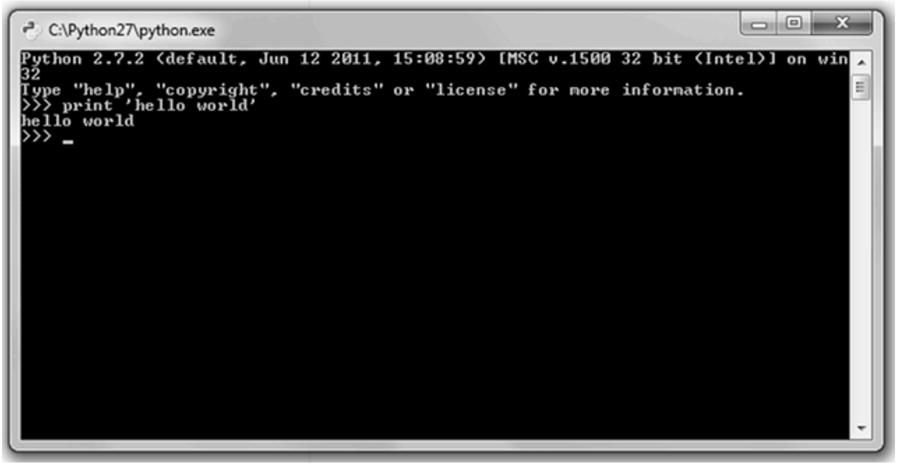
and press Enter. The computer will respond with

```
hello world
```

When we want to represent an interaction with the shell, we will use -> to indicate the shell's response to your command, as shown below:

```
print 'hello world'
-> hello world
```

On your screen, it will look more like this:



```
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>> _
```

Python Shell on Windows

Python Shell in Windows

Now that you and your computer are up and running, we can move onto some more interesting tasks. If you are working through the Python lessons in order, we suggest you next try 'Understanding Web Pages and HTML'.[24]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[24] William J. Turkel and Adam Crymble, 'Understanding Web Pages and HTML', *Programming Historian* (2012).

5. Installing Python Modules with pip

Fred Gibbs – 2013

Lesson Goals

This lesson shows you how to download and install Python modules. There are many ways to install external modules, but for the purposes of this lesson, we're going to use a program called pip.[25] As of Python 2.7.9 and newer, pip is installed by default. This tutorial will be helpful for anyone using older versions of Python (which are still quite common).

Introducing Modules

One of the great things about using Python is the number of fantastic code libraries that are widely and easily available that can save you a lot of coding, or simply make a particular task (like creating a CSV file, or scraping a webpage) much easier. When Googling for solutions to problems, you'll often find sample code that uses code libraries you haven't heard about before. Don't let these scare you away! Once these libraries are installed on your computer, you can use them by importing them at the beginning of your code; you can import as many libraries as you'd like, such as

```
import csv
import requests
import kmlwriter
import pprint
```

For new Python users, it can be a bit intimidating to download and install external modules for the first time. There are many ways of doing it (thus adding to the confusion); this lesson introduces one of the easiest and most common ways of installing python modules.

The goal here is to install software on your computer that can automatically download and install Python modules for us. We're going to use a program called pip.

Note: As of Python 3.4, pip will be included in the regular install. There are many reasons why you might not have this version yet, and in case you don't, these instructions should help.

Mac and Linux instructions

As per the pip documentation, we can download a python script to install pip for us. Using a Mac or Linux, we can install pip via the command line

[25] 'pip': https://pip.pypa.io/en/stable/

by using the curl command, which downloads the pip installation perl script.[26]

```
curl https://bootstrap.pypa.io/get-pip.py
```

once you've downloaded the get-pip.py file, you need to execute it with the python interpreter. However, if you try to execute the script with python like

```
python get-pip.py
```

the script will most likely fail because it won't have permissions to update certain directories on your filesystem that are by default set so that random scripts cannot change important files and give you viruses. In this case—and in all cases where you need to allow a script that you trust to write to your system folders—you can use the `sudo` command (short for "Super User DO") in front of the python command, like

```
sudo python get-pip.py
```

Windows Instructions

As with the above platforms, the easiest way to install pip is through the use of a python program called get-pip.py, which you can download at https://bootstrap.pypa.io/get-pip.py. When you open this link, you might be scared of the massive jumble of code that awaits you. Please don't be. Simply use your browser to save this page under its default name, which is get-pip.py. It might be a good idea to save this file in your python directory, so you know where to find it.

Once you have saved this file, you need to run it, which can be done in two ways. If you prefer using your python interpreter, just right-click on the file get-pip.py and choose "open with" and then choose whatever python interpreter you care to use.

If you prefer to install pip using the windows command line, navigate to whatever directory you've placed python and get-pip.py. For this example, we'll assume this directory is python27, so we'll use the command C:\>cd python27. Once you are in this directory, run the command

```
python get-pip.py to install pip
```

If you are looking for more information, check out the StackOverflow page that seems to be regularly updated.[27]

[26] Lakshmanan Ganapathy, '15 Practical Linux curl Command Examples (curl Download Examples)' *The Geek Stuff* (11 April 2012): http://www.thegeekstuff.com/2012/04/curl-examples/

[27] 'How do I install pip on Windows?' *Stack Overflow*: http://stackoverflow.com/questions/4750806/how-do-i-install-pip-on-windows

Installing Python Modules

Now that you have pip, it is easy to install python modules since it does all the work for you. When you find a module that you want to use, usually the documentation or installation instructions will include the necessary pip command, such as

```
pip install requests
pip install beautifulsoup4
pip install simplekml
```

Remember, for the same reasons explained above, you will probably need to run pip with sudo, like

```
sudo pip install requests
```

Happy installing!

About the Author

Fred Gibbs is an assistant professor of history at the University of New Mexico.

Part Two: Acquiring Data

After your machine is set up, often the first stage of a digital humanities project involves getting the data – the *stuff* – that you will be working with. Increasingly the web is littered with juicy data that could become the basis of a great project. Having the skills to acquire that data efficiently can save you tremendous amounts of time. The seven lessons in this first section provide various ways of extracting targeted information from the web and saving it to your computer.

6. Downloading Web Pages with Python

William J. Turkel and Adam Crymble – 2012

Editor’s Note: This lesson was originally written as part of a series of introduction to Python lessons. You may find it easier to complete this lesson if you have already competed the previous lesson in this series: ‘Code Reuse and Modularity in Python’.[28]

Lesson Goals

This lesson introduces *Uniform Resource Locators* (URLs) and explains how to use Python to download and save the contents of a web page to your local hard drive.

About URLs

A *web page* is a file that is stored on another computer, a machine known as a *web server*. When you "go to" a web page, what is actually happening is that your computer, the *client*, sends a request to the server (the *host*) out over the network, and the server replies by sending a copy of the page back to your machine. One way to get to a web page with your browser is to follow a link from somewhere else. You also have the ability, of course, to paste or type a Uniform Resource Locator (URL) directly into your browser. The URL tells your browser where to find an online resource by specifying the server, directory and name of the file to be retrieved, as well as the kind of *protocol* that the server and your browser will agree to use while exchanging information (like HTTP, the *Hypertext Transfer Protocol*). The basic structure of a URL is

```
protocol://host:port/path?query
```

Let's look at a few examples.

```
http://oldbaileyonline.org
```

The most basic kind of URL simply specifies the protocol and host. If you give this URL to your browser, it will return the main page of The Old

[28] William J. Turkel and Adam Crymble, ‘Code Reuse and Modularity in Python’ *The Programming Historian* (2012).

Bailey Online website.[29] The default assumption is that the main page in a given directory will be named index, usually `index.html`.

The URL can also include an optional *port number*. Without getting into too much detail at this point, the network protocol that underlies the exchange of information on the Internet allows computers to connect in different ways. Port numbers are used to distinguish these different kinds of connection. Since the default port for HTTP is 80, the following URL is equivalent to the previous one.

```
http://oldbaileyonline.org:80
```

As you know, there are usually many web pages on a given website. These are stored in directories on the server, and you can specify the path to a particular page. The "About" page for *The Old Bailey Online* has the following URL.

```
http://oldbaileyonline.org/static/Project.jsp
```

Finally, some web pages allow you to enter queries. *The Old Bailey Online* website, for example, is laid out in such a way that you can request a particular page within it by using a *query string*. The following URL will take you to a search results page for criminal record trials containing the word "arsenic".

```
http://www.oldbaileyonline.org/search.jsp?form=custom&_divs_fulltext=arsenic
```

The snippet after the "?" represents the query. You can learn more about building queries in Downloading Multiple Records Using Query Strings.[30]

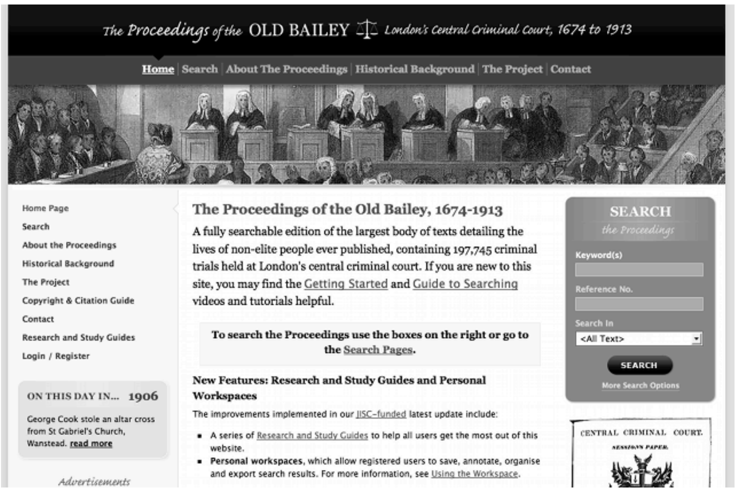
Opening URLs with Python

As a digital historian you will often find yourself wanting to use data held in scholarly databases online. To get this data you could open URLs one at a time and copy and paste their contents to a text file, or you can use Python to automatically harvest and process webpages. To do this, you're going to need to be able to open URLs with your own programs. The Python language includes a number of standard ways to do this.

As an example, let's work with the kind of file that you might encounter while doing historical research. Say you're interested in race relations in eighteenth century Britain. The Old Bailey Online (OBO) is a rich resource that provides trial transcripts from 1674 to 1913 and is one good place to seek sources.

[29] Tim Hitchcock, Robert Shoemaker, Clive Emsley, Sharon Howard and Jamie McLaughlin, *et al.*, *The Old Bailey Proceedings Online, 1674-1913* (www.oldbaileyonline.org, version 7.0, 24 March 2012).

[30] Adam Crymble, 'Downloading Multiple Records Using Query Strings', *The Programming Historian* (2012).



ON THIS DAY IN... 1906

George Cook stole an altar cross from St Gabriel's Church, Wanstead. read more

The Proceedings of the Old Bailey, 1674-1913

A fully searchable edition of the largest body of texts detailing the lives of non-elite people ever published, containing 197,745 criminal trials held at London's central criminal court. If you are new to this site, you may find the Getting Started and Guide to Searching videos and tutorials helpful.

To search the Proceedings use the boxes on the right or go to the Search Pages.

New Features: Research and Study Guides and Personal Workspaces

The improvements implemented in our JISC-funded latest update include:

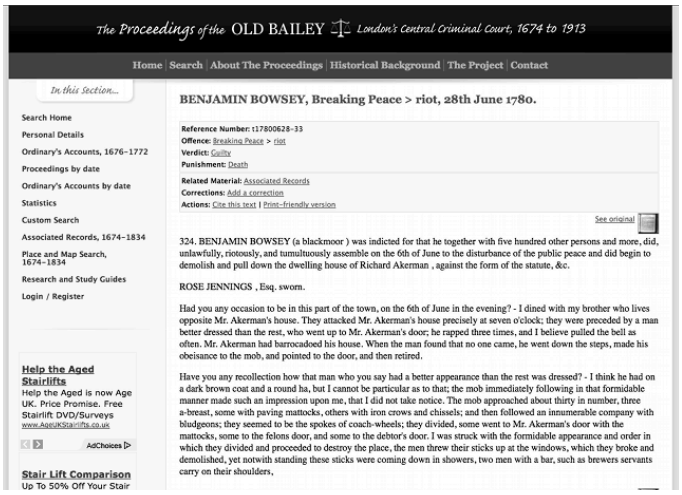
- A series of Research and Study Guides to help all users get the most out of this website.
- **Personal workspaces**, which allow registered users to save, annotate, organise and export search results. For more information, see Using the Workspace.

The Old Bailey Online Homepage

For this example, we will be using the trial transcript of Benjamin Bowsey, a "black moor" who was convicted of breaking the peace during the Gordon Riots of 1780.[31] The URL for the entry is

```
http://www.oldbaileyonline.org/browse.jsp?id=t17800628-33&div=t17800628-33
```

By studying the URL we can learn a few things. First, The OBO is written in JSP (*JavaServer Pages*, a web programming language which outputs HTML), and it's possible to retrieve individual trial entries by making use of the query string. Each is apparently given a unique ID number (*id=t* in the URL), built from the date of the trial session in the format (*YYYYMMDD*) and the trial number from within that court session, in this case: *33*. If you change the two instances of `33` to `34` in your browser and press Enter, you should be taken to the next trial. Unfortunately, not all websites have such readable and reliable URLs.


The Proceedings of the OLD BAILEY London's Central Criminal Court, 1674 to 1913
Home | Search | About The Proceedings | Historical Background | The Project | Contact
In this Section...
Search Home
Personal Details
Ordinary's Accounts, 1676-1772
Proceedings by date
Ordinary's Accounts by date
Statistics
Custom Search
Associated Records, 1674-1834
Place and Map Search, 1674-1834
Research and Study Guides
Login / Register
Help the Aged Stairlifts
Help the Aged is now Age UK. Price Promise. Free Stairlift DVD/Surveys
www.AgeUKStairlifts.co.uk
AdChoices
Stair Lift Comparison
Up To 50% Off Your Stair
See original

BENJAMIN BOWSEY, Breaking Peace > riot, 28th June 1780.

Reference Number: t17800628-33
Offence: Breaking Peace > riot
Verdict: Guilty
Punishment: Death

Related Material: Associated Records
Corrections: Add a correction
Actions: Cite this text | Print-friendly version

324. BENJAMIN BOWSEY (a blackmoor) was indicted for that he together with five hundred other persons and more, did, unlawfully, riotously, and tumultuously assemble on the 6th of June to the disturbance of the public peace and did begin to demolish and pull down the dwelling house of Richard Akerman , against the form of the statute, &c.

ROSE JENNINGS , Esq. sworn.

Had you any occasion to be in this part of the town, on the 6th of June in the evening? - I dined with my brother who lives opposite Mr. Akerman's house. They attacked Mr. Akerman's house precisely at seven o'clock; they were preceded by a man better dressed than the rest, who went up to Mr. Akerman's door; he rapped three times, and I believe pulled the bell as often. Mr. Akerman had barrocadoed his house. When the man found that no one came, he went down the steps, made his obeisance to the mob, and pointed to the door, and then retired.

Have you any recollection how that man who you say had a better appearance than the rest was dressed? - I think he had on a dark brown coat and a round ha, but I cannot be particular as to that; the mob immediately following in that formidable manner made such an impression upon me, that I did not take notice. The mob approached about thirty in number, three a-breast, some with paving mattocks, others with iron crows and chissels; and then followed an innumerable company with bludgeons; they seemed to be the spokes of coach-wheels; they divided, some went to Mr. Akerman's door with the mattocks, some to the felons door, and some to the debtor's door. I was struck with the formidable appearance and order in which they divided and proceeded to destroy the place, the men threw their sticks up at the windows, which they broke and demolished, yet notwith standing these sticks were coming down in showers, two men with a bar, such as brewers servants carry on their shoulders,

Trial Account of Benjamin Bowsey, 1780

[31] 'Gordon Riots' *Wikipedia*: https://en.wikipedia.org/wiki/Gordon_Riots

Spend a few minutes looking at Benjamin Bowsey's trial page. Here we are not so much interested in what the transcript says, but what features the page has. Notice the "View as XML" link at the bottom that takes you to a heavily marked up version of the text which may be useful to certain types of research. You can look at a scan of the original document, which was transcribed to make this resource.[32] And you can access a "Print-friendly version" by clicking a link near the top of the entry.

When you are processing web resources automatically, it is often a good idea to work with printable versions if you can, as they tend to have less formatting. Since we have that option, we will use the printable version in this lesson.

```
http://www.oldbaileyonline.org/print.jsp?div=t17800628-33
```

Now let's try opening the page using Python. Copy the following program into Komodo Edit and save it as `open-webpage.py`. When you execute the program, it will `open` the trial file, `read` its contents into a Python string called webContent and then `print` the first three hundred characters of the string to the "Command Output" pane. Use the `View -> Web Developer -> View Page Source` command in Firefox to verify that the HTML source of the page is the same as the source that your program retrieved. Each browser has a different shortcut key to open the page source. In Firefox on PC it is `CTRL+u`. If you cannot find it on your browser, try using a search engine to find where it is. (See the Python library reference to learn more about urllib2.)[33]

```
# open-webpage.py

import urllib2

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
webContent = response.read()

print webContent[0:300]
```

These five lines of code achieve an awful lot very quickly. Let us take a moment to make sure that everything is clear and that you can recognize the building blocks that allow us to make this program do what we want it to do.

url, *response*, and *webContent* are all variables that we have named ourselves.

[32] 'Page Image: Proceedings of the Old Bailey, 28th June 1780, page 84.' *Old Bailey Online*: http://www.oldbaileyonline.org/images.jsp?doc=178006280084

[33] 'urllib2' *Python*: https://docs.python.org/2/library/urllib2.html

url holds the URL of the web page that we want to download. In this case, it is the trial of Benjamin Bowsey.

On the following line, we call the function `urlopen`, which is stored in a Python module named `urllib2.py`, and we have asked that function to open the website found at the URL we just specified. We then saved the result of that process into a variable named *response*. That variable now contains an open version of the requested website.

We then use the `read` method, which we used earlier, to copy the contents of that open webpage into a new variable named *webContent*.

Make sure you can pick out the variables (there are 3 of them), the modules (1), the methods (2), and the parameters (1) before you move on.

In the resulting output, you will notice a little bit of HTML markup:

```
 <!-- MAIN CONTENT -->

<div id="main" class="full"><div id="main2">
        <div style="font-family:serif;"><i>Old Bailey Proceedings Online</i>
(www.oldbaileyonline.org, version 6.0, 16 March 2013),
June 1780, trial of
BENJAMIN                       BOWSEY
```

Because we are using the printable version there is a lot less than most web pages have, but there is still more than we need. Don't worry; you will soon learn how to remove that excess markup.

Saving a Local Copy of a Web Page

Given what you already know about writing to files, it is quite easy to modify the above program so that it writes the contents of the *webContent* string to a local file on our computer rather than to the "Command Output" pane. Copy the following program into Komodo Edit, save it as `save-webpage.py` and execute it. Using the `File -> Open File` command in Firefox, open the file on your hard drive that it creates (`obo-t17800628-33.html`) to confirm that your saved copy is the same as the online copy.

```
# save-webpage.py

import urllib2

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
webContent = response.read()

f = open('obo-t17800628-33.html', 'w')
f.write(webContent)
f.close
```

So, if you can save a single file this easily, could you write a program to download a bunch of files? Could you step through trial IDs, for example, and make your own copies of a whole bunch of them? Yep. We'll get there soon.

Suggested Readings

Lutz, Mark. "Ch. 4: Introducing Python Object Types", *Learning Python* (O'Reilly, 1999).

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each lesson you can download the "programming-historian" zip file to make sure you have the correct code.

programming-historian-1 (zip):
programminghistorian.org/assets/programming-historian1.zip

If you are following the learning Python lessons, the next tutorial in this series is 'Manipulating Strings in Python'.[34]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[34] William J. Turkel and Adam Crymble, 'Manipulating Strings in Python', *The Programming Historian* (2012).

7. Automated Downloading with Wget

Ian Milligan – 2012

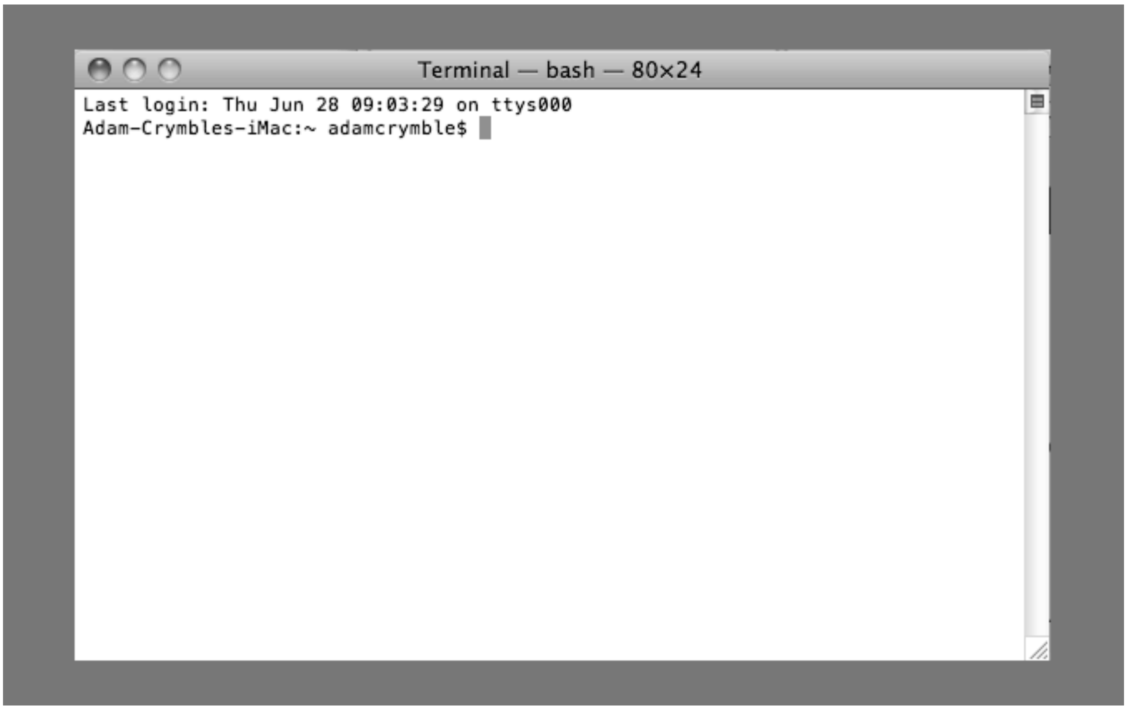
Editor's Note

This lesson requires you to use the command line. If you have no previous experience using the command line you may find it helpful to work through the Scholar's Lab Command Line Bootcamp tutorial.[35]

Lesson Goals

This is a lesson designed for intermediate users, although beginner users should be able to follow along.

Wget is a useful program, run through your computer's command line, for retrieving online material.



```
Last login: Thu Jun 28 09:03:29 on ttys000
Adam-Crymbles-iMac:~ adamcrymble$ 
```

The Mac Command Line, Terminal

It can be useful in the following situations:

Retrieving or mirroring (creating an exact copy of) an entire website. This website might contain historical documents, or it may simply be your own

[35] 'The Command Line' *The Praxis Program at the Scholar's Lab*: http://praxis.scholarslab.org/scratchpad/bash/

personal website that you want to back up. One command can download the entire site onto your computer.
Downloading specific files in a website's hierarchy (all websites within a certain part of a website, such as every page that is contained within the `/papers/` directory of a website).

In this lesson, we will work through three quick examples of how you might use wget in your own work. At the end of the lesson, you will be able to quickly download large amounts of information from the Internet in an automated fashion. If you find a repository of online historical information, instead of right-clicking on every file and saving it to build your dataset, you will have the skills to craft a single command to do so.

First, a caution is in order. You need to be careful about how you use wget. If you consult the manual when in doubt, and work through the lessons here, you should be okay. You should always build a delay into your commands so that you do not overload the servers, and should also always put a limit on the speed to which you download. This is all part of being a good Internet citizen, and can be seen as analogous to sipping from a firehose rather than turning it on all at once (it's not good for you, or the water company).

Be as specific as possible when formulating your download. One joke suggests that you can accidentally download the entire Internet with wget. While that's a bit of an exaggeration, it isn't too far off!

Let's begin.

Step One: Installation

Linux Instructions

If you are using a Linux system, you should already have wget installed. To check if you have it, open up your command line. Type `'wget'` and press enter. If you have wget installed the system will respond with:

```
-> Missing URL.
```

If you do not have wget installed, it will respond with

```
-> command not found.
```

If you are on OS X or Windows, you will need to download the program. If on Linux, you receive the error message indicating that you do not have wget installed, follow the OS X instructions below.

OS X Instructions

OS X Option One: The Preferred Method

On OS X, there are two ways to get wget and install it. The easiest is to install a package manager and use it to automatically install wget. There is a second method, discussed below, that involves compiling it.

Both, however, require that you install Apple's 'Command Line Tools' to use properly. This requires downloading XCode. If you have the 'App Store', you should be able to just download XCode.[36] If not, the following instructions will work.

To download this, go to the Apple Developer website,[37] register as a developer, and then in the downloads for Apple developers section you will need to find the correct version.

It is a big download, and will take some time. Once you have the file, install it.

You will need to install the '**Command Line Tools**' kit in XCode. Open up the 'Preferences' tab, click on 'Downloads,' and then click 'Install' next to Command Line Tools. We are now ready to install a package manager.

The easiest package manager to install is *Homebrew*. Go to http://mxcl.github.io/homebrew/ and review the instructions. There are many important commands, like wget, that are not included by default in OS X. This program facilitates the downloading and installation of all required files.

To install *Homebrew*, open up your terminal window and type the following:

```
ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
```

This uses the ruby programming language, built into OS X, to install Homebrew. To see if the installation worked, type the following into your terminal window:

```
brew
```

A list of documentation options should appear if it has been installed. We have one more command to run to make sure everything is working, which is:

```
brew doctor
```

With *Homebrew* installed, we now have to install wget. This is now an easy step.

[36] 'Xcode' *App Store*: https://itunes.apple.com/en/app/xcode/id497799835?mt=12

[37] 'Xcode' *Apple Developer*: https://developer.apple.com/xcode/

```
brew install wget
```

It will proceed to download the most recent version of wget, which is wget 1.14. After the script stops running, and you are back to your main window, enter the following command into the terminal:

```
wget
```

If you have installed it, you will see:

```
-> Missing URL.
```

If not, you will see:

```
-> command not found.
```

At this point, you should have installed wget successfully. We are now ready to keep going!

OS X Option Two

If for some reason you do not want to install a package manager, you are able to simply download wget alone. This will be applicable if you are using a different packet manager (such as Mac Ports) or if you want to keep your infrastructure to a minimum. Follow the same instructions again to install xcode and the Command Line Tools set.

Then you can subsequently download an uncompiled version of wget from the GNU website[38] (I chose to download the file 'wget-1.13.tar.gz', which you can find by following the link to either the HTTP or FTP[39] download pages), unzip it (by double-clicking on it) into your home directory (on a Mac, this will be your `/user/` directory – for example, my user name is ianmilligan and it appears next to a house icon in my Finder), and then open up Terminal. For this tutorial, we have downloaded `wget-1.13`.

First, we will need to navigate to the directory that the wget files are in. At the terminal, type:

```
cd wget-1.13
```

Note that if you have downloaded a different version of wget, the following steps will work but you may have to replace the above version number (i.e. `1.13`) with your own.

We now need to generate the instructions, or makefile, for the file. This is sort of a blueprint for what the final file is going to look like. Accordingly, type:

```
./configure –with-ssl=openssl
```

[38] 'GNU Wget' *GNU Operating System*: http://www.gnu.org/software/wget/

[39] 'GNU Wget – ftp *GNU Operating System: http://ftp.gnu.org/gnu/wget/; OR ftp://ftp.gnu.org/gnu/wget/*

Now that we have the blueprints, let\'s tell our computer to follow them. Type:

```
make
```

Then, you need to make the final file. By pre-pending the command sudo, you are running the command with highest security privileges. This lets you actually install the file into your system.

```
sudo make install
```

At this point, you will be prompted for your computer's password. Type it.

You should now have wget installed.

Windows Instructions

The easiest way is to download a working version. To do so, visit the eternallybored website[40] and, download `wget.exe` (as of writing it is version 1.16.3, and you should download the 32-bit binary). If you place `wget.exe` in your `C:Windows` directory, you can then use wget from anywhere on your computer. This will make your life easier as you will not have to worry about always running wget from only one place on your system. If it is in this directory, Windows will know that the command can be used anywhere in your terminal window.

Step Two: Learning about the Structure of Wget – Downloading a Specific Set of Files

At this point, users of all three platforms should be on the same page. We use wget through our operating system's command line interface (introduced previously as `Terminal` for Mac and Linux users, where you have been playing around with some Python commands). You need to use your command line, instead of the Komodo Edit client you may have used in other lessons.

The comprehensive documentation for wget can be found on the GNU wget manual page.[41]

Let's take an example dataset. Say you wanted to download all of the papers hosted on the website ActiveHistory.ca. They are all located at: http://activehistory.ca/papers/; in the sense that they are all contained within the `/papers/` directory: for example, the 9th paper published on the website is http://activehistory.ca/papers/historypaper-9/. Think of this structure in the same way as directories on your own computer: if you have a folder labeled `/History/`, it likely contains several files within it. The same structure holds true for websites, and we are using this logic to tell our computer what files we want to download.

[40] 'Windows binaries of GNU Wget' *Eternally Bored*: https://eternallybored.org/misc/wget/

[41] 'GNU Wget 1.17.1 Manual' *GNU*: http://www.gnu.org/software/wget/manual/wget.html

If you wanted to download them all manually, you would either need to write a custom program, or right-click every single paper to do so. If the files are organized in a way that fits your research needs, wget is the quickest approach.

To make sure wget is working, try the following.

In your working directory, make a new directory. Let's call it `wget-activehistory`. You can make this using your Finder/Windows, or if you are at a Terminal window at that path, you can type:

```
mkdir wget-activehistory
```

Either way, you now have a directory that we will be working in. Now open up your command line interface and navigate to the `wget-activehistory` directory. As a reminder, you can type:

```
cd [directory]
```

to navigate to a given directory. If you've made this directory in your home directory, you should be able to type `cd wget-activehistory` to move to your new directory.

Enter the following command:

```
wget http://activehistory.ca/papers/
```

After some initial messages, you should see the following (figures, dates and some details will be different, however):

```
Saving to: `index.html.1'

[] 37,668 --.-K/s in 0.1s

2012-05-15 15:50:26 (374 KB/s) - `index.html.1' saved [37668]
```

What you have done is downloaded just the first page of http://activehistory.ca/papers/, the index page for the papers to your new directory. If you open it, you'll see the main text on the home page of ActiveHistory.ca. So at a glance, we have already quickly downloaded something.

What we want to do now, however, is to download every paper. So we need to add a few commands to wget.

Wget operates on the following general basis:

```
wget [options] [URL]
```

We have just learned about the [URL] component in the previous example, as it tells the program where to go. Options, however, give the program a bit more information about what exactly we want to do. The program knows that an option is an option by the presence of a dash before the variable. This lets it know the difference between the URL and the options. So let's learn a few commands now:

```
-r
```

Recursive retrieval is the most important part of wget. What this means is that the program begins following links from the website and downloading them too. So for example, the http://activehistory.ca/papers/ has a link to http://activehistory.ca/papers/historypaper-9/, so it will download that too if we use recursive retrieval. However, it will also follow any other links: if there was a link to http://uwo.ca somewhere on that page, it would follow that and download it as well. By default, -r sends wget to a depth of five sites after the first one. This is following links, to a limit of five clicks after the first website. At this point, it will be quite indiscriminate. So we need more commands:

```
--no-parent
```

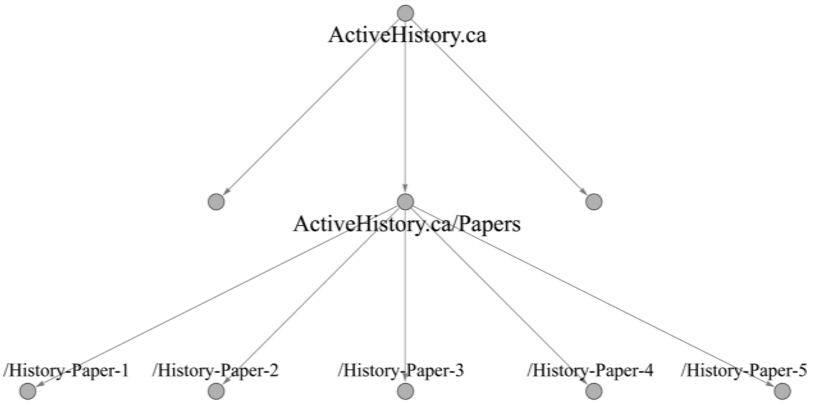
(The double-dash indicates the full-text of a command. All commands also have a short version, this could be initiated using -np).

This is an important one. What this means is that wget should follow links, but not beyond the last parent directory. In our case, that means that it won't go anywhere that is not part of the http://activehistory.ca/papers/ hierarchy. If it was a long path such as:

http://niche-canada.org/projects/events/new-events/not-yet-happened-events/

It would only find files in the `/not-yet-happened-events/` folder. It is a critical command for delineating your search.

Here is a graphical representation:



A graphical representation of how 'no-parent' works with wget

Finally, if you do want to go outside of a hierarchy, it is best to be specific about how far you want to go. The default is to follow each link and carry on to a limit of five pages away from the first page you provide. However, perhaps you just want to follow one link and stop there? In that case, you could input `-l 2`, which takes us to a depth of two web-pages. Note this is a lower-case 'L', not a number 1.

```
-l 2
```

If these commands help direct wget, we also need to add a few more to be nice to servers and to stop any automated countermeasures from thinking the server is under attack! To that end, we have two additional essential commands:

```
-w 10
```

It is not polite to ask for too much at once from a web server. There are other people waiting for information, too, and it is thus important to share the load. The command `-w 10`, then, adds a ten second wait in between server requests. You can shorten this, as ten seconds is quite long. In my own searches, I often use a 2 second wait. On rare occasions, you may come across a site that blocks automated downloading altogether. The website's terms of service, which you should consult, may not mention a policy on automated downloading, but steps to prohibit it may be built into their website's architecture nonetheless. In such rare cases, you can use the command `--random-wait` which will vary the wait by 0.5 and 1.5 times the value you provide here.

Another critical comment is to limit the bandwidth you will be using in the download:

```
--limit-rate=20k
```

This is another important, polite command. You don't want to use up too much of the servers' bandwidth. So this command will limit the maximum download speed to 20kb/s. Opinion varies on what a good limit rate is, but you are probably good up to about 200kb/s for small files – however, not to tax the server, let us keep it at 20k. This will also keep us at `ActiveHistory.ca` happy!

Step Three: Mirror an Entire Website

Ok, with all of this, let's finally download all of the ActiveHistory.ca papers. Note that the trailing slash on the URL is critical – if you omit it, wget will think that papers is a file rather than a directory. Directories end in slashes. Files do not. The command will then download the entire ActiveHistory.ca page. The order of the options does not matter.

```
wget -r --no-parent -w 2 --limit-rate=20k http://activehistory.ca/papers/
```

It will be slower than before, but your terminal will begin downloading all of the ActiveHistory.ca papers. When it is done, you should have a directory labeled `ActiveHistory.ca` that contains the `/papers/` sub-directory –

perfectly mirrored on your system. This directory will appear in the location that you ran the command from in your command line, so likely is in your `USER` directory. Links will be replaced with internal links to the other pages you've downloaded, so you can actually have a fully working ActiveHistory.ca site on your computer. This lets you start to play with it without worrying about your internet speed.

To see if the download was a success, you will also have a log in your command screen. Take a look over it to make sure that all files were downloaded successfully. If it did not download, it will let you know that it failed.

If you want to mirror an entire website, there is a built-in command to wget.

```
-m
```

This command means 'mirror,' and is especially useful for backing up an entire website. It introduces the following set of commands: time-stamping, which looks at the date of the site and doesn't replace it if you already have that version on your system (useful for repeated downloads), as well as infinite recursion (it will go as many layers into the site as necessary). The command for mirroring ActiveHistory.ca would be:

```
wget -m -w 2 --limit-rate=20k http://activehistory.ca
```

A Flexible Tool for Downloading Internet Sources

As you become increasingly comfortable with the command line, you'll find wget a helpful addition to your digital toolkit. If there is an entire set of archival documents that you want to download for text mining, if they're arranged in a directory and are all together (which is not as common as one might think), a quick wget command will be quicker than scraping the links with Python. Similarly, you can then begin downloading things directly from your command line: programs, files, backups, etc.

Further Reading

I've only given a snapshot of some of wget's functionalities. For more, please visit the wget manual.[42]

About the Author

Ian Milligan is an assistant professor of history at the University of Waterloo.

[42] 'GNU Wget 1.17.1 Manual' *GNU*: http://www.gnu.org/software/wget/manual/wget.html

8. Applied Archival Downloading with Wget

Kellen Kurchinski – 2013

Editor's Note: you may find it easier to complete this lesson if you have already completed the previous 'wget' tutorial by Ian Milligan.

Background and Lesson Goals

Now that you have learned how Wget can be used to mirror or download specific files from websites like ActiveHistory.ca via the command line, it's time to expand your web-scraping skills through a few more lessons that focus on other uses for Wget's recursive retrieval function. The following tutorial provides three examples of how Wget can be used to download large collections of documents from archival websites with assistance from the Python programing language. It will teach you how to parse and generate a list of URLs using a simple Python script, and will also introduce you to a few of Wget's other useful features. Similar functions to the ones demonstrated in this lesson can be achieved using curl,[43] an open-source software capable of performing automated downloads from the command line. For this lesson, however, we will focus on Wget and building your Python skills.

Archival websites offer a wealth of resources to historians, but increased accessibility does not always translate into increased utility. In other words, while online collections often allow historians to access hitherto unavailable or cost-prohibitive materials, they can also be limited by the manner in which content is presented and organized. Take for example the Indian Affairs Annual Reports database[44] hosted on the Library and Archives Canada [LAC] website. Say you wanted to download an entire report, or reports for several decades. The current system allows a user the option to read a plaintext version of each page, or click on the "View a scanned page of original Report" link, which will take the user to a page with LAC's embedded image viewer. This allows you to see the original document, but it is also cumbersome because it requires you to scroll through each individual page. Moreover, if you want the document for offline viewing, the only option is to *right click –> save as* each image to a

[43] Konrad M. Lawson, 'Download a Sequential Range of URLs with Curl', *ProfHacker* (2012): http://chronicle.com/blogs/profhacker/download-a-sequential-range-of-urls-with-curl/41055

[44] 'Indian Affairs Annual Reports, 1864-1990' *Library and Archives Canada*: http://www.bac-lac.gc.ca/eng/discover/aboriginal-heritage/first-nations/indian-affairs-annual-reports/Pages/introduction.aspx

directory on your computer. If you want several decades' worth of annual reports, you can see the limits to the current means of presentation pretty easily. This lesson will allow you to overcome such an obstacle.

Recursive Retrieval and Sequential URLs: The Library and Archives Canada Example

Let's get started. The first step involves building a script to generate sequential URLs using Python's ForLoop function. First, you'll need to identify the beginning URL in the series of documents that you want to download. Because of its smaller size we're going to use the online war diary for No. 14 Canadian General Hospital as our example.[45] The entire war diary is 80 pages long. The URL for page 1 is http://data2.archives.ca/e/e061/e001518029.jpg and the URL for page 80 is http://data2.archives.ca/e/e061/e001518109.jpg. Note that they are in sequential order. We want to download the .jpeg images for *all* of the pages in the diary. To do this, we need to design a script to generate all of the URLs for the pages in between (and including) the first and last page of the diary.

Open your preferred text editor (such as Komodo Edit)[46] and enter the code below. Where it says 'integer 1′ type in '8029′, where it says 'integer 2′, type '8110'. The For Loop will generate a list of numbers between '8029' and '8110', but it will not print the last number in the range (i.e. 8110). To download all 80 pages in the diary you must add one to the top-value of the range because it is at this integer where the ForLoop is told to stop. This applies for any sequence of numbers you generate with this function. Additionally, the script will not properly execute if leading zeros[47] are included in the range of integers, so you must exclude them by leaving them in the string (the URL). In this example I have parsed the URL so that only the last four digits of the string are being manipulated by the ForLoop.

```
#URL-Generator.py

urls = '';
f=open('urls.txt','w')
for x in range('integer1', 'integer2'):
    urls = 'http://data2.collectionscanada.ca/e/e061/e00151%d.jpg\n' % (x)
    f.write(urls)
f.close
```

Now replace 'integer1′ and 'integer2′ with the bottom and top ranges of URLs you want to download. The final product should look like this:

[45] 'War diaries – 14th Canadian General Hospital' *Library and Archives Canada*: http://collectionscanada.gc.ca/pam_archives/index.php?fuseaction=genitem.displayItem&lang=eng&rec_nbr=2005110&rec_nbr_list=3366167,3203123,2005097,2005100,2005101,2005099,2005096,2005110,2005108,2005106

[46] 'Komodo Edit' *Komodo IDE*: http://komodoide.com/komodo-edit/

[47] 'Leading zero' *Wikipedia*: https://en.wikipedia.org/wiki/Leading_zero

```
#URL-Generator.py

urls = '';
f=open('urls.txt','w')
for x in range(8029, 8110):
    urls = 'http://data2.collectionscanada.ca/e/e061/e00151%d.jpg\n' % (x)
    f.write(urls)
f.close
```

Save the program as a .py file, and then click run the Python script.

The ForLoop will automatically generate a sequential list of URLs between the range of two integers that you specified in the brackets, and will write them to a .txt file that will be saved in your Programming Historian directory. The `%d` appends each sequential number generated by the ForLoop to the exact position you place it in the string. Adding `\n` to the end of the string removes line-breaks, allowing Wget to read the .txt file.

You do not need to use all of the digits in the URL to specify the range – just the ones between the beginning and end of the sequence you are interested in. This is why only the last 4 digits of the string were selected and `00151` was left intact.

Before moving on to the next stage of the downloading process, make sure you have created a directory where you would like to save your files, and, for ease of use, locate it in the main directory where you keep your documents. For both Mac and Windows users this will normally be the 'Documents' folder. For this example, we'll call our folder 'LAC'. You should move the urls.txt file your Python script created in to this directory. To save time on future downloads, it is advisable to simply run the program from the directory you plan to download to. This can be achieved by saving the URL-Generator.py file to your 'LAC' folder.

For Mac users, under your applications list, select *Utilities -> Terminal*. For Windows Users, you will need to open your system's Command Line utility.

Once you have a shell open, you need to 'call' the directory you want to save your downloaded .jpeg files to. Type:

```
cd ~/Documents
```

and hit enter. Then type:

```
cd 'LAC'
```

and press enter again. You now have the directory selected and are ready to begin downloading.

Based on what you have learned from Ian Milligan's Wget lesson,[48] enter the following into the command line (note you can choose whatever you like

[48] Ian Milligan, 'Automated Downloading with Wget', *The Programming Historian* (2012).

for your 'limit rate', but be a responsible internet citizen and keep it under 200kb/s!):

```
wget -i urls.txt -r --no-parent -nd -w 2 --limit-rate=100k
```

(Note: including '-nd' in the command line will keep Wget from automatically mirroring the website's directories, making your files easier to access and organize).

Within a few moments you should have all 80 pages of the war diary downloaded to this directory. You can copy and move them into a new folder as you please.

A Second Example: The National Archives of Australia

Let's try one more example using this method of recursive retrieval. This lesson can be broadly applied to numerous archives, not just Canadian ones!

Say you wanted to download a manuscript from the National Archives of Australia, which has a much more aesthetically pleasing online viewer than LAC, but is still limited by only being able to scroll through one image at a time. We'll use William Bligh's "Notebook and List of Mutineers, 1789" which provides an account of the mutiny aboard the HMS *Bounty.*[49] On the viewer page (see previous footnote) you'll note that there are 131 'items' (pages) to the notebook. This is somewhat misleading. Click on the first thumbnail in the top right to view the whole page. Now, *right-click -> view image*. The URL should be 'http://nla.gov.au/nla.ms-ms5393-1-s1-v.jpg'. If you browse through the thumbnails, the last one is 'Part 127', which is located at 'http://nla.gov.au/nla.ms-ms5393-1-s127-v.jpg'. The discrepancy between the range of URLs and the total number of files means that you may miss a page or two in the automated download – in this case there are a few URLs that include a letter in the name of the .jpeg ('s126a.v.jpg' or 's126b.v.jpg' for example). This is going to happen from time to time when downloading from archives, so do not be surprised if you miss a page or two during an automated download.

Note that a potential workaround could include using regular expressions to make more complicated queries if appropriate (for more, see the Understanding Regular Expressions lesson by Doug Knox).[50]

Let's run the script and Wget command once more:

[49] 'MS 5393 Notbook and list of mutineers, 1789 [manuscript]', *Digital Collections Manuscripts:* http://www.nla.gov.au/apps/cdview/?pi=nla.ms-ms5393-1

[50] Doug Knox, 'Understanding Regular Expressions', *The Programming Historian* (2013).

```
#Bligh.py

urls = '';
f=open('urls.txt','w')
for x in range(1, 128):
    urls = 'http://www.nla.gov.au/apps/cdview/?pi=nla.ms-ms5393-1-s%d-v.jpg\
n' % (x)
    f.write(urls)
f.close
```

And:

```
wget -i urls.txt -r --no-parent -nd -w 2 --limit-rate=100k
```

You now have a (mostly) full copy of William Bligh's notebook. The missing pages can be downloaded manually using *right-click -> save image as*.

Recursive Retrieval and Wget's 'Accept' (-A) Function

Sometimes automated downloading requires working around coding barriers. It is common to encounter URLs that contain multiple sets of leading zeros, or URLs which may be too complex for someone with a limited background in coding to design a Python script for. Thankfully, Wget has a built-in function called 'Accept' (expressed as '-A') that allows you to define what type of files you would like to download from a specific webpage or an open directory.

For this example we will use one of the many great collections available through the Library of Congress website: The Thomas Jefferson Papers. As with LAC, the viewer for these files is outdated and requires you to navigate page by page. We're going to download a selection from Series 1: General Correspondence. 1651-1827.[51] Open the link and then click on the image (the .jpeg viewer looks awful familiar doesn't it?) The URL for the image also follows a similar pattern to the war diary from LAC that we downloaded earlier in the lesson, but the leading zeros complicate matters and do not permit us to easily generate URLs with the first script we used. Here's a workaround. Look at this webpage:

http://memory.loc.gov/master/mss/mtj/mtj1/001/0000/

The page you just opened is a sub-directory of the website that lists the .jpeg files for a selection of the Jefferson Papers. This means that we can use Wget's '–A' function to download all of the .jpeg images (100 of them) listed on that page. But say you want to go further and download the whole range of files for this set of dates in Series 1 – that's 1487 images. For a task like this where there are relatively few URLs you do not actually need

[51] 'The Thomas Jefferson Papers Series 1. General Correspondence. 1651-1827' *The Library of Congress: http://memory.loc.gov/cgi-bin/ampage?collId=mtj1&fileName=mtj1page001.db&recNum=1&itemLink=/ammem/collections/jefferson_papers/mtjser1.html&linkText=6*

to write a script (although you could using my final example, which discusses the problem of leading zeros). Instead, simply manipulate the URLs in a .txt file as follows:

http://memory.loc.gov/master/mss/mtj/mtj1/001/0000/

http://memory.loc.gov/master/mss/mtj/mtj1/001/0100/

http://memory.loc.gov/master/mss/mtj/mtj1/001/0200/

... all the way up to

http://memory.loc.gov/master/mss/mtj/mtj1/001/1400

This is the last sub-directory on the Library of Congress site for these dates in Series 1. This last URL contains images 1400-1487.

Your completed .txt file should have 15 URLs total. Before going any further, save the file as 'Jefferson.txt' in the directory you plan to store your downloaded files in.

Now, run the following Wget command:

```
wget –i Jefferson.txt –r --no-parent -nd –w 2 –A .jpg, .jpeg --limit-rate=10
0k
```

Voila, after a bit of waiting, you will have 1487 pages of presidential papers right at your fingertips!

More Complicated Recursive Retrieval: A Python Script for Leading Zeros

The Library of Congress, like many online repositories, organizes their collections using a numbering system that incorporates leading zeros within each URL. If the directory is open, Wget's –A function is a great way to get around this without having to do any coding. But what if the directory is closed and you can only access one image at a time? This final example will illustrate how to use a Python script to incorporate leading into a list of URLs. For this example we will be using the Historical Medical Poster Collection,[52] available from the Harvey Cushing/Jack Hay Whitney Medical Library (Yale University).

First, we'll need to identify the URL of the first and last files we want to download. We also want the high-resolution versions of each poster. To locate the URL for the high res image click on the first thumbnail (top left) then look below the poster for the link that says 'Click HERE for Full Image'. If you follow the link, a high-resolution image with a complex URL will appear. As was the case in the Australian Archives example, to get the

[52] 'Historical Medical Poster Collection', *Harvey Cushing/John Hay Whitney Medical Library:* http://cushing.med.yale.edu/gsdl/collect/mdposter/

simplified URL you must *right-click -> view image* using your web-browser. The URL for the first poster should be:

http://cushing.med.yale.edu/images/mdposter/full/poster0001.jpg

Follow the same steps for the last poster in the gallery – the URL should be:

http://cushing.med.yale.edu/images/mdposter/full/poster0637.jpg.

The script we used to download from LAC will not work because the range function cannot comprehend leading zeros. The script below provides an effective workaround that runs three different For Loops and exports the URLs to a .txt file in much the same way as our original script. This approach would also work with the Jefferson Papers, but I chose to use the –A function to demonstrate its utility and effectiveness as a less complicated alternative.

In this script the poster URL is treated in much the same way as the URL in our LAC example. The key difference is that the leading zeros are included as part of the string. For each loop, the number of zeros in the string decreases as the digits increase from single, to double, to triple. The script can be expanded or shortened as needed. In this case we needed to repeat the process three times because we were moving from three leading zeros to one leading zero. To ensure that the script iterates properly, a '+' should be added to each For Loop as in the example below.

We do not recommend actually performing this download because of the size and extent of the files. This example is merely intended to illustrate the how to build and execute the Python script.

```
#Leading-Zeros.py

urls = '';
f=open('leading-zeros.txt','w')

for x in range(1,10):
    urls += 'http://cushing.med.yale.edu/images/mdposter/full/poster000%d.jp
g\n' % (x)

for y in range(10,100):
    urls += 'http://cushing.med.yale.edu/images/mdposter/full/poster00%d.jpg
\n' % (y)

for z in range(100,638):
    urls += 'http://cushing.med.yale.edu/images/mdposter/full/poster0%d.jpg\
n' % (z)

f.write(urls)
f.close
```

Conclusion

These three examples only scratch the surface of Wget's potential. Digital archives organize, store, and present their content in a variety of ways, some of which are more accessible than others. Indeed, many digital repositories store files using URLs that must be manipulated in several different ways to utilize a program like Wget. Wherever your downloading may take you, new challenges and opportunities await. This tutorial has provided you with the core skills for further work in the digital archive and, hopefully, will lead you to undertake your own experiments in an effort to add new tools to the digital historian's toolkit. As new methods for scraping online repositories become available, we will continue to update this lesson with additional examples of Wget's power and potential.

About the Author

Kellen Kurchinski is a doctoral candidate in history at McMaster University and a Research Officer at the University of Waterloo.

9. Downloading Multiple Records Using Query Strings

Adam Crymble – 2012

Module Goals

Downloading a single record from a website is easy, but downloading many records at a time – an increasingly frequent need for a historian – is much more efficient using a programming language such as Python. In this lesson, we will write a program that will download a series of records from the Old Bailey Online[53] using custom search criteria, and save them to a directory on our computer. This process involves interpreting and manipulating URL *Query Strings*. In this case, the tutorial will seek to download sources that contain references to people of African descent that were published in the *Old Bailey Proceedings* between 1700 and 1750.

For Whom is this Useful?

Automating the process of downloading records from an online database will be useful for anyone who works with historical sources that are stored online in an orderly and accessible fashion and who wishes to save copies of those sources on their own computer. It is particularly useful for someone who wants to download many specific records, rather than just a handful. If you want to download *all* or *most* of the records in a particular database, you may find Ian Milligan's tutorial on 'Automated Downloading with WGET' more suitable.[54]

The present tutorial will allow you to download discriminately, isolating specific records that meet your needs. Downloading multiple sources automatically saves considerable time. What you do with the downloaded sources depends on your research goals. You may wish to create visualizations or perform various data analysis methods, or simply reformat them to make browsing easier. Or, you may just want to keep a backup copy so you can access them without Internet access.

This lesson is for intermediate Python users. If you have not already tried the Python Programming Basics lessons, you may find that a useful starting point.[55]

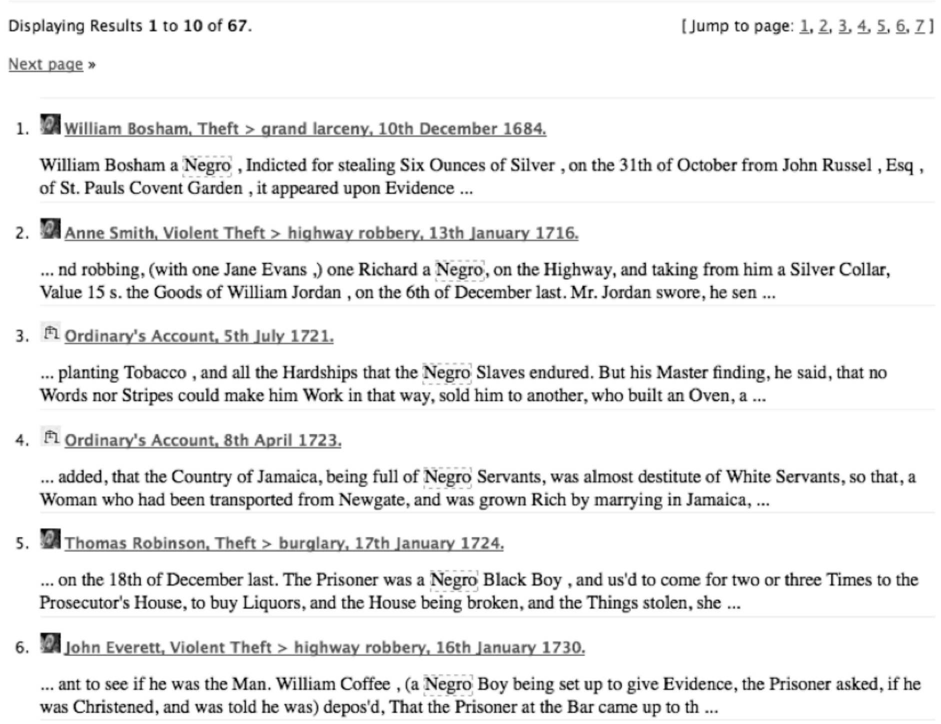
[53] Tim Hitchcock, Robert Shoemaker, Clive Emsley, Sharon Howard and Jamie McLaughlin, *et al.*, *The Old Bailey Proceedings Online, 1674-1913* (www.oldbaileyonline.org, version 7.0, 24 March 2012).

[54] Ian Milligan, 'Automated Downloading with Wget', *The Programming Historian* (2012).

[55] William J. Turkel & Adam Crymble, 'Python Introduction and Installation', *The Programming Historian* (2012).

Applying our Historical Knowledge

In this lesson, we are trying to create our own corpus of cases related to people of African descent. From Benjamin Bowsey's case[56] at the Old Bailey in 1780, we might note that "black" can be a useful keyword for us to use for locating other cases involving defendants of African descent. However, when we search for "black" on the Old Bailey website, we find it often refers to other uses of the word: black horses, or black cloth. The task of disambiguating this use of language will have to wait for another lesson. For now, let's turn to easier cases. As historians, we can probably think of keywords related to African descendants that would be worth pursuing. The infamous "n-word" of course is not useful, as this term did not come into regular usage until the mid-nineteenth century. "Negro" and "mulatto" are however, much more relevant to the early eighteenth century. These keywords are less ambiguous than "black" and are much more likely to be immediate references to people in our target demographic. If we try these two terms in separate simple searches on the Old Bailey website, we get results like in these screenshots:



Displaying Results **1** to **10** of **67**. [Jump to page: 1, 2, 3, 4, 5, 6, 7]

Next page »

1. William Bosham, Theft > grand larceny, 10th December 1684.

   William Bosham a Negro , Indicted for stealing Six Ounces of Silver , on the 31th of October from John Russel , Esq , of St. Pauls Covent Garden , it appeared upon Evidence ...

2. Anne Smith, Violent Theft > highway robbery, 13th January 1716.

   ... nd robbing, (with one Jane Evans ,) one Richard a Negro, on the Highway, and taking from him a Silver Collar, Value 15 s. the Goods of William Jordan , on the 6th of December last. Mr. Jordan swore, he sen ...

3. Ordinary's Account, 5th July 1721.

   ... planting Tobacco , and all the Hardships that the Negro Slaves endured. But his Master finding, he said, that no Words nor Stripes could make him Work in that way, sold him to another, who built an Oven, a ...

4. Ordinary's Account, 8th April 1723.

   ... added, that the Country of Jamaica, being full of Negro Servants, was almost destitute of White Servants, so that, a Woman who had been transported from Newgate, and was grown Rich by marrying in Jamaica, ...

5. Thomas Robinson, Theft > burglary, 17th January 1724.

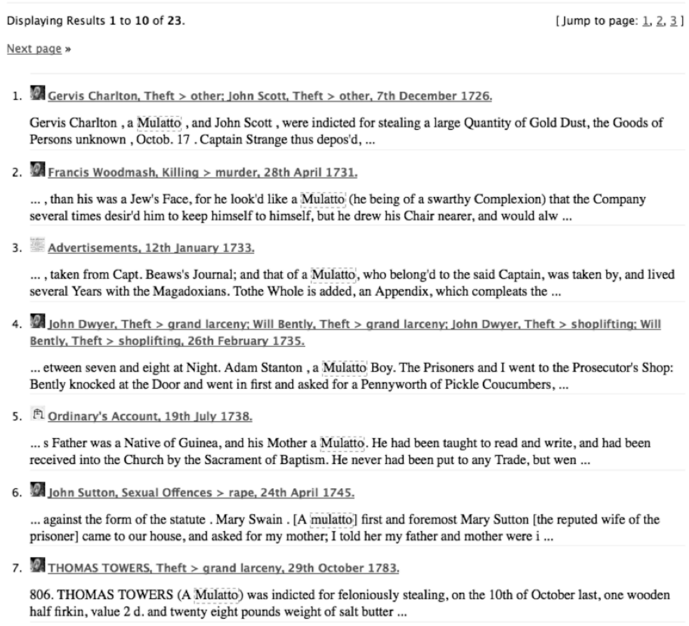
   ... on the 18th of December last. The Prisoner was a Negro Black Boy , and us'd to come for two or three Times to the Prosecutor's House, to buy Liquors, and the House being broken, and the Things stolen, she ...

6. John Everett, Violent Theft > highway robbery, 16th January 1730.

   ... ant to see if he was the Man. William Coffee , (a Negro Boy being set up to give Evidence, the Prisoner asked, if he was Christened, and was told he was) depos'd, That the Prisoner at the Bar came up to th ...

Search results for 'negro' in the Old Bailey Online

[56] *Old Bailey Proceedings Online* (www.oldbaileyonline.org, version 7.2, 23 February 2016), June 1780, trial of BENJAMIN BOWSEY: http://www.oldbaileyonline.org/browse.jsp?ref=t17800628-33.



Displaying Results **1** to **10** of **23**. [Jump to page: 1, 2, 3]

Next page »

1. Gervis Charlton, Theft > other; John Scott, Theft > other, 7th December 1726.

   Gervis Charlton , a Mulatto , and John Scott , were indicted for stealing a large Quantity of Gold Dust, the Goods of Persons unknown , Octob. 17 . Captain Strange thus depos'd, ...

2. Francis Woodmash, Killing > murder, 28th April 1731.

   ... , than his was a Jew's Face, for he look'd like a Mulatto (he being of a swarthy Complexion) that the Company several times desir'd him to keep himself to himself, but he drew his Chair nearer, and would alw ...

3. Advertisements, 12th January 1733.

   ... , taken from Capt. Beaws's Journal; and that of a Mulatto, who belong'd to the said Captain, was taken by, and lived several Years with the Magadoxians. Tothe Whole is added, an Appendix, which compleats the ...

4. John Dwyer, Theft > grand larceny; Will Bently, Theft > grand larceny; John Dwyer, Theft > shoplifting; Will Bently, Theft > shoplifting, 26th February 1735.

   ... etween seven and eight at Night. Adam Stanton , a Mulatto Boy. The Prisoners and I went to the Prosecutor's Shop: Bently knocked at the Door and went in first and asked for a Pennyworth of Pickle Coucumbers, ...

5. Ordinary's Account, 19th July 1738.

   ... s Father was a Native of Guinea, and his Mother a Mulatto. He had been taught to read and write, and had been received into the Church by the Sacrament of Baptism. He never had been put to any Trade, but wen ...

6. John Sutton, Sexual Offences > rape, 24th April 1745.

   ... against the form of the statute . Mary Swain . [A mulatto] first and foremost Mary Sutton [the reputed wife of the prisoner] came to our house, and asked for my mother; I told her my father and mother were i ...

7. THOMAS TOWERS, Theft > grand larceny, 29th October 1783.

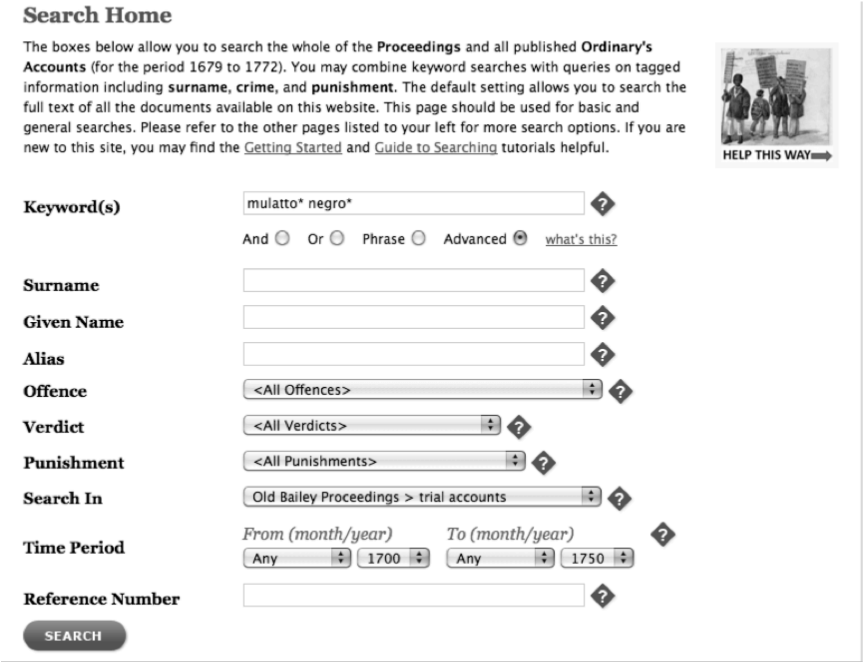
   806. THOMAS TOWERS (A Mulatto) was indicted for feloniously stealing, on the 10th of October last, one wooden half firkin, value 2 d. and twenty eight pounds weight of salt butter ...

Search results for 'mulatto' in the Old Bailey Online

After glancing through these search results, it seems clear that these are references to people, rather than horses or cloth or other things that may be black. We want to download them all to use in our analysis. We could, of course, download them one at a time, manually. But let's find a programmatic way to automate this task.

The Advanced Search on OBO

Every website's search features work differently. While searches work similarly, the intricacies of database searches may not be entirely obvious. Therefore it's important to think critically about database search options and, when available, read the documentation provided on the website. Prudent historical researchers always interrogate their sources; the procedures behind your search boxes should receive the same attention. The Old Bailey Online's advanced search form lets you refine your searches based on ten different fields including simple keywords, a date range, and a crime type. As each website's search feature is different it always pays to take a moment or two to play with and read about the search options available. In this case, read over the short explanation of the "Advanced" features by clicking on the "what's this?" link, which will explain how to refine your search further. Since we have already done the simple searches for "negro" and "mulatto", we know there will be results. However, let's use the advanced search to limit our results to records published in the Old Bailey Proceedings trial accounts from 1700 to 1750 only. You can of course change this to whatever you like, but this will make the example easier to follow. Perform the search shown in the image below. Make sure you tick the "Advanced" radio button and include the * wildcards to include pluralized entries or those with an extra "e" on the end.



Search Home

The boxes below allow you to search the whole of the **Proceedings** and all published **Ordinary's Accounts** (for the period 1679 to 1772). You may combine keyword searches with queries on tagged information including **surname**, **crime**, and **punishment**. The default setting allows you to search the full text of all the documents available on this website. This page should be used for basic and general searches. Please refer to the other pages listed to your left for more search options. If you are new to this site, you may find the Getting Started and Guide to Searching tutorials helpful.

Keyword(s): mulatto* negro*
And ○ Or ○ Phrase ○ Advanced ● what's this?

Surname:
Given Name:
Alias:
Offence: <All Offences>
Verdict: <All Verdicts>
Punishment: <All Punishments>
Search In: Old Bailey Proceedings > trial accounts
Time Period: From (month/year) Any 1700 To (month/year) Any 1750
Reference Number:

Old Bailey Advanced Search Example

Execute the search and then click on the “Calculate Total” link to see how many entries there are. We now have 13 results (if you have a different number go back and make sure you copied the example above exactly). What we want to do at this point is download all of these trial documents and analyze them further. Again, for only 13 records, you might as well download each record manually. But as more and more data comes online, it becomes more common to need to download 1,300 or even 130,000 records, in which case downloading individual records becomes impractical and an understanding of how to automate the process becomes that much more valuable. To automate the download process, we need to step back and learn how the search URLs are created on the Old Bailey website, a method common to many online databases and websites.

Understanding URL Queries

Take a look at the URL produced with the last search results page. It should look like this:

```
http://www.oldbaileyonline.org/search.jsp?foo=bar&form=searchHomePage&_divs_
fulltext=mulatto*+negro*&kwparse=advanced&_divs_div0Type_div1Type=sessionsPa
per%7CtrialAccount&fromYear=1700&fromMonth=00&toYear=1750&toMonth=99&start=
0&count=0
```

We had a look at URLs in ‘Understanding Web Pages and HTML’,[57] but this looks a lot more complex. Although longer, it is actually *not* that much more complex. But it is easier to understand by noticing how our search criteria get represented in the URL.

[57] William J. Turkel and Adam Crymble, ‘Understanding Web Pages and HTML’, *The Programming Historian* (2012).

```
http://www.oldbaileyonline.org/search.jsp
?foo=bar
&form=searchHomePage
&_divs_fulltext=mulatto*+negro*
&kwparse=advanced
&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount
&fromYear=1700
&fromMonth=00
&toYear=1750
&toMonth=99
&start=0
&count=0
```

In this view, we see more clearly our 12 important pieces of information that we need to perform our search (one per line). On the first is the Old Bailey's base website URL, followed by a query: "?" (don't worry about the `foo=bar` bit; the developers of the Old Bailey Online say that it does not do anything.) and a series of 10 *name/value pairs* put together with & characters. Together these 10 name/value pairs comprise the query string, which tells the search engine what variables to use in specific stages of the search. Notice that each name/value pair contains both a variable name: toYear, and then assigns that variable a value: 1750. This works in exactly the same way as *Function Arguments* by passing certain information to specific variables. In this case, the most important variable is `\_divs\_fulltext=` which has been given the value:

```
mulatto*+negro*
```

This holds the search term we have typed into the search box. The program has automatically added a + sign in place of a blank space (URLs cannot contain spaces); otherwise that's exactly what we've asked the Old Bailey site to find for us. The other variables hold values that we defined as well. *fromYear* and *toYear* contain our date range. Since no year has 99 months as suggested in the *toMonth* variable, we can assume this is how the search algorithm ensures all records from that year are included. There are no hard and fast rules for figuring out what each variable does because the person who built the site gets to name them. Often you can make an educated guess. All of the possible search fields on the Advanced Search page have their own name/value pair. If you'd like to find out the name of the variable so you can use it, do a new search and make sure you put a value in the field in which you are interested. After you submit your search, you'll see your value and the name associated with it as part of the URL of the search results page. With the Old Bailey Online, as with many other websites, the search form (advanced or not) essentially helps you to construct URLs that tell the database what to search for. If you can understand how the search fields are represented in the URL – which is often quite straightforward – then it becomes relatively simple to programmatically construct these URLs and thus to automate the process of downloading records.

Now try changing the "`start=0`" to "`start=10`" and hit enter. You should now have results 11-13. The "start" variable tells the website which entry should be shown at the top of the search results list. We should be able to use this knowledge to create a series of URLs that will allow us to download all 13 files. Let's turn to that now.

Systematically Downloading Files

In 'Downloading Web Pages with Python'[58] we learned that Python can download a webpage as long as we have the URL. In that lesson we used the URL to download the trial transcript of Benjamin Bowsey. In this case, we're trying to download multiple trial transcripts that meet the search criteria we outlined above without having to repeatedly re-run the program. Instead, we want a program that will download everything we need in one go. At this point we have a URL to a search results page that contains the first ten entries of our search. We also know that by changing the "start" value in the URL we can sequentially call each search results page, and ultimately retrieve all of the trial documents from them. Of course the research results don't give us the trial documents themselves, but only links to them. So we need to extract the link to the underlying records from the search results. On the Old Bailey Online website, the URLs for the individual records (the trial transcript files) can be found as links on the search results pages. We know that all trial transcript URLs contain a trial id that takes the form: "t" followed by at least 8 numbers (e.g. t17800628-33). By looking for links that contain that pattern, we can identify trial transcript URLs. As in previous lessons, let's develop an algorithm so that we can begin tackling this problem in a manner that a computer can handle. It seems this task can be achieved in four steps. We will need to:

Generate the URLs for each search results page by incrementing the "start" variable by a fixed amount an appropriate number of times.

Download each search results page as an HTML file.

Extract the URLs of each trial transcript (using the trial ID as described above) from the search results HTML files.

Cycle through those extracted URLs to download each trial transcript and save it to a directory on our computer

You'll recall that this is fairly similar to the tasks we achieved in 'Downloading Web Pages with Python'[59] and 'From HTML to a List of Words 2'.[60] First we download, then we parse out the information we're after. And in this case, we download some more.

[58] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python' *The Programming Historian* (2012).

[59] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python' *The Programming Historian* (2012).

[60] William J. Turkel and Adam Crymble, 'From HTML to a List of Words (2)', *The Programming Historian* (2012).

Downloading the search results pages

First we need to generate the URLs for downloading each search results page. We have already got the first one by using the form on the website:

```
http://www.oldbaileyonline.org/search.jsp?foo=bar&form=searchHomePage&_divs_
fulltext=mulatto*+negro*&kwparse=advanced&_divs_div0Type_div1Type=sessionsPa
per%7CtrialAccount&fromYear=1700&fromMonth=00&toYear=1750&toMonth=99&start=0
&count=0
```

We could type this URL out twice and alter the '*start*' variable to get us all 13 entries, but let's write a program that would work no matter how many search results pages or records we had to download, and no matter what we decide to search for. Study this code and then add this function to a module named `obo.py` (create a file with that name and save it to the directory where you want to do your work). The comments in the code are meant to help you decipher the various parts.

```
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth):

    import urllib2
    startValue = 0

    #each part of the URL. Split up to be easier to read.
    url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=searchHomePage&_di
vs_fulltext='
    url += query
    url += '&kwparse=' + kwparse
    url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
    url += '&fromYear=' + fromYear
    url += '&fromMonth=' + fromMonth
    url += '&toYear=' + toYear
    url += '&toMonth=' + toMonth
    url += '&start=' + str(startValue)
    url += '&count=0'

    #download the page and save the result.
    response = urllib2.urlopen(url)
    webContent = response.read()
    filename = 'search-result'
    f = open(filename + ".html", 'w')
    f.write(webContent)
    f.close
```

In this function we have split up the various Query String components and used Function Arguments so that this function can be reused beyond our specific needs right now. When we call this function we will replace the arguments with the values we want to search for. We then download the search results page in a similar manner as done in 'Downloading Web Pages with Python'.[61] Now, make a new file: `download-searches.py` and copy into it the following code. Note, the values we have passed as arguments are exactly the same as those used in the example above. Feel free to play with these to get different results or see how they work.

[61] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python' *The Programming Historian* (2012).

```
#download-searches.py
import obo

query = 'mulatto*+negro*'

obo.getSearchResults(query, "advanced", "1700", "00", "1750", "99")
```

When you run this code you should find a new file: “search-results.html” in your programming-historian directory containing the first search results page for your search. Check that this downloaded properly and then delete the file. We’re going to adapt our program to download the other page containing the other 3 entries at the same time so we want to make sure we get both. Let’s refine our getSearchResults function by adding another function argument called “entries” so we can tell the program how many pages of search results we need to download. We will use the value of entries and some simple math to determine how many search results pages there are. This is fairly straightforward since we know there are ten trial transcripts listed per page. We can calculate the number of search results pages by dividing the value of entries by 10. We will save this result to an integer variable named pageCount. It looks like this:

```
#determine how many files need to be downloaded.
pageCount = entries / 10
```

However, because pageCount is an integer and cannot have decimal places or remainders, Python will drop the remainder. You can test this by running this code in your Terminal (Mac & Linux) / Python Command Line (Windows) and printing out the value held in pageCount. (Note, from here on, we will use the word Terminal to refer to this program).

```
entries = 13
pageCount = entries / 10
print pageCount
-> 1
```

We know this should read 2 (one page containing entries 1-10, and one page containing entries 11-13). Since there is a remainder to this problem (of 3, but it doesn’t matter what the remainder is), the last 3 results won’t be downloaded, as we’ll only grab 1 page of 10 results. To get around this problem we use the modulo operator (%) in place of the usual division operator (/).[62] Modulo divides the first value by the second and returns the remainder. So if the remainder is more than 0, we know there is a partial page of results, and we need to increase the pageCount value by one. The code should now look like this:

[62] ‘Binary arithmetic operations’ *Python Reference Manual*: https://docs.python.org/release/2.5.2/ref/binary.html

```
#determine how many files need to be downloaded.
pageCount = entries / 10
remainder = entries % 10
if remainder > 0:
    pageCount += 1
```

If we add this to our `getSearchResults` function just under the *startValue = 0* line, our program, the code can now calculate the number of pages that need to be downloaded. However, at this stage it will still only download the first page since we have only told the downloading section of the function to run once. To correct this, we can add that downloading code to a for loop which will download once for every number in the `pageCount` variable. If it reads 1, then it will download once; if it reads 5 it will download five times, and so on. Immediately after the if statement you have just written, add the following line and indent everything down to `f.close` one additional tab so that it is all enclosed in the for loop:

```
for pages in range(1, pageCount+1):
    print pages
```

Since this is a for loop, all of the code we want to run repeatedly needs to be intended as well. You can see if you have done this correctly by looking at the finished code example below. This loop takes advantage of Python's range funciton.[63] To understand this for loop it is probably best to think of `pageCount` as equal to 2 as it is in the example. This two lines of code then means: start running with an initial loop value of 1, and each time you run, add 1 more to that value. When the loop value is the same as `pageCount`, run once more and then stop. This is particularly valuable for us because it means we can tell our program to run exactly once for each search results page and provides a flexible new skill for controlling how many times a for loop runs. If you would like to practice with this new and powerful way of writing for loops, you can open your Terminal and play around.

```
pageCount = 2
for pages in range(1, pageCount+1):
    print pages

-> 1
-> 2
```

Before we add all of this code together to our `getSearchResults` function, we have to make two final adjustments. At the end of the for loop (but still inside the loop), and after our downloading code has run we will need to change the `startValue` variable, which is used in building the URL of the page we want to download. If we forget to do this, our program will repeatedly download the first search results page since we are not actually changing anything in the initial URL. The `startValue` variable, as

[63] 'The range() function' *Python*: https://docs.python.org/2/tutorial/controlflow.html#the-range-function

discussed above, is what controls which search results page we want to download. Therefore, we can request the next search results page by increasing the value of `startValue` by 10 after the initial download has completed. If you are not sure where to put this line you can peek ahead to the finished code example below.

Finally, we want to ensure that the name of the file we have downloaded is different for each file. Otherwise, each download will save over the previous download, leaving us with only a single file of search results. To solve this, we can adjust the contents of the `filename` variable to include the value held in `startValue` so that each time we download a new page, it gets a different name. Since `startValue` is an integer, we will have to convert it to a string before we can add it to the filename variable. Adjust the line in your program that pertains to the `filename` variable to looks like this:

```
filename = 'search-result' + str(startValue)
```

You should now be able to add these new lines of code to your getSearchResults function. Recall we have made the following additions:

Add entries as an additional function argument right after toMonth

Calculate the number of search results pages and add this immediately after the line that begins with startValue = 0 (before we build the URL and start downloading)

Follow this immediately with a for loop that will tell the program to run once for each search results page, and indent the rest of the code in the function so that it is inside the new loop.

The last line in the for loop should now increase the value of the startValue variable each time the loop runs.

Adjust the existing filename variable so that each time a search results page is downloaded it gives the file a unique name.

The finished function code in your `obo.py` file should look like this:

```
#create URLs for search results pages and save the files
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth, e
ntries):

    import urllib2

    startValue = 0

    #this is new! Determine how many files need to be downloaded.
    pageCount = entries / 10
    remainder = entries % 10
    if remainder > 0:
        pageCount += 1

    #this line is new!
    for pages in range(1, pageCount +1):

        #each part of the URL. Split up to be easier to read.
        url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=search
HomePage&_divs_fulltext='
        url += query
        url += '&kwparse=' + kwparse
        url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
        url += '&fromYear=' + fromYear
        url += '&fromMonth=' + fromMonth
        url += '&toYear=' + toYear
        url += '&toMonth=' + toMonth
        url += '&start=' + str(startValue)
        url += '&count=0'

        #download the page and save the result.
        response = urllib2.urlopen(url)
        webContent = response.read()
        filename = 'search-result' + str(startValue)
        f = open(filename + ".html", 'w')
        f.write(webContent)
        f.close

        #this lines is new!
        startValue = startValue + 10
```

To run this new function, add the extra argument to `download-searches.py` and run the program again:

```
#download-searches.py
import obo

query = 'mulatto*+negro*'

obo.getSearchResults(query, "advanced", "1700", "00", "1750", "99", 13)
```

Great! Now we have both search results pages, called `search-result0.html` and `search-result10.html`. But before we move onto the next step in the algorithm, let's take care of some housekeeping. Our `programming-historian` directory will quickly become unwieldy if we download multiple search results pages and trial transcripts. Let's have

Python make a new directory named after our search terms. Study and then copy the following to `obo.py`.

```
def newDir(newDir):
    import os

    dir = newDir

    if not os.path.exists(dir):
        os.makedirs(dir)
```

We want to call this new function in `getSearchResults`, so that our search results pages are downloaded to a directory with the same name as our search query. This will keep our `programming-historian` directory more organized. To do this we will create a new directory using the `os` library, short for “operating system”. That library contains a function called `makedirs`, which, unsurprisingly, makes a new directory. You can try this out using the Terminal.

```
import os

query = "myNewDirectory"
if not os.path.exists(query):
    os.makedirs(query)
```

This program will check to see if your computer already has a directory with this name. If not, you should now have a directory called `myNewDirectory` on your computer. On a Mac this is probably located in your `/Users/username/` directory, and on Windows you should be able to find it in the `Python` directory on your computer, the same in which you opened your command line program. If this worked you can delete the directory from your hard drive, since it was just for practice. Since we want to create a new directory named after the query that we input into the Old Bailey Online website, we will make direct use of the query function argument from the getSearchResults function. To do this, import the `os` directory after you have imported `urllib2` and then add the code you have just written immediately below. Your getSearchResults function should now look like this:

```
#create URLs for search results pages and save the files
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth, e
ntries):

    import urllib2, os

    #This line is new! Create a new directory
    if not os.path.exists(query):
        os.makedirs(query)

    startValue = 0

    #Determine how many files need to be downloaded.
    pageCount = entries / 10
    remainder = entries % 10
    if remainder > 0:
        pageCount += 1

    for pages in range(1, pageCount +1):

        #each part of the URL. Split up to be easier to read.
        url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=search
HomePage&_divs_fulltext='
        url += query
        url += '&kwparse=' + kwparse
        url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
        url += '&fromYear=' + fromYear
        url += '&fromMonth=' + fromMonth
        url += '&toYear=' + toYear
        url += '&toMonth=' + toMonth
        url += '&start=' + str(startValue)
        url += '&count=0'

        #download the page and save the result.
        response = urllib2.urlopen(url)
        webContent = response.read()

        #save the result to the new directory
        filename = 'search-result' + str(startValue)

        f = open(filename + ".html", 'w')
        f.write(webContent)
        f.close

        startValue = startValue + 10
```

The last step for this function is to make sure that when we save our search results pages, we save them in this new directory. To do this we can make a minor adjustment to the filename variable so that the file ends up in the right place. There are many ways we can do this, the easiest of which is just to append the new directory name plus a slash to the name of the file:

```
filename = query + '/' + 'search-result' + str(startValue)
```

If your computer is running Windows you will need to use a backslash instead of a forward slash in the above example. Add the above line to your `getSearchResults` page in lieu of the current `filename` description.

If you are running Windows, chances are your `downloadSearches.py` program will now crash when you run it because you are trying to create a director with a * in it. Windows does not like this. To get around this problem we can use regular expressions[64] to remove any non-Windows-friendly characters. We used regular expressions previously in 'Counting Word Frequencies with Python'.[65] To remove non-alpha-numeric characters from the query, first import the regular expressions library immediately after you have imported the `os` library, then use the `re.sub()` function to create a new string named `cleanQuery` that contains only alphanumeric characters. You will then have to substitute `cleanQuery` as the variable used in the `os.path.exists()`, `os.makedirs()`, and `filename` declarations.

```
import urllib2, os, re
cleanQuery = re.sub(r'\W+', '', query)
if not os.path.exists(cleanQuery):
        os.makedirs(cleanQuery)

...

filename = cleanQuery + '/' + 'search-result' + str(startValue)
```

The final version of your function should look like this:

```
#create URLs for search results pages and save the files
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth, e
ntries):

    import urllib2, os, re

    cleanQuery = re.sub(r'\W+', '', query)

    #Create a new directory
    if not os.path.exists(cleanQuery):
        os.makedirs(cleanQuery)

    startValue = 0

    #determine how many files need to be downloaded.
    pageCount = entries / 10
    remainder = entries % 10
    if remainder > 0:
        pageCount += 1

    for pages in range(1, pageCount+1):

        #each part of the URL. Split up to be easier to read.
```

[64] 'Regular expression operations', *Python*: https://docs.python.org/2/library/re.html

[65] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *the Programming Historian* (2012).

```
        url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=search
HomePage&_divs_fulltext='
        url += query
        url += '&kwparse=' + kwparse
        url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
        url += '&fromYear=' + fromYear
        url += '&fromMonth=' + fromMonth
        url += '&toYear=' + toYear
        url += '&toMonth=' + toMonth
        url += '&start=' + str(startValue)
        url += '&count=0'

        #download the page and save the result.
        response = urllib2.urlopen(url)
        webContent = response.read()

        #save the result to the new directory
        filename = cleanQuery + '/' + 'search-result' + str(startValue)
        f = open(filename + ".html", 'w')
        f.write(webContent)
        f.close

        startValue = startValue + 10
```

This time we tell the program to download the trials and put them in the new directory rather than our `programming-historian` directory. Run `download-searches.py` once more to ensure this worked and you understand how to save files to a particular directory using Python.

Downloading the individual trial entries

At this stage we have created a function that can download all of the search results HTML files from the Old Bailey Online website for an advanced search that we have defined, and have done so programmatically. Now for the next step in the algorithm: Extract the URLs of each trial transcript from the search results HTML files. In the lessons that precede this one (eg, 'Downloading Web Pages with Python'[66]), we have worked with the printer friendly versions of the trial transcripts, so we will continue to do so. We know that the printer friendly version of Benjamin Bowsey's trial is located at the URL:

```
http://www.oldbaileyonline.org/print.jsp?div=t17800628-33
```

In the same way that changing query strings in the URLs yields different search results, changing the URL for trial records – namely substituting one trial ID for another – we will get the transcript for that new trial. This means that to find and download the 13 matching files, all we need are these trial IDs. Since we know that search results pages on websites generally contain a link to the pages described, there is a good chance that we can find these links embedded in the HTML code. If we can scrape this information from the downloaded search results pages, we can then use

[66] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python', *The Programming Historian* (2012).

that information to generate a URL that will allow us to download each trial transcript. This is a technique that you can use for most search result pages, not just Old Bailey Online! To do this, we must first find where the trial IDs are amidst the HTML code in the downloaded files, and then determine a way to consistently isolate them using code so that no matter which search results page we download from the site we are able to find the trial transcripts. First, open `search-results0.html` in Komodo Edit and have a look for the list of the trials. The first entry starts with "Anne Smith" so you can use the "find" feature in Komodo Edit to jump immediately to the right spot. Notice Anne's name is part of a link:

```
http://www.oldbaileyonline.org/browse.jsp?id=t17160113-18&div=t17160113-18&t
erms=mulatto|negro#highlight
```

Perfect, the link contains the trial ID! Scroll through the remaining entries and you'll find the same is true. Lucky for us, the site is well formatted and it looks like each link starts with "browse.jsp?id=" followed by the trial ID and finished with an &, in Anne's case: "browse.jsp?id=t17160113-18&". We can write a few lines of code that can isolate those IDs. Take a look at the following function. This function also uses the `os` library, in this case to list all of the files located in the directory created in the previous section. The `os` library contains a range of useful functions that mirror the types of tasks you would expect to be able to do with your mouse in the Mac Finder or Windows such as opening, closing, creating, deleting, and moving files and directories, and is a great library to master – or at least familiarize yourself with.

```
def getIndivTrials(query):
    import os, re

    cleanQuery = re.sub(r'\W+', '', query)
    searchResults = os.listdir(cleanQuery)

    print searchResults
```

Create and run a new program called `extract-trial-ids.py` with the following code. Make sure you input the same value into the query argument as you did in the previous example:

```
import obo

obo.getIndivTrials("mulatto*+negro*")
```

If everything went right, you should see a list containing the names of all the files in your new "mulatto*+negro*" directory, which at this point should be the two search results pages. Ensure this worked before moving forward. Since we saved all of the search results pages with a filename that includes "search-results", we now want to open each file with a name containing "search-results", and extract all trial IDs found therein. In this case we know we have 2, but we want our code to be as reusable as possible (with reason, of course!) Restricting this action to files named "searchresults" will mean that this program will work as intended even if the directory contains many other unrelated files because the program will skip over anything with a different name. Add the following to your getIndivTrials() function, which will check if each file contains "search-results" in its name. If it does, the file will be opened and the contents saved to a variable named text. That text variable will then be parsed looking for the trial ID, which we know always follows "browse.jsp?id=". If and when that trial ID is found it will be saved to a list and printed to the command output, which leaves us with all of the information we need to then write a program that will download the desired trials.

```
    import os, re

    cleanQuery = re.sub(r'\W+', '', query)
    searchResults = os.listdir(cleanQuery)

    urls = []

    #find search-results pages
    for files in searchResults:
        if files.find("search-result") != -1:
            f = open(cleanQuery + "/" + files, 'r')
            text = f.read().split(" ")
            f.close()

            #look for trial IDs
            for words in text:
                if words.find("browse.jsp?id=") != -1:
                    #isolate the id
                    urls.append(words[words.find("id=") +3: words.find("&")])

    print urls
```

That last line of the for loop may look tricky, but make sure you understand it before moving on. The words variable is checked to see if it contains the characters "id=" (without the quotes), which of course refers to a specific trial transcript ID. If it does, we use the slice string method to capture only the chunk between *id=* and *&* and append it to the url list. If we knew the exact index positions of this substring we could have used those numerical values instead. However, by using the *find()* string method we have created a much more flexible program. The following code does exactly the same thing as that last line in a less condensed manner.

```
idStart = words.find("id=") + 3
idEnd = words.find("&")
trialID = words[idStart: idEnd]

urls.append(trialID)
```

When you re-run `extract-trial-ids.py`, you should now see a list of all the trial IDs. We can add a couple extra lines to turn these into proper URLs and download the whole list to our new directory. We'll also use the `time` library to pause our program for three seconds between downloads– a technique called throttling. It's considered good form not to pound someone's server with many requests per second; and the slight delay makes it more likely that all the files will actually download rather than time out.[67] Add the following code to the end of your `getIndivTrials()` function. This code will generate the URL of each individual page, download the page to your computer, place it in your new directory, save the file, and pause for 3 seconds before moving on to the next trial. This work is all contained in a for loop, and will run once for every trial in your url list.

```
def getIndivTrials(query):
    #...
    import urllib2, time

    #import built-in python functions for building file paths
    from os.path import join as pjoin

    for items in urls:
        #generate the URL
        url = "http://www.oldbaileyonline.org/print.jsp?div=" + items

        #download the page
        response = urllib2.urlopen(url)
        webContent = response.read()

        #create the filename and place it in the new directory
        filename = items + '.html'
        filePath = pjoin(cleanQuery, filename)

        #save the file
        f = open(filePath, 'w')
        f.write(webContent)
        f.close

        #pause for 3 second
        time.sleep(3)
```

If we put this all together into a single function it should look something like this. (Note, we've put all the "import" calls at the top to keep things cleaner).

```
def getIndivTrials(query):
    import os, re, urllib2, time

    #import built-in python functions for building file paths
    from os.path import join as pjoin

    cleanQuery = re.sub(r'\W+', '', query)
    searchResults = os.listdir(cleanQuery)

    urls = []

    #find search-results pages
    for files in searchResults:
        if files.find("search-result") != -1:
```

[67] 'HTTP Error 408 Request Timeout' *Checkupdown*: http://www.checkupdown.com/status/E408.html

```
            f = open(cleanQuery + "/" + files, 'r')
            text = f.read().split(" ")
            f.close()

            #look for trial IDs
            for words in text:
                if words.find("browse.jsp?id=") != -1:
                    #isolate the id
                    urls.append(words[words.find("id=") +3: words.find("&")])

    #new from here down!
    for items in urls:
        #generate the URL
        url = "http://www.oldbaileyonline.org/print.jsp?div=" + items

        #download the page
        response = urllib2.urlopen(url)
        webContent = response.read()

        #create the filename and place it in the new directory
        filename = items + '.html'
        filePath = pjoin(cleanQuery, filename)

        #save the file
        f = open(filePath, 'w')
        f.write(webContent)
        f.close

        #pause for 3 seconds
        time.sleep(3)
```

Let’s add the same three-second pause to our `getSearchResults` function to be kind to the Old Bailey Online servers:

```
#create URLs for search results pages and save the files
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth, entri
es):

    import urllib2, os, re, time

    cleanQuery = re.sub(r'\W+', '', query)
    if not os.path.exists(cleanQuery):
        os.makedirs(cleanQuery)

    startValue = 0

    #Determine how many files need to be downloaded.
    pageCount = entries / 10
    remainder = entries % 10
    if remainder > 0:
        pageCount += 1

    for pages in range(1, pageCount +1):

        #each part of the URL. Split up to be easier to read.
        url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=searchHome
Page&_divs_fulltext='
        url += query
        url += '&kwparse=' + kwparse
        url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
        url += '&fromYear=' + fromYear
        url += '&fromMonth=' + fromMonth
        url += '&toYear=' + toYear
        url += '&toMonth=' + toMonth
        url += '&start=' + str(startValue)
        url += '&count=0'

        #download the page and save the result.
        response = urllib2.urlopen(url)
        webContent = response.read()

        #save the result to the new directory
        filename = cleanQuery + '/' + 'search-result' + str(startValue)

        f = open(filename + ".html", 'w')
        f.write(webContent)
        f.close

        startValue = startValue + 10

        #pause for 3 seconds
        time.sleep(3)
```

Finally, call the function in the `download-searches.py` program.

```
#download-searches.py
import obo

query = 'mulatto*+negro*'

obo.getSearchResults(query, "advanced", "1700", "00", "1750", "99", 13)

obo.getIndivTrials(query)
```

Now, you've created a program that can request and download files from the Old Bailey website based on search parameters you define, all without visiting the site!

In case a file does not download

Check to make all thirteen files have downloaded properly. If that's the case for you, that's great! However, there's a possibility that this program stalled along the way. That's because our program, though running on your own machine, relies on two factors outside of our immediate control: the speed of the Internet, and the response time of the Old Bailey Online server at that moment. It's one thing to ask Python to download a single file, but when we start asking for a file every 3 seconds there's a greater chance the server will either time out or fail to send us the file we are after.

If we were using a web browser to make these requests, we'd eventually get a message that the "connection had timed out" or something of the sort. We all see this from time to time. However, our program isn't built to handle or relay such error messages, so instead you'll realize it when you discover that the program has not returned the expected number of files or just seemingly does nothing. To prevent frustration and uncertainty, we want a fail-safe in our program that will attempt to download each trial. If for whatever reason it fails, we'll make a note of it and move on to the next trial.

To do this, we will make use of the Python try / except error handling[68] mechanism, as well as a new library: socket. Try and Except are a lot like an *if / else* statement. When you ask Python to *try* something, it will attempt to run the code; if the code fails to achieve what you have defined, it will run the *except* code. This is often used when dealing with errors, known as *error handling*. We can use this to our advantage by telling our program to attempt downloading a page. If it fails, we'll ask it to let us know which file failed and then move on. To do this we need to use the `socket` library, which will allow us to put a time limit on a download attempt before moving on. This involves altering the `getIndivTrials` function.

First, we need to load the `socket` library, which should be done in the same way as all of our previous library imports. We will also need to set the default socket timeout length – how long do we want to try to download a page before we give up. This should go immediately after the comment that begins with `#download the page`

```
import os, urllib2, time, socket

    #...
        #download the page
        socket.setdefaulttimeout(10)
```

Then, we need a new python list that will hold all of the urls that failed to download. We will call this `failedAttempts` and you can insert it immediately after the `import` instructions:

```
 failedAttempts = []
```

Finally, we can add the *try / except* statement, which is added in much the same way as an *if / else* statement would be. In this case, we will put all of the code designed to download and save the trials in the try statement, and in the except statement we will tell the program what we want it to do if that should fail. Here, we will append the url that failed to download to our new list, `failedAttempts`

```
#...

        socket.setdefaulttimeout(10)

        try:
            response = urllib2.urlopen(url)
            webContent = response.read()

            #create the filename and place it in the new "trials" directory
            filename = items + '.html'
            filePath = pjoin(newDir, filename)
```

[68] 'Errors and Exceptions', *Python*: https://docs.python.org/2/tutorial/errors.html

```
            #save the file
            f = open(filePath, 'w')
            f.write(webContent)
            f.close
        except:
            failedAttempts.append(url)
```

Finally, we will tell the program to print the contents of the list to the command output so we know which files failed to download. This should be added as the last line in the function.

```
print "failed to download: " + str(failedAttempts)
```

Now when you run the program, should there be a problem downloading a particular file, you will receive a message in the Command Output window of Komodo Edit. This message will contain any URLs of files that failed to download. If there are only one or two, it's probably fastest just to visit the pages manually and use the "Save As" feature of your browser. If you are feeling adventurous, you could modify the program to automatically download the remaining files. The final version of your `getSearchResults()`, `getIndivTrials()`, and `newDir()` functions should now look like this:

```
def getSearchResults(query, kwparse, fromYear, fromMonth, toYear, toMonth, entri
es):

    import urllib2, os, re, time

    cleanQuery = re.sub(r'\W+', '', query)
    if not os.path.exists(cleanQuery):
        os.makedirs(cleanQuery)

    startValue = 0
    #determine how many files need to be downloaded.
    pageCount = entries / 10
    remainder = entries % 10
    if remainder > 0:
        pageCount += 1

    for pages in range(1, pageCount+1):

        #each part of the URL. Split up to be easier to read.
        url = 'http://www.oldbaileyonline.org/search.jsp?foo=bar&form=searchHome
Page&_divs_fulltext='
        url += query
        url += '&kwparse=' + kwparse
        url += '&_divs_div0Type_div1Type=sessionsPaper%7CtrialAccount'
        url += '&fromYear=' + fromYear
        url += '&fromMonth=' + fromMonth
        url += '&toYear=' + toYear
        url += '&toMonth=' + toMonth
        url += '&start=' + str(startValue)
        url += '&count=0'

        #download the page and save the result.
        response = urllib2.urlopen(url)
        webContent = response.read()
        filename = cleanQuery + '/' + 'search-result' + str(startValue)
        f = open(filename + ".html", 'w')
        f.write(webContent)
        f.close

        startValue = startValue + 10

        #pause for 3 seconds
        time.sleep(3)

def getIndivTrials(query):
    import os, re, urllib2, time, socket

    failedAttempts = []

    #import built-in python functions for building file paths
    from os.path import join as pjoin

    cleanQuery = re.sub(r'\W+', '', query)
    searchResults = os.listdir(cleanQuery)

    urls = []

    #find search-results pages
    for files in searchResults:
        if files.find("search-result") != -1:
            f = open(cleanQuery + "/" + files, 'r')
            text = f.read().split(" ")
            f.close()

            #look for trial IDs
            for words in text:
                if words.find("browse.jsp?id=") != -1:
                    #isolate the id
                    urls.append(words[words.find("id=") +3: words.find("&")])

    for items in urls:
        #generate the URL
        url = "http://www.oldbaileyonline.org/print.jsp?div=" + items

        #download the page
        socket.setdefaulttimeout(10)
        try:
            response = urllib2.urlopen(url)
            webContent = response.read()

            #create the filename and place it in the new directory
            filename = items + '.html'
            filePath = pjoin(cleanQuery, filename)

            #save the file
            f = open(filePath, 'w')
            f.write(webContent)
            f.close
        except:
            failedAttempts.append(url)
        #pause for 3 seconds
        time.sleep(3)
    print "failed to download: " + str(failedAttempts)

def newDir(newDir):
```

```
import os

dir = newDir

if not os.path.exists(dir):
    os.makedirs(dir)
```

Further Reading

For more advanced users, or to become a more advanced user, you may find it worthwhile to read about achieving this same process using Application Programming Interfaces (API). A website with an API will generally provide instructions on how to request certain documents. It’s a very similar process to what we just did by interpreting the URL Query Strings, but without the added detective work required to decipher what each variable does. If you are interested in the Old Bailey Online, they have recently released an API and the documentation can be quite helpful:

Old Bailey Online API (http://www.oldbaileyonline.org/static/DocAPI.jsp)

Python Best way to create directory if it doesn’t exist for file write? (http://stackoverflow.com/questions/273192/python-best-way-to-create-directory-if-it-doesnt-exist-for-file-write)

About the Author

Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

10. Intro to the Zotero API

Amanda Morton – 2013

Lesson Goals

In this lesson, you'll learn how to use python with the Zotero API to interact with your Zotero library. The Zotero API is a powerful interface that would allow you to build a complete Zotero client from scratch if you so desired. But like most APIs, it works in small, discrete steps, so we have to build our way up to the complicated requests we might want to use to access our Zotero libraries. But this incremental building gives us plenty of time to learn as we go along.

What is Zotero?

Zotero is a browser-based research tool that allows you to collect and store content. If you are new to Zotero or do not regularly use it, you may want to familiarize yourself with the Zotero site[69] and its helpful Quick Start Guide.[70] Additionally, while you will not need it for this introductory lesson, we advise that you download the current version of the libZotero GitHub library[71] and store it in the directory you have chosen to use for these lessons.

Installing libZotero

Using what you learned in Fred Gibbs' lesson on 'Installing Python Modules with pip',[72] we'll use pip[73] to install libZotero, a python library that will allow us to interact with the Zotero API. To install the library, in your command line/terminal window enter:

```
pip install libZotero
```

Remember that you may need to use the `sudo` preface and enter your password to allow the installation to proceed.

Zotero Hello World

Once *libZotero* is installed, we can use it to talk to the Zotero server using Python. In your text editor, run the following:

```
#make the libZotero library available
from libZotero import zotero
```

[69] *Zotero: https://www.zotero.org/*

[70] 'Quick Start Guide', *Zotero*: https://www.zotero.org/support/quick_start_guide

[71] 'libZotero Github library', *Github*: https://github.com/fcheslack/libZotero

[72] Fred Gibbs, 'Installing Python Modules with pip', *The Programming Historian* (2013).

[73] 'Pip', *Python*: https://pypi.python.org/pypi/pip

Once we've successfully imported the name zotero from the library, we can create and define a Zotero library "object" (*zlib*, in this example), which will be our means of creating a request for the Zotero server and returning its data. When we create the library object we will need to specify whether we're accessing an individual or group library and include the Zotero library's ID number. Depending on the type of library we're accessing and the things we plan to do with it, we may also need to include an authentication key, which functions sort of like a password.

```
#create Zotero library object called "zlib"
zlib=zotero.Library('group','<insert group ID>','<null>',
'<insert API key>')
```

For this lesson, you can use your own group or individual library, or you can use the library we've created for this lesson at 'Programming Historian 2'.[74]

If you want to use your own group or individual library, you will need to retrieve your group or user ID and your own API key. If you use your individual library, you'll also need to replace the word `group` with the word `user` in the above code.

Your group ID can be found by hovering over the RSS option on your library feed. The ID is the numeric part of the URL. Your group API key, if one has been created, is located in your account settings. If there is no key assigned to the group, and you are the end user, you can create a new key on the same page.

To use the 'Programming Historian 2' group library, use the following:

```
 Group ID: 155975
 API key: 9GLmvmZ1K1qGAz9QWcdlyf6L
```

```
zlib=zotero.Library('group','155975','<null>',
'9GLmvmZ1K1qGAz9QWcdlyf6L')
```

Once we've defined our object, we can use it to interact with the information in the library.

Retrieving Item Information

Zotero has parent items and child items. Parents are typically top-level objects with metadata, and children are usually things like notes and file attachments. For this portion of the lesson, we'll be pulling information from the first five top-level items in our collection.

```
# retrieve the first five top-level items.
items = zlib.fetchItemsTop({'limit': 5, 'content': 'json,bib,coins'})
```

[74] 'Programming Historian 2 – Zotero Group', *Zotero*: https://www.zotero.org/groups/programming_historian_2

Your output for this step, if you are using our sample collection, should look like this:

```
value stored in cache - https://api.zotero.org/groups/155975/items/top?limit
=5&content=
json%2Cbib%2Ccoins&key=9GLmvmZ1K1qGAz9QWcdlyf6L
```

Next, we can print some basic information about these items.

```
# print some data about these five items
for item in items:
print 'Item Type: %s | Key: %s | Title: %s' % (item.itemType,
item.itemKey, item.title)
```

This step should retrieve the item type (journal article, webpage, etc.), the key, and item title.

```
Item Type: webpage | Key: TK5Z4H9J | Title: Benjamin Bowsey
Item Type: webpage | Key: 3A2RWZ8A | Title: Y a t-il une
Histoire Numerique 2.0?
Item Type: webpage | Key: 79U2EACW | Title: Digitization
boosts access, collaboration, UCLA prof says
Item Type: journalArticle | Key: 39V7A2SZ | Title: History
and the second decade of the Web
Item Type: journalArticle | Key: JRCM2PM7 | Title: The
Pasts and Futures of Digital History
```

We can also pull the bibliographic information associated with our first five items:

```
for item in items:
    print item.bibContent
```

Running this command will print the bibliographic content stored on the Zotero servers for these items:

```
<div class="csl-bib-body" style="line-height: 1.35; padding-left: 2em; text-
indent:-2em;" xmlns="http://www.w3.org/1999/xhtml">
<div class="csl-entry">“Benjamin Bowsey.” Accessed March 29, 2013. http://ww
w.oldbaileyonline.org/print.jsp?div=t17800628-33.</div>
</div>
<div class="csl-bib-body" style="line-height: 1.35; padding-left: 2em; text-
indent:-2em;" xmlns="http://www.w3.org/1999/xhtml">
  <div class="csl-entry">Noiret, Serge. “Y a T-il Une Histoire Numerique 2.0
?” Contribution to book. Accessed July 21, 2011. http://cadmus.eui.eu/handle
/1814/18074.</div>
</div>
<div class="csl-bib-body" style="line-height: 1.35; padding-left: 2em; text-
indent:-2em;" xmlns="http://www.w3.org/1999/xhtml">
  <div class="csl-entry">Rushton, Tullia. “Digitization Boosts Access, Colla
boration, UCLA Prof Says.” <i>Chronicle of Higher Education</i>, January 20,
 2010. http://dukechronicle.com/article/digitization-boosts-access-collabora
tion-ucla-prof-says.</div>
</div>
<div class="csl-bib-body" style="line-height: 1.35; padding-left: 2em; text-
indent:-2em;" xmlns="http://www.w3.org/1999/xhtml">
  <div class="csl-entry">Cohen, Daniel J. “History and the Second Decade of
```

```
the Web.” <i>Rethinking History</i> 8, no. 2 (2004): 293. doi:10.1080/136425
20410001683950.</div>
</div>
<div class="csl-bib-body" style="line-height: 1.35; padding-left: 2em; text-
indent:-2em;" xmlns="http://www.w3.org/1999/xhtml">
  <div class="csl-entry">Ayers, Edward L. “The Pasts and Futures of Digital
History” (1999). http://www.vcdh.virginia.edu/PastsFutures.html.</div>
</div>
```

Now that we have worked through retrieving information using the Zotero API, we can continue to use it to interact with the items stored in our library.

Editor’s Note: This lesson is the first of three lessons on the Zotero API. The next lesson in this series is ‘Creating New Items in Zotero’.[75]

About the Author

Amanda Morton is a DH Fellow at the Center for History and New Media.

[75] Amanda Morton, ‘Creating New Items in Zotero’, *The Programming Historian* (2013).

11. Data Mining the Internet Archive Collection

Caleb McDaniel – 2014

The collections of the Internet Archive include many digitized historical sources.[76] Many contain rich bibliographic data in a format called MARC. In this lesson, you'll learn how to use Python to automate the downloading of large numbers of MARC files from the Internet Archive and the parsing of MARC records for specific information such as authors, places of publication, and dates. The lesson can be applied more generally to other Internet Archive files and to MARC records found elsewhere.

Lesson Goals

The collections of the Internet Archive (IA) include many digitized sources of interest to historians, including early JSTOR journal content,[77] John Adams's personal library,[78] and the Haiti collection at the John Carter Brown Library.[79] In short, to quote Programming Historian Ian Milligan, "The Internet Archive rocks."

In this lesson, you'll learn how to download files from such collections using a Python module specifically designed for the Internet Archive. You will also learn how to use another Python module designed for parsing MARC XML records, a widely used standard for formatting bibliographic metadata.

For demonstration purposes, this lesson will focus on working with the digitized version of the Anti-Slavery Collection at the Boston Public Library in Copley Square.[80] We will first download a large collection of MARC records from this collection, and then use Python to retrieve and analyze bibliographic information about items in the collection. For example, by the end of this lesson, you will be able to create a list of every named place from which a letter in the antislavery collection was written, which you could then use for a mapping project or some other kind of analysis.

[76] 'Internet Archive': https://archive.org/

[77] 'JSTOR Early Journal Content': https://archive.org/details/jstor_ejc

[78] 'The John Adams Library at the Boston Public Library': https://archive.org/details/johnadamsBPL

[79] 'John Carter Brown Library – Haiti Collection': https://archive.org/details/jcbhaiti

[80] 'Boston Public Library Anti-Slavery Collection': https://archive.org/details/bplscas

For Whom Is This Useful?

This intermediate lesson is good for users of the Programming Historian who have completed general lessons on downloading files and performing text analysis on them, but would like an applied example of these principles. It will also be of interest to historians or archivists who work with the MARC format or the Internet Archive on a regular basis.

Before You Begin

We will be working with two Python modules that are not included in Python's standard library.

The first, internetarchive,[81] provides programmatic access to the Internet Archive. The second, pymarc,[82] makes it easier to parse MARC records.

The easiest way to download both is to use pip, the python package manager. Begin by installing `pip` using Fred Gibbs' 'Installing Python Modules with pip' tutorial.[83] Then issue these commands at the command line: To install `internetarchive`:

```
sudo pip install internetarchive
```

To install `pymarc`:

```
sudo pip install pymarc
```

Now you are ready to go to work!

The Antislavery Collection at the Internet Archive

The Boston Public Library's anti-slavery collection at Copley Square contains not only the letters of William Lloyd Garrison, one of the icons of the American abolitionist movement, but also large collections of letters by and to reformers somehow connected to him. And by "large collection," I mean large. According to the library's estimates, there are over 16,000 items at Copley.

As of this writing, approximately 7,000 of those items have been digitized and uploaded to the Internet Archive. This is good news, not only because the Archive is committed to making its considerable cultural resources free for download, but also because each uploaded item is paired with a wealth of metadata suitable for machine-reading.

[81] 'internetarchive' *Python*: https://pypi.python.org/pypi/internetarchive

[82] 'pymarc' *Python*: https://pypi.python.org/pypi/pymarc/

[83] Fred Gibbs, 'Installing Python Modules with pip', *The Programming Historian* (2013).

Take a letter sent by Frederick Douglass to William Lloyd Garrison.[84] Anyone can read the original manuscript online,[85] without making the trip to Boston, and that alone may be enough to revolutionize and democratize future abolitionist historiography. But you can also download multiple files related to the letter that are rich in metadata,[86] like a Dublin Core record[87] and a fuller MARCXML record[88] that uses the Library of Congress's MARC 21 Format for Bibliographic Data.[89]

Stop and think about that for a moment: every item uploaded from the Collection contains these things. Right now, that means historians have access to rich metadata, full images, and partial descriptions for thousands of antislavery letters, manuscripts, and publications.[90]

Accessing an IA Collection in Python

Internet Archive (IA) collections and items all have a unique identifier, and URLs to collections and items all look like this:

```
http://archive.org/details/[IDENTIFIER]
```

So, for example, here is a URL to the Archive item discussed above, Douglass's letter to Garrison:

```
http://archive.org/details/lettertowilliaml00doug
```

And here is a URL to the entire antislavery collection at the Boston Public Library:

```
http://archive.org/details/bplscas/
```

Because the URLs are so similar, the only way to tell that you are looking at a collection page, instead of an individual item page, is to examine the page layout. An item page usually has a left-hand sidebar that says "View the Book" and lists links for reading the item online or accessing other file formats. A collection page will probably have a "Spotlight Item" in the lefthand sidebar instead. You can browse to different collections through

[84] ‘Letter from Frederick Douglass to William Lloyd Garrison, 1846’, *Internet Archive*: https://archive.org/details/lettertowilliaml00doug

[85] ‘Letter to William Lloyd Garrison – manuscript’, *Internet Archive*: http://archive.org/stream/lettertowilliaml00doug/39999066767938#page/n0/mode/2up

[86] ‘Index of /29/items/lettertowilliaml00doug/’, *Internet Archive*: http://archive.org/download/lettertowilliaml00doug

[87] ‘Letter to William Lloyd Garrison – Dublin Core’: *http://ia801703.us.archive.org/29/items/lettertowilliaml00doug/lettertowilliaml00doug_dc.xml*

[88] ‘Letter to William Lloyd Garrison – MARCXML’: http://ia801703.us.archive.org/29/items/lettertowilliaml00doug/lettertowilliaml00doug_marc.xml

[89] ‘MARC 21 Format for Bibliographic Data’, *Library of Congress* (1999): http://www.loc.gov/marc/bibliographic/

[90] ‘Search bplscas’, *Internet Archive*: http://archive.org/search.php?query=collection%3Abplscas&sort=-publicdate

the eBook and Texts portal,[91] and you may also want to read a little bit about the way that items and item URLs are structured.[92]

Once you have a collection's identifier—in this case, `bplscas`—seeing all of the items in the collection is as easy as navigating to the Archive's advanced search page,[93] selecting the id from the drop down menu next to "Collection," and hitting the search button. Performing that search with `bplscas` selected, as of this writing showed 7,029 results.

We can also search the Archive using the Python module that we installed,[94] and doing so makes it easier to iterate over all the items in the collection for purposes of further inspection and downloading.

For example, let's modify the sample code from the module's documentation to see if we can tell, with Python, how many items are in the digital Antislavery Collection. The sample code looks something like what you see below. The only difference is that instead of importing only the `search_items` module from `internetarchive`, we are going to import the whole library.

```
import internetarchive
search = internetarchive.search_items('collection:nasa')
print search.num_found
```

All we should need to modify is the collection identifier, from `nasa` to `bplscas`. After starting your computer's Python interpreter, try entering each of the above lines, followed by enter, but modify the collection id in the second command:

```
search = internetarchive.search_items('collection:bplscas')
```

After hitting enter on the print command, you should see a number that matches the number of results you saw when doing the advanced search for the collection in the browser.

Accessing an IA Item in Python

The `internetarchive` module also allows you to access individual items using their identifiers. Let's try that using the documentation's sample code,[95] modifying it in order to get the Douglass letter we discussed earlier.

If you are still at your Python interpreter's command prompt, you don't need to `import internetarchive` again. Since we imported the whole

[91] 'eBooks and Texts', *Internet Archive*: https://archive.org/details/texts

[92] Alexis Rossi, 'How Archive.org items are structured', *Internet Archive Blogs* (31 March 2011): http://blog.archive.org/2011/03/31/how-archive-org-items-are-structured/

[93] 'Advanced Search', *Internet Archive*: https://archive.org/advancedsearch.php

[94] 'Searching from Python', *Python*: https://pypi.python.org/pypi/internetarchive#searching-from-python

[95] 'Downloading from Python', *Python*: https://pypi.python.org/pypi/internetarchive#downloading-from-python

module, we also need to modify the sample code so that our interpreter will know that get_item is from the internetarchive module. We also need to change the sample identifier stairs to our item identifier, *lettertowilliaml00doug* (note that the character before the two zeroes is a lowercase L, not the number 1):

```
item = internetarchive.get_item('lettertowilliaml00doug')
item.download()
```

Enter each of those lines in your interpreter, followed by enter. Depending on your Internet connection speed, it will now probably take a minute or two for the command prompt to return, because your computer is downloading all of the files associated with that item, including some pretty large images. But when it's done downloading, you should be see a new directory on your computer whose name is the item identifier. To check, first exit your Python interpreter:

```
exit()
```

Then list the contents of the current directory to see if a folder now appears named lettertowilliaml00doug. If you list the contents of that folder, you should see a list of files similar to this:

```
39999066767938.djvu
39999066767938.epub
39999066767938.gif
39999066767938.pdf
39999066767938_abbyy.gz
39999066767938_djvu.txt
39999066767938_djvu.xml
39999066767938_images.zip
39999066767938_jp2.zip
39999066767938_scandata.xml
lettertowilliaml00doug_archive.torrent
lettertowilliaml00doug_dc.xml
lettertowilliaml00doug_files.xml
lettertowilliaml00doug_marc.xml
lettertowilliaml00doug_meta.mrc
lettertowilliaml00doug_meta.xml
lettertowilliaml00doug_metasource.xml
```

Now that we know how to use the Search and Item functions in the internetarchive module, we can turn to thinking about how to make this process more effective for downloading lots of information from the collection for further analysis.

Downloading MARC Records from a Collection

Downloading one item is nice, but what if we want to look at thousands of items in a collection? We're in luck, because the internetarchive module's Search function allows us to iterate over all the results in a search.

To see how, let's first start our Python interpreter again. We'll need to import our module again, and perform our search again:

```
import internetarchive
search = internetarchive.search_items('collection:bplscas')
```

Now let's enter the documentation's sample code for printing out the item identifier of every item returned by our search:

```
for result in search:
   print result['identifier']
```

Note that after entering the first line, your Python interpreter will automatically print an ellipsis on line two. This is because you have started a *for loop,* and Python is expecting there to be more. It wants to know what you want to do for each result in the search. That's also why, once you hit enter on the second line, you'll see a third line with another ellipsis, because Python doesn't know whether you are finished telling it what to do with each result. Hit enter again to end the for loop and execute the command.

You should now see your terminal begin to print out the identifiers for each result returned by our *bplscas search*---in this case, all 7,029 of them! You can interrupt the print out by hitting `Ctrl-C` on your keyboard, which will return you to the prompt.

If you didn't see identifiers printing out to your screen, but instead saw an error like this, you may have forgotten to enter a few spaces before your print command:

```
for result in search:
   print result['identifier']
File "", line 2
   print result['identifier']
      ^
IndentationError: expected an indented block
```

Remember that whitespace matters in Python, and you need to indent the lines in a for loop so that Python can tell which command(s) to perform on each item in the loop.

Understanding the for loop

The *for loop,* expressed in plain English, tells Python to do something to each thing in a collection of things. In the above case, we printed the identifier for each result in the results of our collection search. Two additional points about the *for loop:*

First, the word we used after `for` is what's called a *local variable* in Python. It serves as a placeholder for whatever instance or item we are going to be working with inside the loop. Usually it makes sense to pick a name that describes what kind of thing we are working with—in this case, a search result—but we could have used other names in place of that one. For example, try running the above for loop again, but substitute a different name for the local variable, such as:

```
for item in search:
   print item['identifier']
```

You should get the same results.

The second thing to note about the *for loop* is that the indented block could could have contained other commands. In this case, we printed each individual search result's identifier. But we could have chosen to do, for each result, anything that we could do to an individual Internet Archive item.

For example, earlier we downloaded all the files associated with the item *lettertowilliaml00doug.* We could have done that to each item returned by our search by changing the line `print result['identifier']` in our *for loop* to `result.download()`.

We probably want to think twice before doing that, though—downloading all the files for each of the 7,029 items in the bplscas collection is a lot of files. Fortunately, the download function in the `internetarchive` module also allows you to download specific files associated with an item.[96] If we had only wanted to download the MARC XML record associated with a particular item, we could have instead done this:

```
item = internetarchive.get_item('lettertowilliaml00doug')
marc = item.get_file('lettertowilliaml00doug_marc.xml')
marc.download()
```

Because Internet Archive item files are named according to specific rules,[97] we can also figure out the name of the MARC file we want just by knowing the item's unique identifier. And armed with that knowledge, we can proceed to …

Download All MARC XML Files from a Collection

For the next section, we're going to move from using the Python shell to writing a Python script that downloads the MARC record from each item in the BPL Antislavery Collection. Try putting this script into Komodo or your preferred text editor:

[96] 'Downloading from Python', *Python*: https://pypi.python.org/pypi/internetarchive#downloading-from-python

[97] 'Frequently Asked Questions', *Internet Archive*: https://archive.org/about/faqs.php#140

```
#!/usr/bin/python

import internetarchive

search = internetarchive.search_items('collection:bplscas')

for result in search:
    itemid = result['identifier']
    item = internetarchive.get_item(itemid)
    marc = item.get_file(itemid + '_marc.xml')
    marc.download()
    print "Downloading " + itemid + " ..."
```

This script looks a lot like the experiments we have done above with the Frederick Douglass letter, but since we want to download the MARC record for each item returned by our collection search, we are using an itemid variable to account for the fact that the identifier and filename will be different for each result.

Before running this script (which, I should note, is going to download thousands of small XML files to your computer), make a directory where you want those MARC records to be stored and place the above script in that directory. Then run the script from within the directory so that the files will be downloaded in an easy-to-find place.

(Note that if you receive what looks like a `ConnectionError` on your first attempt, check your Internet connection, wait a few minutes, and then try running the script again.)

If all goes well, when you run your script, you should see the program begin to print out status updates telling you that it is downloading MARC records. But allowing the script to run its full course will probably take a couple of hours, so let's stop the script and look a little more closely at ways to improve it. Pressing `Ctrl-C` while in your terminal window should make the script stop.

Building Error Reporting into the Script

Since downloading all of these records will take some time, we are probably going to want to walk away from our computer for a while. But the chances are high that during those two hours, something could go wrong that would prevent our script from working.

Let's say, for example, that we had forgotten that we already downloaded an individual file into this directory. Or maybe your computer briefly loses its Internet connection, or some sort of outage happens on the Internet Archive server that prevents the script from getting the file it wants.

In those and other error cases, Python will raise an "exception" telling you what the problem is. Unfortunately, an exception will also crash your script instead of continuing on to the next item.

To prevent this, we can use what's called a *try statement* in Python, which does exactly what it sounds like. The statement will try to execute a certain snippet of code until it hits an exception, in which case you can give it some other code to execute instead. You can read more about handling exceptions in the Python documentation,[98] but for now let's just update our above script so that it looks like this:

```
#!/usr/bin/python

import internetarchive
import time

error_log = open('bpl-marcs-errors.log', 'a')

search = internetarchive.search_items('collection:bplscas')

for result in search:
    itemid = result['identifier']
    item = internetarchive.get_item(itemid)
    marc = item.get_file(itemid + '_marc.xml')
    try:
        marc.download()
    except Exception as e:
        error_log.write('Could not download ' + itemid + ' because of error:
 %s\n' % e)
        print "There was an error; writing to log."
    else:
        print "Downloading " + itemid + " ..."
        time.sleep(1)
```

The main thing we've added here, after our module import statements, is a line that opens a text file called `bpl-marcs-errors.log` and prepares it to have text appended to it. We are going to use this file to log exceptions that the script raises. The *try statement* that we have added to our *for loop* will attempt to download the MARC record. If it can't, it will write a descriptive statement about what went wrong to our log file. That way we can go back to the file later and identify which items we will need to try to download again. If the try clause succeeds and can download the record, then the script will execute the code in the *else* clause.

One other thing we have added, upon successful download, is this line:

```
time.sleep(1)
```

This line uses the `time` module that we are now importing at the beginning to tell our script to pause for one second before proceeding, which is basically just a way for us to be nice to Internet Archive's servers by not clobbering them every millisecond or so with a request.

Try updating your script to look like the above lines, and run it again in the directory where you want to store your MARC files. Don't be surprised if

[98] 'Handling Exceptions', *Python*: https://docs.python.org/2/tutorial/errors.html#handling-exceptions

you immediately encounter a string of error messages; that means the script is doing what it's supposed to do! Calmly go into your text editor, while leaving the script running, and open the `bpl-marcs-errors.log` to see what exceptions have been recorded there. You'll probably see that the script raised the exception "File already exists" for each of the files that you had already downloaded when running our earlier, shorter program.

If you leave the program running for a little while, the script will eventually get to items that you have not already downloaded and resume collecting your MARCs!

Scraping Information from a MARC Record

Once your download script has completed, you should find yourself in the possession of nearly 7,000 detailed MARC XML records about items in the Anti-Slavery Collection (or whichever other collection you may have downloaded instead; the methods above should work on any collection whose items have MARC files attached to them).

Now what?

The next step depends on what sort of questions about the collection you want to answer. The MARC formatting language captures a wealth of data about an item, as you can see if you return to the MARC XML record for the Frederick Douglass letter mentioned at the outset.[99]

Notice, for example, that the Douglass letter contains information about the place where the letter was written in the *datafield* that is tagged *260,* inside the subfield coded *a.* The person who prepared this MARC record knew to put place information in that specific field because of rules specified for the 260 datafield[100] by the MARC standards.[101]

That means that it should be possible for us to look inside all of the MARC records we have downloaded, grab the information inside of datafield *260,* subfield *a,* and make a list of every place name where items in the collection were published.

To do this, we'll use the other helpful Python module that we downloaded with `pip` at the beginning: pymarc.[102]

That module makes it easy to get information out of subfields. Assuming that we have a MARC record prepared for parsing by the module assigned to the variable record, we could get the information about publication place names this way:

[99] 'Letter to William Lloyd Garrison – MARC XML', *Internet Archive*: http://ia801703.us.archive.org/29/items/lettertowilliaml00doug/lettertowilliaml00doug_marc.xml

[100] '260 – Publication, Distribution, etc. (Imprint) (R)' *Library of Congress*: http://www.loc.gov/marc/bibliographic/bd260.html

[101] 'MARC Standards', *Library of Congress*: http://www.loc.gov/marc/

[102] 'pymarc', *Github*: https://github.com/edsu/pymarc

```
place_of_pub = record['260']['a']
```

The documentation for pymarc is a little less complete than that for the Internet Archive, especially when it comes to parsing XML records. But a little rooting around in the source code for the module reveals some functions that it provides for working with MARC XML records.[103] One of these, called map_xml() is described this way:

```
def map_xml(function, *files):
    """
    map a function onto the file, so that for each record that is
    parsed the function will get called with the extracted record

    def do_it(r):
    print r

    map_xml(do_it, 'marc.xml')
    """
```

Translated into plain English, this function means that we can take an XML file containing MARC data (like the nearly 7,000 we now have on our computer), pass it to the map_xml function in the pymarc module, and then specify another function (that we will write) telling our program what to do with the MARC data retrieved from the XML file. In rough outline, our code will look something like this:

```
import pymarc

def get_place_of_pub(record):
    place_of_pub = record['260']['a']
    print place_of_pub

pymarc.map_xml(get_place_of_pub, 'lettertowilliam00doug_marc.xml')
```

Try saving that code to a script and running it from a directory where you already have the Douglass letter XML saved. If all goes well, the script should spit out this:

```
Belfast, [Northern Ireland],
```

Voila! Of course, this script would be much more useful if we scraped the place of publication from every letter in our collection of MARC records. Putting together what we've learned from earlier in the lesson, we can do that with a script that looks like this:

[103] ‘marcxml.py’, Github: https://github.com/edsu/pymarc/blob/master/pymarc/marcxml.py

```
#!/usr/bin/python

import os
import pymarc

path = '/path/to/dir/with/xmlfiles/'

def get_place_of_pub(record):
    try:
        place_of_pub = record['260']['a']
        print place_of_pub
    except Exception as e:
        print e

for file in os.listdir(path):
    if file.endswith('.xml'):
        pymarc.map_xml(get_place_of_pub, path + file)
```

This script modifies our above code in several ways. First, it uses a *for loop* to iterate over each file in our directory. In place of the `internetarchive` search results that we iterated over in our first part of this lesson, we iterate over the files returned by `os.listdir(path)` which uses the built-in Python module `os` to list the contents of the directory specified in the path variable, which you will need to modify so that it matches the directory where you have downloaded all of your MARC files.

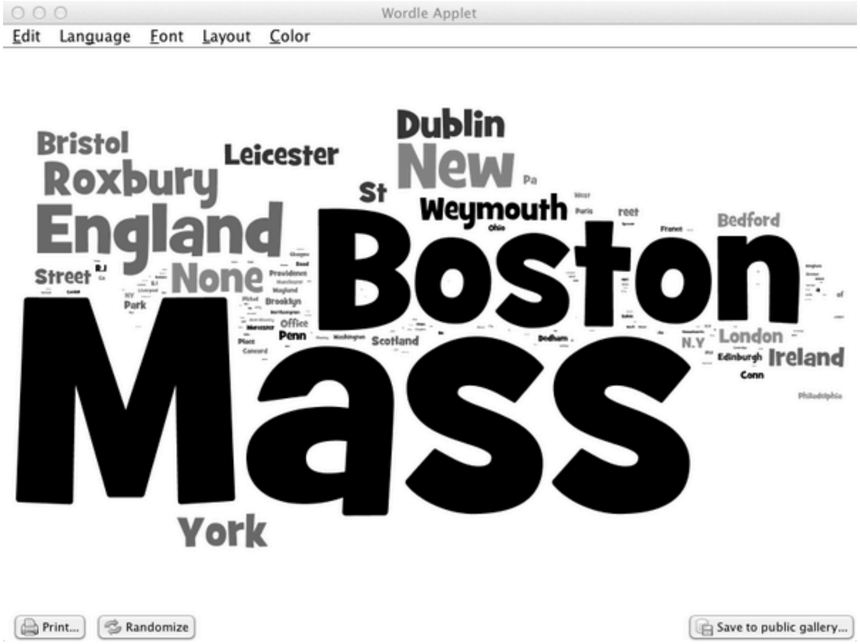
We have also added some error handling to our `get_place_of_pub()` function to account for the fact that some records may (for whatever reason) not contain the information we are looking for. The function will try to print the place of publication, but if this raises an Exception, it will print out the information returned by the Exception instead. In this case, if the try statement failed, the exception will probably print `None`. Understanding why is a subject for another lesson on Python Type errors, but for now the None printout is descriptive enough of what happened, so it could be useful to us.

Try running this script. If all goes well, your screen should fill with a list of the places where these letters were written. If that works, try modifying your script so that it saves the place names to a text file instead of printing them to your screen. You could then use the Counting Frequencies with Python lesson by Turkel and Crymble,[104] to figure out which place names are most common in the collection. You could work with the place names to find coordinates that could be placed on a map using the 'Intro to Google Maps and Google Earth' tutorial by Clifford *et al.*[105]

[104] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

[105] Jim Clifford, Josh MacFadyen, and Daniel Macfarlane, 'Intro to Google Maps and Google Earth', *The Programming Historian* (2013).

Or, to get a very rough visual sense of the places where letters were written, you could do what I've done below and simply make a Wordle word cloud of the text file.[106]



Wordle wordcloud of places of publication for abolitionist letters

Of course, to make such techniques useful would require more cleaning of your data, such as those discussed by Laura Turner O'Hara in 'Cleaning OCR'd text with Regular Expressions'.[107] And other applications of this lesson may prove more useful. For example, working with the MARC data fields for personal names, you could create a network of correspondents. Or you could analyze which subjects are common in the MARC records. Now that you have the MARC records downloaded and can use `pymarc` to extract information from the fields, the possibilities can multiply rapidly!

About the Author

Caleb McDaniel is an associate professor of history at Rice University.

[106] 'Wordle': http://www.wordle.net/

[107] Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).

12. Using SPARQL to access Linked Open Data

Matthew Lincoln – 2015

Lesson Goals

This lesson explains why many cultural institutions are adopting graph databases, and how researchers can access these data though the query language called SPARQL.

Table of contents

Graph Databases, RDF, and Linked Open Data

Many cultural institutions now offer access to their collections information through web Application Programming Interfaces.[108] While these APIs are a powerful way to access individual records in a machine-readable manner, they are not ideal for cultural heritage data because they are structured to work for a predetermined set of queries. For example, a museum may have information on donors, artists, artworks, exhibitions, and provenance, but

[108] For example, see: Amanda Morton, 'Intro to the Zotero API', *The Programming Historian* (2013).

its web API may offer only object-wise retrieval, making it difficult or impossible to search for associated data about donors, artists, provenance, etc. This structure is great if you come looking for information about particular objects. However, it makes it difficult to aggregate information about every artist or donor that happens to be described in the dataset as well.

RDF databases are well-suited to expressing complex relationships between many entities, like people, places, events, and concepts tied to individual objects. These databases are often referred to as "graph" databases because they structure information as a graph or network, where a set of resources, or nodes, are connected together by edges that describe the relationships between each resource.

Because RDF databases support the use of URLs (weblinks), they can be made available online and linked to other databases, hence the term "Linked Open Data". Major art collections including the British Museum,[109] Europeana,[110] the Smithsonian American Art Museum,[111] and the Yale Center for British Art[112] have published their collections data as LOD. The Getty Vocabulary Program,[113] has also released their series of authoritative databases on geographic place names, terms for describing art and architecture, and variant spellings of artist names, as LOD.

SPARQL is the language used to query these databases. This language is particularly powerful because it does not presuppose the perspectives that users will bring to the data. A query about objects and a query about donors is basically equivalent to such a database. Unfortunately, many tutorials on SPARQL use extremely simplified data models that don't resemble the more complex datasets released by cultural heritage institutions. This tutorial gives a crash course on SPARQL using a dataset that a humanist might actually find in the wilds of the Internet. In this tutorial, we will learn how to query the British Museum Linked Open Data collection.

RDF in brief

RDF represents information in a series of three-part "statements" that comprise a subject, predicate, and an object, e.g.:

```
<The Nightwatch> <was created by> <Rembrandt van Rijn> .
```

(Note that just like any good sentence, they each have a period at the end.)

[109] 'British Museum Semantic Web Collection Online', *British Museum*: http://collection.britishmuseum.org/

[110] 'Europeana Linked Open Data' *Europeana Labs*: http://labs.europeana.eu/api/linked-open-data-introduction

[111] 'Smithsonian American Art Museum': http://americanart.si.edu/

[112] 'Yale Centre for British Art': http://britishart.yale.edu/collections/using-collections/technology/linked-open-data

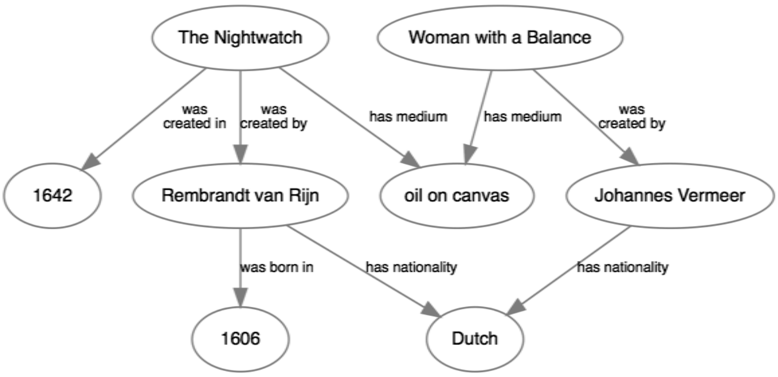
[113] 'The Getty Vocabularies': http://vocab.getty.edu/

Here, the subject `<The Nightwatch>` and the object `<Rembrandt van Rijn>` can be thought of as two nodes of the graph, with the predicate `<was created by>` defining an edge between them. (Technically, `<was created by>` can, in other queries, be treated as an object or subject itself, but that is beyond the scope of this tutorial.)

A pseudo-RDF database might contain interrelated statements like these:

```
...
<The Nightwatch> <was created by> <Rembrandt van Rijn> .
<The Nightwatch> <was created in> <1642> .
<The Nightwatch> <has medium> <oil on canvas> .
<Rembrandt van Rijn> <was born in> <1606> .
<Rembrandt van Rijn> <has nationality> <Dutch> .
<Johannes Vermeer> <has nationality> <Dutch> .
<Woman with a Balance> <was created by> <Johannes Vermeer> .
<Woman with a Balance> <has medium> <oil on canvas> .
...
```

If we were to visualize these statements as nodes and edges within network graph, it would appear like so:



A network visualization of the pseud-RDF shown above. Arrows indicate the 'direction' of the predicate. For example, that 'Woman with a Balance' was created by Vermeer', and not the other way around.

A traditional relational database might split attributes about artworks and attributes about artists into separate tables. In an RDF/graph database, all these data points belong to the same interconnected graph, which allows users maximum flexibility in deciding how they wish to query it.

Searching RDF with SPARQL

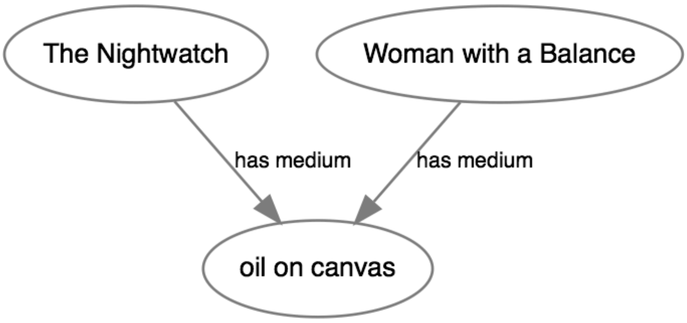
SPARQL lets us translate heavily interlinked, graph data into normalized, tabular data with rows and columns you can open in programs like Excel, or import into a visualization suite such as plot.ly or Palladio.[114]

[114] 'Plotly': https://plot.ly/; 'Palladio': http://palladio.designhumanities.org/

It is useful to think of a SPARQL query as a Mad Lib - a set of sentences with blanks in them.[115] The database will take this query and find every set of matching statements that correctly fill in those blanks, returning the matching values to us as a table. Take this SPARQL query:

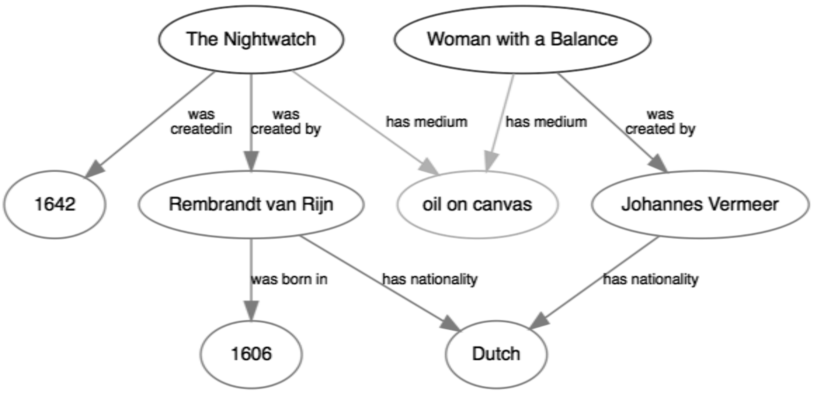
```
SELECT ?painting
WHERE {
  ?painting <has medium> <oil on canvas> .
}
```

`?painting` in this query stands in for the node (or nodes) that the database will return. On receiving this query, the database will search for all values of `?painting` that properly complete the RDF statement `<has medium> <oil on canvas> .`:



A visualization of what our query is looking for

When the query runs against the full database, it looks for the subjects, predicates, and objects that match this statement, while excluding the rest of the data:



A visualization of the SPARQL query.

[115] 'Mad Libs', *Wikipedia*: https://en.wikipedia.org/wiki/Mad_Libs

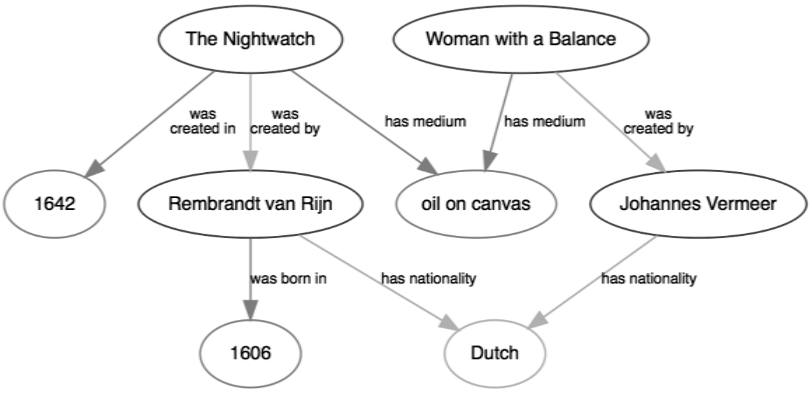
And our results might look like this table:

painting
The Nightwatch
Woman with a Balance

What makes RDF and SPARQL powerful is the ability to create complex queries that reference many variables at a time. For example, we could search our pseudo-RDF database for paintings by any artist who is Dutch:

```
SELECT ?artist ?painting
WHERE {
  ?artist <has nationality> <Dutch> .
  ?painting <was created by> ?artist .
}
```

Here we've introduced a second variable, `?artist`. The RDF database will return all matching combinations of `?artist` and `?painting` that fulfill both of these statements.



A visualization of the SPARQL query.

artist	painting
Rembrandt van Rijn	The Nightwatch
Johannes Vermeer	Woman with a Balance

URIs and Literals

So far, we have been looking at a toy representation of RDF that uses easy-to-read text. However, RDF is primarily stored as URIs (Uniform Resource Identifiers) that separate conceptual entities from their plain-English (or other language!) labels. (Note that a URL, or Uniform Resource Locator, is a URI for a resource that is accessible on the web) In real RDF, our original statement:

```
<The Nightwatch>   <was created by>   <Rembrandt van Rijn> .
```

would more likely look something like this:

```
<http://data.rijksmuseum.nl/item/8909812347> <http://purl.org/dc/terms/creat
or>  <http://dbpedia.org/resource/Rembrandt>.
```

N.B. the Rijksmuseum has not (yet) built their own Linked Data site, so the URI in this query is just for demo purposes.

In order to get the human-readable version of the information represented by each of these URIs, what we're really doing is just retrieving more RDF statements. Even the predicate in that statement has its own literal label:

```
<http://data.rijksmuseum.nl/item/8909812347> <http://purl.org/dc/terms/title
> "The Nightwatch" .

<http://purl.org/dc/terms/creator> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#label> "was created by" .

<http://dbpedia.org/resource/Rembrandt> <http://xmlns.com/foaf/0.1/name> "Re
mbrandt van Rijn" .
```

You will notice that, unlike the URIs in the query that are surrounded by `<>`, the *objects* of these statements are just strings of text within quotation marks, known as *literals*. Literals are unlike URIs in that they represent values, rather than references. For example:

`<http://dbpedia.org/resource/Rembrandt>` represents an entity that may reference (and be referenced by) any number of other statements (say, birth dates, students, or family members), while the text string `"Rembrandt van Rijn"` stands only for itself. Literals do not point to other nodes in the graph, and they can only ever be objects in an RDF statement. Other literal values in RDF include dates and numbers.

See the *predicates* in these statements, with domain names like `purl.org`, `w3.org`, and `xmlns.com`? These are some of the many providers of ontologies that help standardize the way we describe relationships between bits of information like "title", "label", "creator", or "name". The more RDF/LOD that you work with, the more of these providers you'll find.

URIs can become unwieldy when composing SPARQL queries, which is why we'll use *prefixes*. These are shortcuts that allow us to skip typing out entire long URIs. For example, remember that predicate for retrieving the title of the *Nightwatch*, `<http://purl.org/dc/terms/title>`? With these prefixes, we just need to type `dct:title` whenever we need to use a `purl.org` predicate. `dct:` stands in for `http://purl.org/dc/terms/`, and `title` just gets pasted onto the end of this link.

For example, with the prefix `PREFIX rkm: <http://data.rijksmuseum.nl/>`, appended to the start of our SPARQL query, `<http://data.rijksmuseum.nl/item/8909812347>` becomes `rkm:item/8909812347` instead.

Many live databases, such as the British Museum database that we will try our first new queries on, already define these prefixes for us, so we won't have to explicitly state them in our queries. However, you should now be able to recognize whenever we use one in a SPARQL query. Also be aware that, prefixes can be arbitrarily assigned with whatever abbreviations you like, different endpoints may use slightly different prefixes for the same namespace (e.g. `dct` vs. `dcterms` for `<http://purl.org/dc/terms/>`).

Terms to review

SPARQL - *Protocol and RDF Query Language* - The language used to query RDF graph databases

RDF - *Resource Description Framework* - A method for structuring data as a graph or network of connected statements, rather than a series of tables.

LOD - *Linked Open Data* - LOD is RDF data published online with dedicated URIs in such a manner than developers can reliably reference it.

statement - Sometimes also called a "triple", an RDF statement is a quantum of knowledge comprising a *subject*, *predicate*, and *object*.

URI - *Uniform Resource Identifier* - a string of characters for identifying a resource. RDF statements use URIs to link various resources together. A URL, or uniform resource locator, is a type of URI that points to resources on the web.

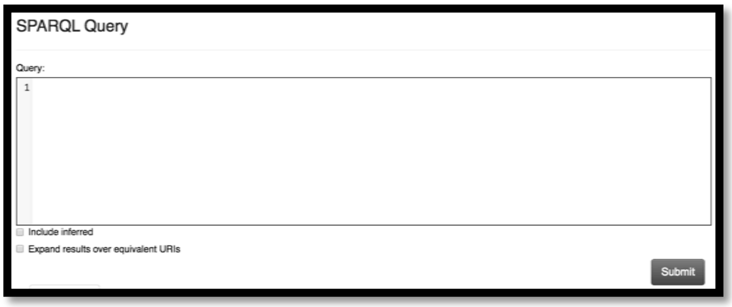
literal - Some objects in RDF statements do not refer to other resources with a URI, but instead convey a value, such as text (`"Rembrandt van Rijn"`), a number (`5`), or a date (`1606-06-15`). These are known as literals.

prefix - In order to simplify SPARQL queries, a user may specify prefixes that act as abbreviations for full URIs. These abbreviations, or **QNames**, are also used in namespaced XML documents.

Real-world queries

All the statements for one object

Let's start our first query using the British Museum SPARQL endpoint.[116] A SPARQL endpoint is a web address that accepts SPARQL queries and returns results. The BM endpoint is like many others: if you navigate to it in a web browser, it presents you with a text box for composing queries.



[116] 'SPARQL Query', *The British Museum*: http://collection.britishmuseum.org/sparql

The BM SPARQL endpoint webpage. For all the queries in this tutorial, make sure that you have left the 'Include inferred' and 'Expand results over equivalent URIs' boxes unchecked.

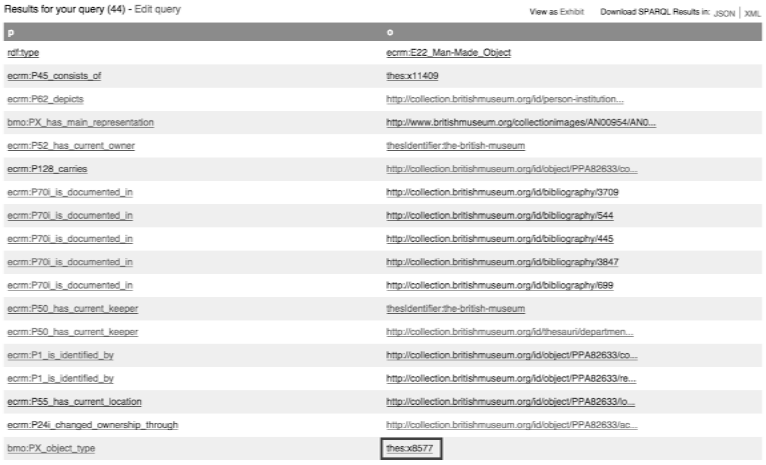
When starting to explore a new RDF database, it helps to look at the relationships that stem from a single example object.[117]

(For each of the following queries, click on the "Run query" link below to see the results. Click on the "Edit query" link to go to a page with the query automatically pasted the query into the BM's endpoint. You can then run it as is, or modify it before requesting the results. Remember when editing the query before running to uncheck the 'Include inferred' box.)

```
SELECT ?p ?o
WHERE {
  <http://collection.britishmuseum.org/id/object/PPA82633> ?p ?o .
}
```

Run query[118] / Edit query[119]

By calling `SELECT ?p ?o` we're asking the database to return the values of `?p` and `?o` as described in the `WHERE {}` command. This query returns every statement for which our example artwork, `<http://collection.britishmuseum.org/id/object/PPA82633>`, is the subject.



Results for your query (44) - Edit query

View as Exhibit   Download SPARQL Results in: JSON | XML

p	o
rdf:type	ecrm:E22_Man-Made_Object
ecrm:P45_consists_of	thes:x11409
ecrm:P62_depicts	http://collection.britishmuseum.org/id/person-institution...
bmo:PX_has_main_representation	http://www.britishmuseum.org/collectionimages/AN00954/AN0...
ecrm:P52_has_current_owner	thesIdentifier:the-british-museum
ecrm:P128_carries	http://collection.britishmuseum.org/id/object/PPA82633/co...
ecrm:P70i_is_documented_in	http://collection.britishmuseum.org/id/bibliography/3709
ecrm:P70i_is_documented_in	http://collection.britishmuseum.org/id/bibliography/544
ecrm:P70i_is_documented_in	http://collection.britishmuseum.org/id/bibliography/445
ecrm:P70i_is_documented_in	http://collection.britishmuseum.org/id/bibliography/3847
ecrm:P70i_is_documented_in	http://collection.britishmuseum.org/id/bibliography/699
ecrm:P50_has_current_keeper	thesIdentifier:the-british-museum
ecrm:P50_has_current_keeper	http://collection.britishmuseum.org/id/thesauri/departmen...
ecrm:P1_is_identified_by	http://collection.britishmuseum.org/id/object/PPA82633/co...
ecrm:P1_is_identified_by	http://collection.britishmuseum.org/id/object/PPA82633/re...
ecrm:P55_has_current_location	http://collection.britishmuseum.org/id/object/PPA82633/lo...
ecrm:P24i_changed_ownership_through	http://collection.britishmuseum.org/id/object/PPA82633/ac...
bmo:PX_object_type	thes:x8577

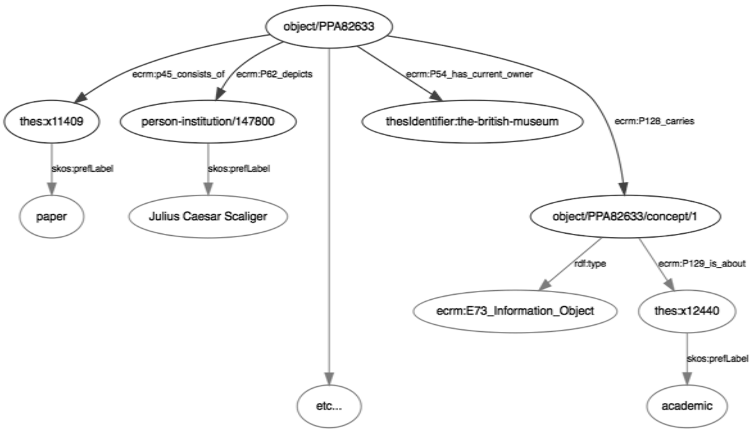
[117] The Example Object is: 'PPA82633', *The British Museum*: http://collection.britishmuseum.org/resource?uri=http://collection.britishmuseum.org/id/object/PPA82633

[118] To run the query, visit: http://collection.britishmuseum.org/sparql?query=SELECT+*%0D%0AWHERE+{%0D%0A++%3Chttp%3A%2F%2Fcollection.britishmuseum.org%2Fid%2Fobject%2FPPA82633%3E+%3Fp+%3Fo+.%0D%0A++}&_implicit=false&_equivalent=false&_form=%2Fsparql

[119] To edit the query, visit: http://collection.britishmuseum.org/sparql?sample=SELECT+*%0D%0AWHERE+{%0D%0A++%3Chttp%3A%2F%2Fcollection.britishmuseum.org%2Fid%2Fobject%2FPPA82633%3E+%3Fp+%3Fo+.%0D%0A++}

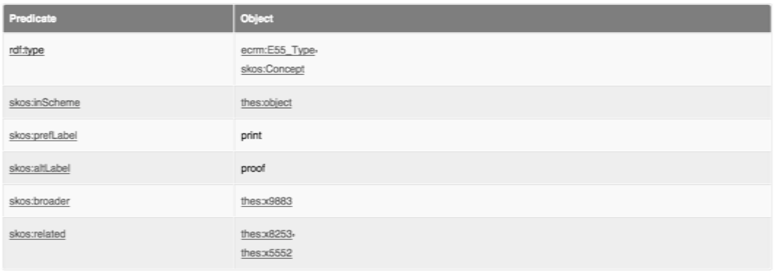
An initial list of all the predicates and objects associated with one artwork in the British Museum.

The BM endpoint formats the results table with hyperlinks for every variable that is itself an RDF node, so by clicking on any one of these links you can shift to seeing all the predicates and objects for that newly-selected node. Note that BM automatically includes a wide range of SPARQL prefixes in its queries, so you will find many hyperlinks are displayed in their abbreviated versions; if you mouse over them your browser will display their unabbreviated URIs.



Visualizing a handful of the nodes returned by the first query to the BM. Additional levels in the hierarchy are included as a preview of how this single print connects to the larger BM graph.

Let's find out how they store the object type information: look for the predicate `<bmo:PX_object_type>` (highlighted in the figure above) and click on the link for `thes:x8577` to navigate to the node describing the particular object type "print":



Predicate	Object
rdf:type	ecrm:E55_Type, skos:Concept
skos:inScheme	thes:object
skos:prefLabel	print
skos:altLabel	proof
skos:broader	thes:x9883
skos:related	thes:x8253, thes:x5552

The resource page for thes:x8577 ('print') in the British Museum LOD

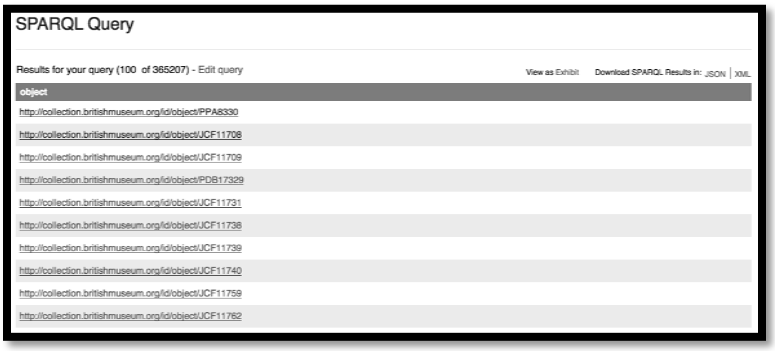
You'll note how this node has an plain-text label, as well as ties to related artwork type nodes within the database.

Complex queries

To find other objects of the same type with the preferred label "print", we can call this query:

```
SELECT ?object
WHERE {

  # Search for all values of ?object that have a given "object type"
  ?object bmo:PX_object_type ?object_type .

  # That object type should have the label "print"
  ?object_type skos:prefLabel "print" .
}
```

Run query[120] / Edit query[121]



SPARQL Query

Results for your query (100 of 365207) - Edit query

View as Exhibit   Download SPARQL Results in: JSON | XML

object
http://collection.britishmuseum.org/id/object/PPA8330
http://collection.britishmuseum.org/id/object/JCF11708
http://collection.britishmuseum.org/id/object/JCF11709
http://collection.britishmuseum.org/id/object/PDB17329
http://collection.britishmuseum.org/id/object/JCF11731
http://collection.britishmuseum.org/id/object/JCF11738
http://collection.britishmuseum.org/id/object/JCF11739
http://collection.britishmuseum.org/id/object/JCF11740
http://collection.britishmuseum.org/id/object/JCF11759
http://collection.britishmuseum.org/id/object/JCF11762

A one-column table returned by our query for every object with type 'print'

Remember that, because `"print"` here is a *literal*, we enclose it within quotation marks in our query. When you include literals in a SPARQL query, the database will only return *exact* matches for those values.

Note that, because `?object_type` is not present in the `SELECT` command, it will not show up in the results table. However, it is essential to structuring our query, because it connects the dots from `?object` to the label `"print"`.

FILTER

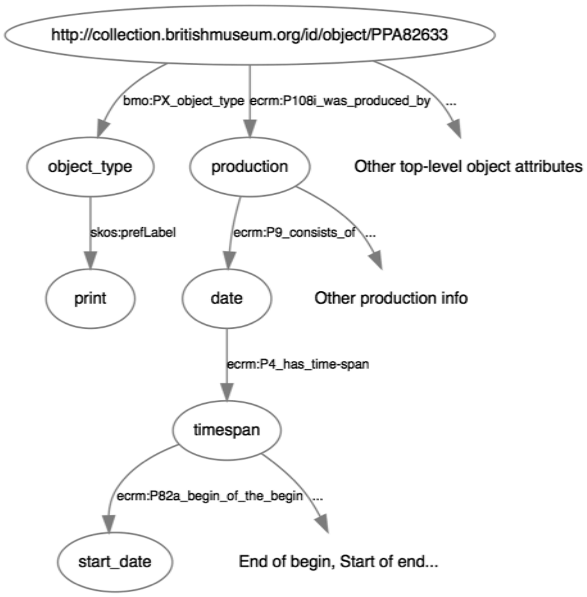
In the previous query, our SPARQL query searched for an exact match for the object type with the text label "print". However, often we want to match literal values that fall within a certain range, such as dates. For this, we'll use the `FILTER` command.

To find URIs for all the prints in the BM created between 1580 and 1600, we'll need to first figure out where the database stores dates in relationship

[120] To run the query, visit: http://collection.britishmuseum.org/sparql?query=SELECT+%3Fobject%0D%0AWHERE+{%0D%0A++%3Fobject+bmo%3APX_object_type+%3Fobject_type+.%0D%0A++%3Fobject_type+skos%3AprefLabel+%22print%22+.%0D%0A}&_implicit=false&implicit=false&_equivalent=false&_form=%2Fsparql (note, you can also find this as a clickable link on the online version of the tutorial)

[121] To edit the query, visit: http://collection.britishmuseum.org/sparql?sample=PREFIX+bmo%3A+%3Chttp%3A%2F%2Fcollection.britishmuseum.org%2Fid%2Fontology%2F%3E%0APREFIX+skos%3A+%3Chttp%3A%2F%2Fwww.w3.org%2F2004%2F02%2Fskos%2Fcore%23%3E%0ASELECT+%3Fobject%0D%0AWHERE+{%0D%0A++%3Fobject+bmo%3APX_object_type+%3Fobject_type+.%0D%0A++%3Fobject_type+skos%3AprefLabel+%22print%22+.%0D%0A} (note, you can also find this as a clickable link on the online version of the tutorial)

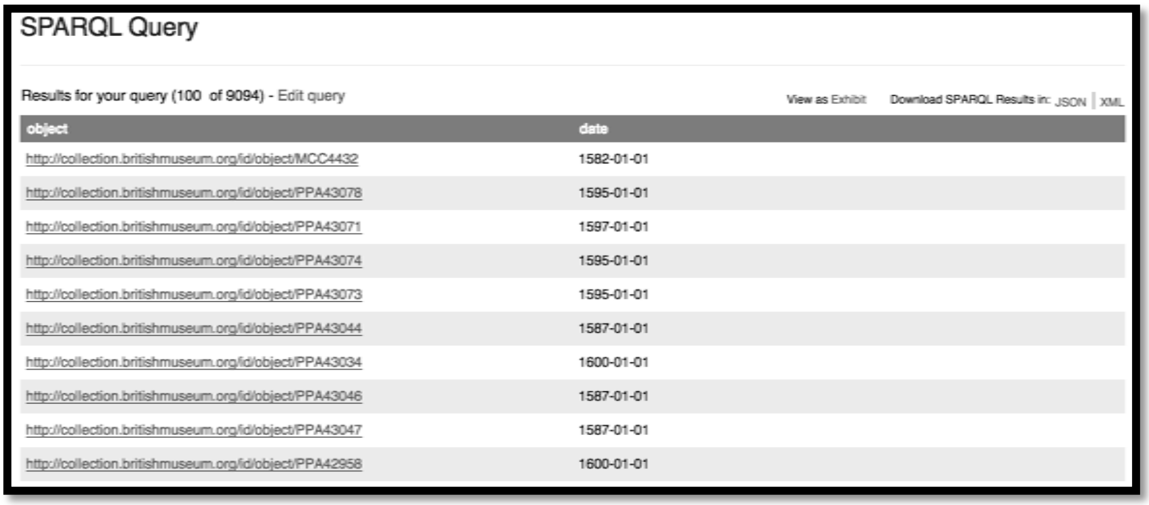
to the object node, and then add references to those dates in our query. Similar to the way that we followed a single link to determine an object type, we must hop through several nodes to find the production dates associated with a given object:


http://collection.britishmuseum.org/id/object/PPA82633
bmo:PX_object_type
ecrm:P108i_was_produced_by
...
object_type
production
Other top-level object attributes
skos:prefLabel
ecrm:P9_consists_of
...
print
date
Other production info
ecrm:P4_has_time-span
timespan
ecrm:P82a_begin_of_the_begin
...
start_date
End of begin, Start of end...

Visualizing part of the British Museum's data model where production dates are connected to objects.

```
# Return object links and creation date
SELECT ?object ?date
WHERE {

  # We'll use our previous command to search only for
  # objects of type "print"
  ?object bmo:PX_object_type ?object_type .
  ?object_type skos:prefLabel "print" .

  # We need to link though several nodes to find the
  # creation date associated with an object
  ?object ecrm:P108i_was_produced_by ?production .
  ?production ecrm:P9_consists_of ?date_node .
  ?date_node ecrm:P4_has_time-span ?timespan .
  ?timespan ecrm:P82a_begin_of_the_begin ?date .

  # As you can see, we need to connect quite a few dots
  # to get to the date node! Now that we have it, we can
  # filter our results. Because we are filtering by date,
  # we must attach the tag ^^xsd:date after our date strings.
  # This tag tells the database to interpret the string
  # "1580-01-01" as the date 1 January 1580.

  FILTER(?date >= "1580-01-01"^^xsd:date &&
         ?date <= "1600-01-01"^^xsd:date)
}
```

Run query / Edit query[122]



SPARQL Query

Results for your query (100 of 9094) - Edit query

object	date
http://collection.britishmuseum.org/id/object/MCC4432	1582-01-01
http://collection.britishmuseum.org/id/object/PPA43078	1595-01-01
http://collection.britishmuseum.org/id/object/PPA43071	1597-01-01
http://collection.britishmuseum.org/id/object/PPA43074	1595-01-01
http://collection.britishmuseum.org/id/object/PPA43073	1595-01-01
http://collection.britishmuseum.org/id/object/PPA43044	1587-01-01
http://collection.britishmuseum.org/id/object/PPA43034	1600-01-01
http://collection.britishmuseum.org/id/object/PPA43046	1587-01-01
http://collection.britishmuseum.org/id/object/PPA43047	1587-01-01
http://collection.britishmuseum.org/id/object/PPA42958	1600-01-01

ALL BM prints made between 1580 and 1600

Aggregation

So far we have only used the `SELECT` command to return a table of objects. However, SPARQL allows us to do more advanced analysis such as grouping, counting, and sorting.

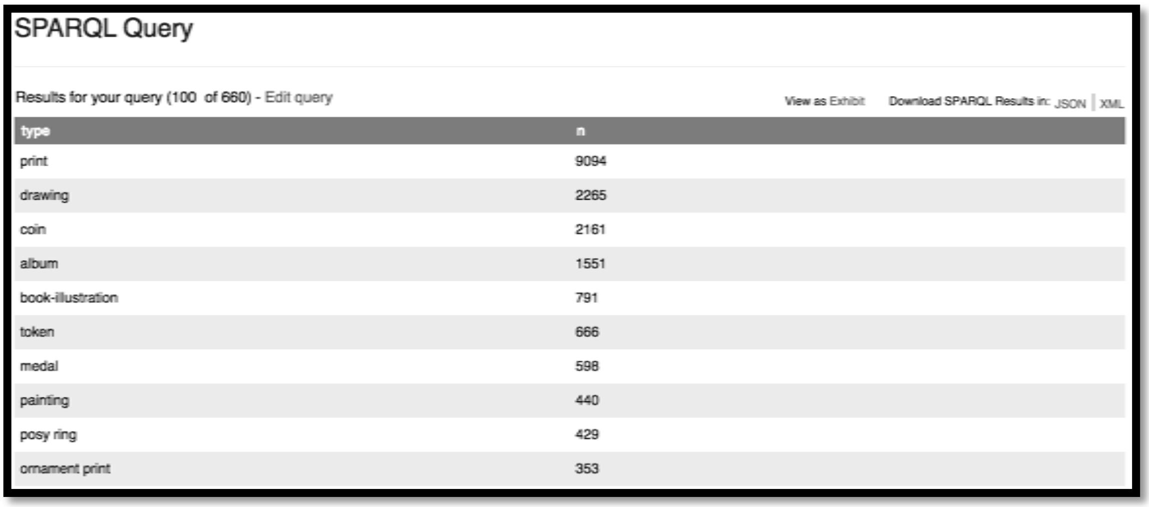
Say we would like to keep looking at objects made between 1580 and 1600, but we want to understand how many objects of each type the BM has in its collections. Instead of limiting our results to objects of type "print", we will instead use `COUNT` to tally our search results by type.

```
SELECT ?type (COUNT(?type) as ?n)
WHERE {
  # We still need to indicate the ?object_type variable,
  # however we will not require it to match "print" this time

  ?object bmo:PX_object_type ?object_type .
  ?object_type skos:prefLabel ?type .

  # Once again, we will also filter by date
  ?object ecrm:P108i_was_produced_by ?production .
  ?production ecrm:P9_consists_of ?date_node .
  ?date_node ecrm:P4_has_time-span ?timespan .
  ?timespan ecrm:P82a_begin_of_the_begin ?date .
  FILTER(?date >= "1580-01-01"^^xsd:date &&
         ?date <= "1600-01-01"^^xsd:date)
```

[122] The links to run and edit the query can be found in the online version of the tutorial. Trust us, you wouldn't want to have to type it out by hand: http://programminghistorian.org/lessons/graph-databases-and-SPARQL

```
}
# The GROUP BY command designates the variable to tally by,
# and the ORDER BY DESC() command sorts the results by
# descending number.
GROUP BY ?type
ORDER BY DESC(?n)
```

Run query / Edit query[123]



SPARQL Query

Results for your query (100 of 660) - Edit query

type	n
print	9094
drawing	2265
coin	2161
album	1551
book-illustration	791
token	666
medal	598
painting	440
posy ring	429
ornament print	353

Counts of objects by type produced between 1580 and 1600

Linking multiple SPARQL endpoints

Up until now, we have constructed queries that look for patterns in one dataset alone. In the ideal world envisioned by Linked Open Data advocates, multiple databases can be interlinked to allow very complex queries dependent on knowledge present in different locations. However, this is easier said than done, and many endpoints (the BM's included) do not yet reference outside authorities.

One endpoint that does, however, is Europeana's.[124] They have created links between the objects in their database and records about individuals in DBPedia[125] and VIAF,[126] places in GeoNames,[127] and concepts in the Getty Art & Architecture thesaurus. SPARQL allows you to insert `SERVICE` statements that instruct the database to "phone a friend" and run a portion of the query on an outside dataset, using the results to complete the query

[123] The links to run and edit the query can be found in the online version of the tutorial. Trust us, you wouldn't want to have to type it out by hand: http://programminghistorian.org/lessons/graph-databases-and-SPARQL

[124] 'SPARQL Queries' *Europeana*: http://europeana.ontotext.com/sparql

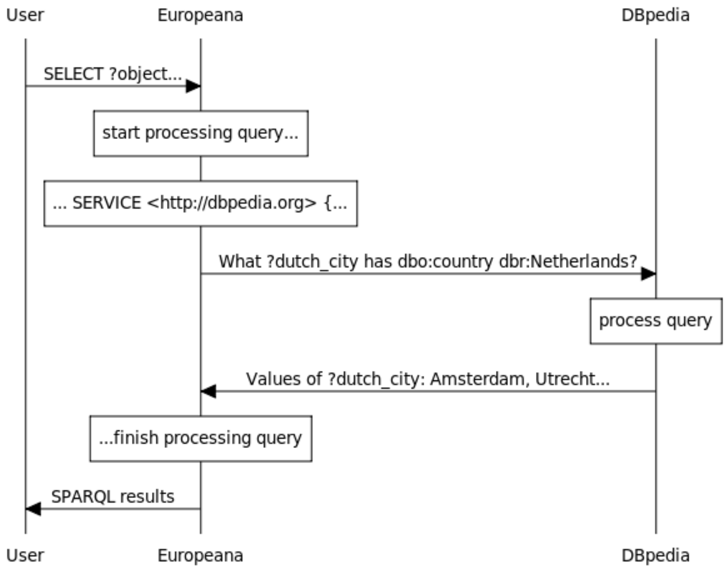
[125] 'DBPedia': http://wiki.dbpedia.org/

[126] 'VIAF: The Virtual International Authority File': https://viaf.org/

[127] 'GeoNames': http://sws.geonames.org/

on the local dataset. While this lesson will go into the data models in Europeana and DBpedia in depth, the following query illustrates how a SELECT statement works. You may run it yourself by copying and pasting the query text into the Europeana endpoint.

```
PREFIX edm:    <http://www.europeana.eu/schemas/edm/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo:    <http://dbpedia.org/ontology/>
PREFIX dbr:    <http://dbpedia.org/resource/>
PREFIX rdaGr2: <http://rdvocab.info/ElementsGr2/>

# Find all ?object related by some ?property to an ?agent born in a
# ?dutch_city
SELECT ?object ?property ?agent ?dutch_city
WHERE {
    ?proxy ?property ?agent .
    ?proxy ore:proxyFor ?object .

    ?agent rdf:type edm:Agent .
    ?agent rdaGr2:placeOfBirth ?dutch_city .

    # ?dutch_city is defined by having "Netherlands" as its broader
    # country in DBpedia. The SERVICE statement asks
    # http://dbpdeia.org/sparql which cities have the country
    # "Netherlands". The answers to that sub-query will then be
    # used to finish off our original query about objects in the
    # Europeana database

    SERVICE <http://dbpedia.org/sparql> {
        ?dutch_city dbo:country dbr:Netherlands .
   }
}
# This query can potentially return a lot of objects, so let's
# just request the first 100 in order to speed up the search
LIMIT 100
```


User
Europeana
DBpedia
SELECT ?object...
start processing query...
... SERVICE <http://dbpedia.org> {...
What ?dutch_city has dbo:country dbr:Netherlands?
process query
Values of ?dutch_city: Amsterdam, Utrecht...
...finish processing query
SPARQL results
User
Europeana
DBpedia

www.websequencediagrams.com

Visualizing the query sequence of the above SPARQL request

An interlinked query like this means that we can ask Europeana questions about its objects that rely on information about geography (what cities are in the Netherlands?) that Europeana does not need to store and maintain itself. In the future, more cultural LOD will hopefully link to authority databases like the Getty's Union List of Artist Names, allowing, for example, the British Museum to outsource biographical data to the more complete resources at the Getty.

Working with SPARQL results

Having constructed and run a query... what do we do with the results? Many endpoints offer, like the British Museum, a web-based browser that returns human-readable results. However, SPARQL endpoints are designed to return structured data to be used by other programs.

Export results to CSV

In the top right corner of the results page for the BM endpoint, you will find links for both JSON and XML downloads. Other endpoints may also offer the option for a CSV/TSV download, however this option is not always available. The JSON and XML output from a SPARQL endpoint contain not only the values returned from the `SELECT` statement, but also additional metadata about variable types and languages.

Parsing the XML verson of this output may be done with a tool like Beautiful Soup (see Wieringa's *Programming Historian* lesson)[128] or Open Refine.[129] To quickly convert JSON results from a SPARQL endpoint into a tabular format, I recommend the free command line utility jq.[130] (For a tutorial on using command line programs, see "Introduction to the Bash Command Line".)[131] The following query will convert the special JSON RDF format into a CSV file, which you may load into your preferred program for further analysis and visualization:

```
jq -r '.head.vars as $fields | ($fields | @csv), (.results.bindings[] | [.[$
fields[]].value] | @csv)' sparql.json > sparql.csv
```

Export results to Palladio

The popular data exploration platform Palladio[132] can directly load data from a SPARQL endpoint. On the "Create a new project" screen, a link at the bottom to "Load data from a SPARQL endpoint (beta)" will provide you a field to enter the endpoint address, and a box for the query itself.

[128] Jeri Wieringa, 'Intro to Beautiful Soup', *The Programming Historian* (2012).

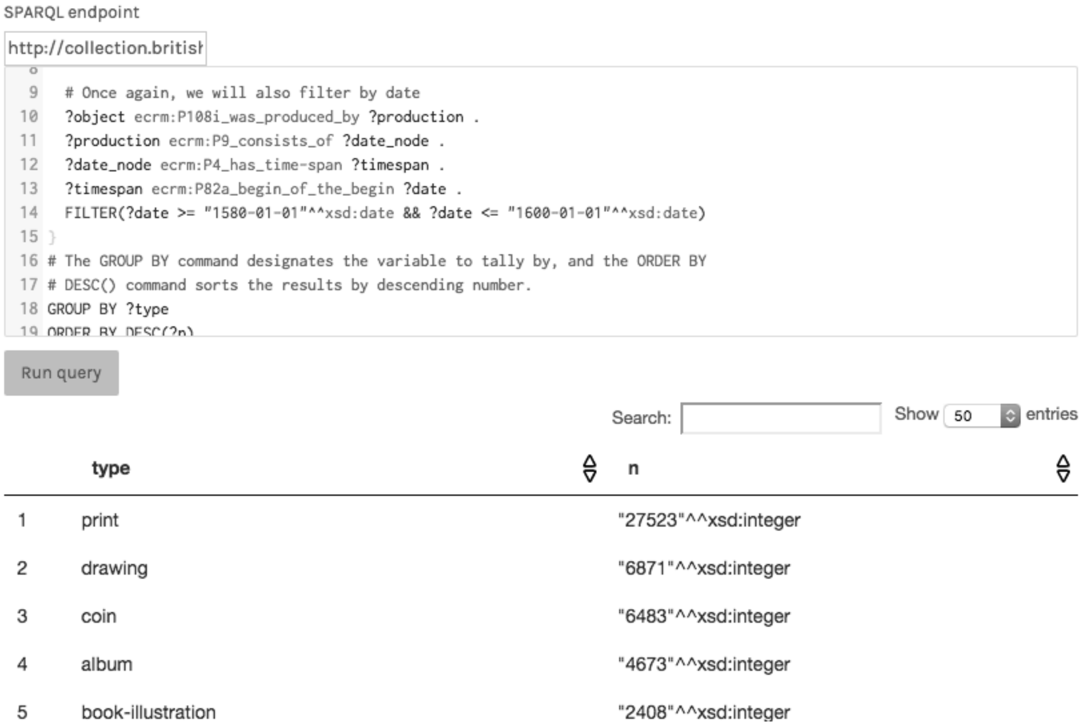
[129] 'Open Refine': http://openrefine.org/

[130] 'Download jq': https://stedolan.github.io/jq/download/

[131] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historian* (2014).

[132] 'Palladio': http://palladio.designhumanities.org/#/

Depending on the endpoint, you may need to specify the file output type in the endpoint address; for example, to load data from the BM endpoint you must use the address `http://collection.britishmuseum.org/sparql.json`. Try pasting in the aggregation query we used above to count artworks by type and clicking on "Run query". Palladio should display a preview table.



SPARQL endpoint

http://collection.britisl

```
  # Once again, we will also filter by date
  ?object ecrm:P108i_was_produced_by ?production .
  ?production ecrm:P9_consists_of ?date_node .
  ?date_node ecrm:P4_has_time-span ?timespan .
  ?timespan ecrm:P82a_begin_of_the_begin ?date .
  FILTER(?date >= "1580-01-01"^^xsd:date && ?date <= "1600-01-01"^^xsd:date)
}
# The GROUP BY command designates the variable to tally by, and the ORDER BY
# DESC() command sorts the results by descending number.
GROUP BY ?type
ORDER BY DESC(?n)
```

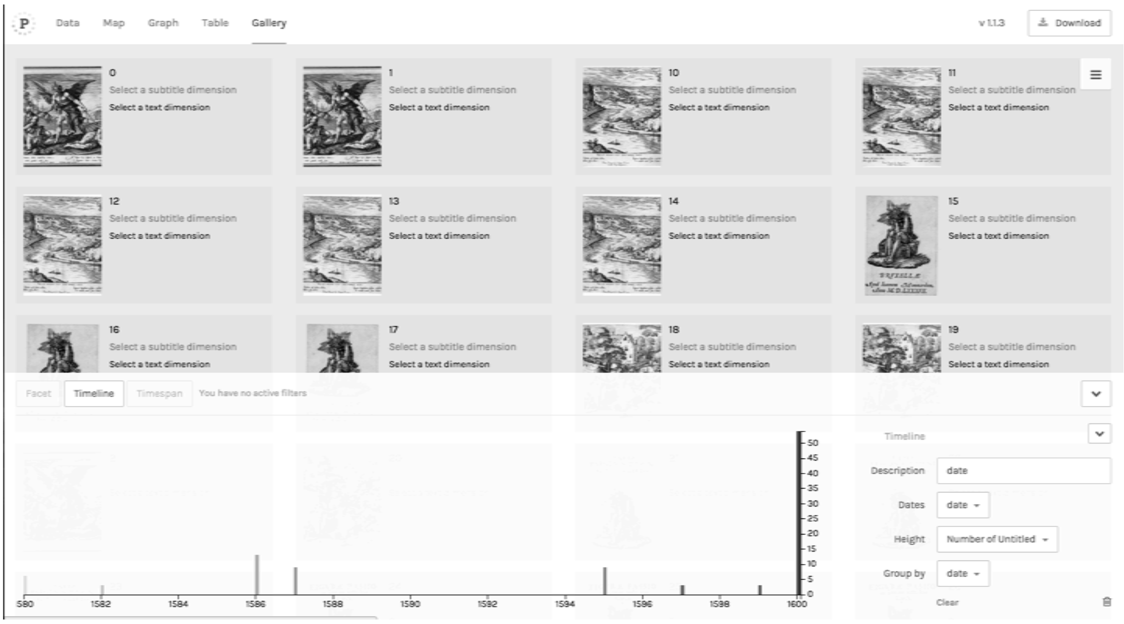
	type	n
1	print	"27523"^^xsd:integer
2	drawing	"6871"^^xsd:integer
3	coin	"6483"^^xsd:integer
4	album	"4673"^^xsd:integer
5	book-illustration	"2408"^^xsd:integer

Palladio's SPARQL query interface

After previewing the data returned by the endpoint, click on the "Load data" button at the bottom of the screen to begin manipulating it. (See Düring's lesson on networks for a more in-depth tutorial on Palladio.)[133] For example, we might make a query that returns links to the images of prints made between 1580 and 1600, and render that data as a grid of images sorted by date:[134]

[133] Martin Düring, 'From Hermeneutics to Data to Networks: Data Extraction and Network Visualization of Historical Sources', *The Programming Historian* (2015): http://programminghistorian.org/lessons/creating-network-diagrams-from-historical-sources.html#visualize-network-data-in-palladio

[134] Link available for this query on the online version of the tutorial. Trust us, you wouldn't want to type it out: http://programminghistorian.org/lessons/graph-databases-and-SPARQL



Description date

Dates date

Height Number of Untitled

Group by date

A gallery of images with a timeline of their creation dates generated using Palladio

Note that Palladio is designed to work with relatively small amounts of data (on the order of hundreds or thousands of rows, not tens of thousands), so you may have to use the `LIMIT` command that we used when querying the Europeana endpoint to reduce the number of results that you get back, just to keep the software from freezing.

Further reading

In this tutorial we got a look at the structure of LOD as well as a real-life example of how to write SPARQL queries for the British Museum's database. You also learned how to use aggregation commands in SPARQL to group, count, and sort results rather than simply list them.

There are even more ways to modify these queries, such as introducing `OR` and `UNION` statements (for describing conditional queries), and `CONSTRUCT` statements (for inferring new links based on defined rules), full-text searching, or doing other mathematical operations more complex than counting. For a more complete rundown of the commands available in SPARQL, see these resources:

How to SPARQL: http://rdf.myexperiment.org/howtosparql?
Wikibooks SPARQL tutorial:
https://en.wikibooks.org/wiki/XQuery/SPARQL_Tutorial

Both the Europeana and Getty Vocabularies LOD sites also offer extensive, and quite complex example queries which can be good sources for understanding how to search their data:

Europeana SPARQL how-to: http://labs.europeana.eu/api/linked-open-data-SPARQL-endpoint

Getty Vocabularies Example Queries: http://vocab.getty.edu/queries

About the Author

Matthew Lincoln is a PhD candidate at the University of Maryland, College Park. He is interested in the potential for computer-aided analysis of cultural datasets to help model long-term artistic trends in iconography, art markets, and social relations between artists in the early modern period.

Part Three: Transforming Data

Once you have data, you'll almost invariably have to transform it in some way. Sometimes this is 'cleaning' messy data. Other times it's just changing it into a format that's useable in your analysis. Knowing how to work with data systematically and at scale can save you weeks worth of work.

13. Working with Text Files in Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Setting Up an Integrated Development Environment for Python'.[135]

Lesson Goals

In this lesson you will learn how to manipulate text files using Python. This includes opening, closing, reading from, and writing to `.txt` files.

The next few lessons will involve downloading a web page from the Internet and reorganizing the contents into useful chunks of information. You will be doing most of your work using Python code written and executed in Komodo Edit.

Working with Text Files

Python makes it easy to work with files and text. Let's begin with files.

Creating and Writing to a Text File

Let's start with a brief discussion of terminology. In a previous lesson (depending on your operating system: Mac Installation, Windows Installation, or Linux Installation), you saw how to send information to the "Command Output" window of your text editor by using Python's `print` command.[136]

```
print 'hello world'
```

The Python programming language is *object-oriented*. That is to say that it is constructed around a special kind of entity, an *object*, which contains both *data* and a number of *methods* for accessing and altering that data. Once an object is created, it can interact with other objects.

In the example above, we see one kind of object, the *string* "hello world". The string is the sequence of characters enclosed by quotes. You can write a string one of three ways:

[135] William J. Turkel and Adam Crymble, 'Setting up an Integrated Development Environment for Python' (Mac, Linux, or Windows), *The Programming Historian* (2012).

[136] 'The Print statement', *Python*: https://docs.python.org/2/reference/simple_stmts.html#the-print-statement

```
message1 = 'hello world'
message2 = "hello world"
message3 = """hello
hello
hello world"""
```

The important thing to note is that in the first two examples you can use single or double quotes / inverted commas, but you cannot mix the two within one string.

For instance, the following are all wrong:

```
message1 = "hello world'
message2 = 'hello world"
message3 = 'I can't eat pickles'
```

Count the number of single quotes in message3. For that to work you would have to *escape* the apostrophe:

```
message3 = 'I can\'t eat pickles'
```

Or, rewrite the phrase as:

```
message3 = "I can't eat pickles"
```

In the third example, the triple quotes signify a string that covers more than one line.

`Print` is a command that prints objects in textual form. The print command, when combined with the string, produces a *statement*.

You will use `print` like this in cases where you want to create information that needs to be acted upon right away. Sometimes, however, you will be creating information that you want to save, to send to someone else, or to use as input for further processing by another program or set of programs. In these cases you will want to send information to files on your hard drive rather than to the "Command Output" pane. Enter the following program into your text editor and save it as `file-output.py`.

```
# file-output.py
f = open('helloworld.txt','w')
f.write('hello world')
f.close()
```

In Python, any line that begins with a hash mark (#) is known as a *comment* and is ignored by the Python interpreter. Comments are intended to allow programmers to communicate with one another (or to remind themselves of what their code does when they sit down with it a few months later). In a larger sense, programs themselves are typically written and formatted in a way that makes it easier for programmers to communicate with one another. Code that is closer to the requirements of the machine is referred to as *low-level*, whereas code that is closer to natural language is *high-level*. One of the benefits of using a language like

Python is that it is very high level, making it easier for us to communicate with you (at some cost in terms of computational efficiency).

In this program *f* is a *file object*, and `open`, `write` and `close` are *file methods*. In other words, open, write and close do something to the object *f* which is in this case defined as a `.txt` file. This is likely a different use of the term "method" than you might expect and from time to time you will find that words used in a programming context have slightly (or completely) different meanings than they do in everyday speech. In this case recall that methods are bits of code which perform actions. They do something to something else and return a result. You might try to think of it using a real-world example such giving commands to the family dog. The dog (the object) understands commands (i.e., has "methods") such as "bark", "sit", "play dead", and so on. We will discuss and learn how to use many other methods as we go along.

f is a variable name chosen by us; you could have named it just about anything you like. In Python, variable names can be made from upper- and lowercase letters, numbers and underscores…but you can't use the names of Python commands as variables. If you tried to name your file variable "print" for example, your program would not work because that is a `reserved word` that is part of the programming language.[137]

Python variable names are also *case-sensitive*, which means that foobar, Foobar and FOOBAR would all be different variables.

When you run this program, the `open` method will tell your computer to create a new text file `helloworld.txt` in the same folder as you have saved the `file-output.py` program. The *w parameter* says that you intend to write content to this new file using Python.

Note that since both the file name and the parameter are surrounded by single quotes you know they are both stored as strings; forgetting to include the quotation marks will cause your program to fail.

On the next line, your program writes the message "hello world" (another string) to the file and then closes it. (For more information about these statements, see the section on `File Objects` in the Python Library Reference.)[138]

Double-click on your "Run Python" button in Komodo Edit to execute the program (or the equivalent in whichever text-editor you have decided to use: e.g., click on the "#!" and "Run" in TextWrangler). Although nothing will be printed to the "Command Output" pane, you will see a status message that says something like

```
`/usr/bin/python file-output.py` returned 0.
```

[137] 'Keywords', *Python Reference Manual*: https://docs.python.org/release/2.5.4/ref/keywords.html

[138] 'File Objects', *Python*: https://docs.python.org/2/library/stdtypes.html#bltin-file-objects

in Mac or Linux, or

```
'C:\Python27\Python.exe file-output.py' returned 0.
```

in Windows.

This means that your program executed successfully. If you use *File -> Open -> File* in your Komodo Edit, you can open the file `helloworld.txt`. It should contain your one-line message:

```
Hello World!
```

Since text files include a minimal amount of formatting information, they tend to be small, easy to exchange between different platforms (i.e., from Windows to Linux or Mac or vice versa), and easy to send from one computer program to another. They can usually also be read by people using a text editor like Komodo Edit.

Reading From a Text File

Python also has methods which allow you to get information from files. Type the following program into your text editor and save it as `file-input.py`. When you click on "Run" to execute it, it will open the text file that you just created, read the one-line message from it, and print the message to the "Command Output" pane.

```
# file-input.py
f = open('helloworld.txt','r')
message = f.read()
print message
f.close()
```

In this case, the *r* parameter is used to indicate that you are opening a file to `read` from it. Parameters let you choose among the different options a particular method allows. Returning to the family dog example, the dog may be trained to bark once when he gets a beef-flavoured snack and twice when he gets a chicken-flavoured one. The flavour of the snack is a parameter. Each method is different in terms of what parameters it will accept. You cannot, for example, ask the dog to sing an Italian opera – unless your dog is particularly talented. You can look up the possible parameters for a particular method on the Python website, or often you can find them by typing the method into a search engine along with "Python".

`Read` is another file method. The contents of the file (the one-line message) are copied into *message*, which is what we've decided to call this string, and then the `print` command is used to send the contents of *message* to the "Command Output" pane.

Appending to a Pre-Existing Text File

A third option is to open a pre-existing file and add more to it. Note that if you `open` a file and use the `write` method, *the program will overwrite* *whatever might have been contained in the file*. This isn't an issue when you are creating a new file, or when you want to overwrite the contents of an existing file, but it might be undesirable when you are creating a log of events or compiling a large set of data into one file. So, instead of `write` you will want to use the `append` method, designated by `a`.

Type the following program into your text editor and save it as `file-append.py`. When you run this program it will open the same `helloworld.txt` file created earlier and append a second "hello world" to the file. The '\n' stands for new line.

```
# file-append.py
f = open('helloworld.txt','a')
f.write('\n' + 'hello world')
f.close()
```

After you have run the program, open the `helloworld.txt` file and see what happened. Close the text file and re-run `file-append.py` a few more times. When you open `helloworld.txt` again you should notice a few extra 'hello world' messages waiting for you.

In the next section, we will discuss modularity and reusing code.

Suggested Readings

Non-Programmer's Tutorial for Python 2.6/Hello, World[139]

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Code Reuse and Modularity in Python'.[140]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[139] 'Non-Programmer's Tutorial for Python 2.6/Hello, World' *Wikibooks*: https://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6/Hello,_World

[140] William J. Turkel and Adam Crymble, 'Code Reuse and Modularity in Python', *The Programming Historian*, 2012.

14. Code Reuse and Modularity in Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Working with Text Files in Python'.[141]

Lesson Goals

Computer programs can become long, unwieldy and confusing without special mechanisms for managing complexity. This lesson will show you how to reuse parts of your code by writing *Functions* and break your programs into *Modules*, in order to keep everything concise and easier to debug. Being able to remove a single dysfunctional module can save time and effort.

Functions

You will often find that you want to re-use a particular set of statements, usually because you have a task that you need to do over and over. Programs are mostly composed of routines that are powerful and general-purpose enough to be reused. These are known as functions, and Python has mechanisms that allow you to define new functions. Let's work through a very simple example of a function. Suppose you want to create a general purpose function for greeting people. Copy the following function definition into Komodo Edit and save it as `greet.py`.

```
# greet.py

def greetEntity (x):
    print "hello " + x

greetEntity("Everybody")
greetEntity("Programming Historian")
```

The line beginning with `def` is the function declaration. We are going to define ("def") a function, which in this case we have named "greetEntity". The `(x)` is the function's parameter. You should understand how that works in a moment. The second line contains the code of the function. This could be as many lines as we need, but in this case it is only a single line.

Note that *indentation* is very important in Python. The blank space before the `print` statement tells the interpreter that it is part of the function that

[141] William J. Turkel and Adam Crymble, 'Working with Text Files in Python', *The Programming Historian* (2012).

is being defined. You will learn more about this as we go along; for now, make sure to keep indentation the way we show it. Run the program, and you should see something like this:

```
hello Everybody
hello Programming Historian
```

This example contains one function: *greetEntity*. This function is then "called" (sometimes referred to as "invoked") two times. Calling or invoking a function just means we have told the program to execute the code in that function. Like giving the dog his chicken-flavoured treat (*woof* *woof*). In this case each time we have called the function we have given it a different parameter. Try editing `greet.py` so that it calls the *greetEntity* function a third time using your own name as a parameter. Run the program again. You should now be able to figure out what `(x)` does in the function declaration.

Before moving on to the next step, edit `greet.py` to delete the function calls, leaving only the function declaration. You're going to learn how to call the function from another program. When you are finished, your `greet.py` file should look like this:

```
# greet.py

def greetEntity (x):
    print "hello " + x
```

Modularity

When programs are small like the above example, they are typically stored in a single file. When you want to run one of your programs, you can simply send the file to the interpreter. As programs become larger, it makes sense to split them into separate files known as modules. This modularity makes it easier for you to work on sections of your larger programs. By perfecting each section of the program before putting all of the sections together, you not only make it easier to reuse individual modules in other programs, you make it easier to fix problems by being able to pinpoint the source of the error. When you break a program into modules, you are also able to hide the details for how something is done within the module that does it. Other modules don't need to know how something is accomplished if they are not responsible for doing it. This need-to-know principle is called “encapsulation“.

Suppose you were building a car. You could start adding pieces willy nilly, but it would make more sense to start by building and testing one module — perhaps the engine — before moving on to others. The engine, in turn, could be imagined to consist of a number of other, smaller modules like the carburettor and ignition system, and those are comprised of still smaller and more basic modules. The same is true when coding. You try to break a problem into smaller pieces, and solve those first.

You already created a module when you wrote the `greet.py` program. Now you are going to write a second program, `using-greet.py` which will `import` code from your module and make use of it. Python has a special `import` statement that allows one program to gain access to the contents of another program file. This is what you will be using.

Copy this code to Komodo Edit and save it as `using-greet.py`. This file is your program; `greet.py` is your module.

```
# using-greet.py

import greet
greet.greetEntity("everybody")
greet.greetEntity("programming historian")
```

We have done a few things here. First, we have told Python to `import` (load) the `greet.py` module, which we previously created.

You will also notice that whereas before we were able to run the function by calling only its name: *greetEntity("everybody")*, we now need to include the module's name followed by a dot (.) in front of the function name. In plain English this means: run the *greetEntity* function, which you should find in the `greet.py` module.

You can run your `using-greet.py` program with the "Run Python" command that you created in Komodo Edit. Note that you do not have to run your module…just the program that calls it. If all went well, you should see the following in the Komodo Edit output pane:

```
hello everybody
hello programming historian
```

Make sure that you understand the difference between loading a data file (e.g., `helloworld.txt`) and importing a program file (e.g. `greet.py`) before moving on.

Suggested Readings

Python Basics: http://www.astro.ufl.edu/~warner/prog/python.html

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Downloading Web Pages with Python'.[142]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[142] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python', *The Programming Historian*, 2012.

15. Manipulating Strings in Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Downloading Web Pages with Python'.[143]

Lesson Goals

This lesson is a brief introduction to string manipulation techniques in Python. Knowing how to manipulate strings plays a crucial role in most text processing tasks. If you'd like to experiment with the following lessons, you can write and execute short programs as we've been doing, or you can open up a Python shell / Terminal to try them out on the command line.

Manipulating Python Strings

If you have been exposed to another programming language before, you might have learned that you need to *declare* or *type* variables before you can store anything in them. This is not necessary when working with strings in Python. We can create a string simply by putting content wrapped with quotation marks into it with an equal sign (=):

```
message = "Hello World"
```

String Operators: Adding and Multiplying

As we mentioned previously, a string is a type of object, one that consists of a series of characters. Python already knows how to deal with a number of general-purpose and powerful representations, including strings. One way to manipulate strings is by using *string operators*. These operators are represented by symbols that you likely associate with mathematics, such as +, -, *, /, and =. When used with strings, they perform actions that are similar to, but not the same as, their mathematical counterparts.

Concatenate

This term means to join strings together. The process is known as *concatenating* strings and it is done using the plus (+) operator. Note that you must be explicit about where you want blank spaces to occur by placing them between single quotation marks also.

[143] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python', *The Programming Historian* (2012).

In this example, the string "message1" is given the content "hello world".

```
message1 = 'hello' + ' ' + 'world'
print message1
-> hello world
```

Multiply

If you want multiple copies of a string, use the multiplication (*) operator. In this example, string *message2a* is given the content "hello" times three; string *message 2b* is given content "world"; then we print both strings.

```
message2a = 'hello ' * 3
message2b = 'world'
print message2a + message2b
-> hello hello hello world
```

Append

What if you want to add material to the end of a string successively? There is a special operator for that (+=).

```
message3 = 'howdy'
message3 += ' '
message3 += 'world'
print message3
-> howdy world
```

String Methods: Finding, Changing

In addition to operators, Python comes pre-installed with dozens of string methods that allow you to do things to strings. Used alone or in combination, these methods can do just about anything you can imagine to strings. The good news is that you can reference a list of String Methods on the Python website,[144] including information on how to use each properly. To make sure that you've got a basic grasp of string methods, what follows is a brief overview of some of the more commonly used ones:

Length

You can determine the number of characters in a string using `len`. Note that the blank space counts as a separate character.

```
message4 = 'hello' + ' ' + 'world'
print len(message4)
-> 11
```

Find

You can search a string for a *substring* and your program will return the starting index position of that substring. This is helpful for further processing. Note that indexes are numbered from left to right and that the count starts with position 0, not 1.

[144] 'String Methods', *Python*: https://docs.python.org/2/library/stdtypes.html#string-methods

```
message5 = "hello world"
message5a = message5.find("worl")
print message5a
-> 6
```

If the substring is not present, the program will return a value of -1.

```
message6 = "Hello World"
message6b = message6.find("squirrel")
print message6b
-> -1
```

Lower Case

Sometimes it is useful to convert a string to lower case. For example, if we standardize case it makes it easier for the computer to recognize that "Sometimes" and "sometimes" are the same word.

```
message7 = "HELLO WORLD"
message7a = message7.lower()
print message7a
-> hello world
```

The opposite effect, raising characters to upper case, can be achieved by changing `.lower()` to `.upper()`.

Replace

If you need to replace a substring throughout a string you can do so with the `replace` method.

```
message8 = "HELLO WORLD"
message8a = message8.replace("L", "pizza")
print message8a
-> HEpizzapizzaO WORpizzaD
```

Slice

If you want to `slice` off unwanted parts of a string from the beginning or end you can do so by creating a substring. The same kind of technique also allows you to break a long string into more manageable components.

```
message9 = "Hello World"
message9a = message9[1:8]
print message9a
-> ello Wo
```

You can substitute variables for the integers used in this example.

```
startLoc = 2
endLoc = 8
message9b = message9[startLoc: endLoc]
print message9b
-> llo Wo
```

This makes it much easier to use this method in conjunction with the `find` method as in the next example, which checks for the letter "d" in the first six characters of "Hello World" and correctly tells us it is not there (-1). This technique is much more useful in longer strings – entire documents for example. Note that the absence of an integer before the colon signifies we want to start at the beginning of the string. We could use the same technique to tell the program to go all the way to the end by putting no integer after the colon. And remember, index positions start counting from 0 rather than 1.

```
message9 = "Hello World"
print message9[:5].find("d")
-> -1
```

There are lots more, but the string methods above are a good start. Note that in this last example, we are using square brackets instead of parentheses. This difference in *syntax* signals an important distinction. In Python, parentheses are usually used to *pass an argument* to a function. So when we see something like

```
print len(message7)
```

it means pass the string *message7* to the function `len` then send the returned value of that function to the `print` statement to be printed. If a function can be called without an argument, you often have to include a pair of empty parentheses after the function name anyway. We saw an example of that, too:

```
message7 = "HELLO WORLD"
message7a = message7.lower()
print message7a
-> hello world
```

This statement tells Python to apply the `lower` function to the string *message7* and store the returned value in the string *message7a*.

The square brackets serve a different purpose. If you think of a string as a sequence of characters, and you want to be able to access the contents of the string by their location within the sequence, then you need some way of giving Python a location within a sequence. That is what the square brackets do: indicate a beginning and ending location within a sequence as we saw when using the `slice` method.

Escape Sequences

What do you do when you need to include quotation marks within a string? You don’t want the Python interpreter to get the wrong idea and end the string when it comes across one of these characters. In Python, you can put a backslash (\) in front of a quotation mark so that it doesn't terminate the string. These are known as escape sequences.

```
print '\"'
-> "

print 'The program printed \"hello world\"'
-> The program printed "hello world"
```

Two other escape sequences allow you to print tabs and newlines:

```
print 'hello\thello\thello\nworld'
->hello hello hello
world
```

Suggested Reading

Lutz, Learning Python
Ch. 7: Strings
Ch. 8: Lists and Dictionaries
Ch. 10: Introducing Python Statements
Ch. 15: Function Basics

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your programming-historian directory. At the end of each chapter you can download the programming-historian zip file to make sure you have the correct code. Note we have removed unneeded files from earlier lessons. Your directory may contain more files and that's ok!

programming-historian-2 (zip):
http://programminghistorian.org/assets/programming-historian2.zip

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'From HTML to List of Words (part 1)'.[145]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[145] William J. Turkel and Adam Crymble, 'From HTML to List of Words (part 1)', *The Programming Historian*, 2012.

16. From HTML to List of Words (part 1)

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Manipulating Strings in Python'.[146]

Lesson Goals

In this two-part lesson, we will build on what you've learned about 'Downloading Web Pages with Python',[147] learning how to remove the *HTML markup* from the webpage of Benjamin Bowsey's 1780 criminal trial transcript.[148] We will achieve this by using a variety of *string operators*, *string methods* and close reading skills. We introduce *looping* and *branching* so that programs can repeat tasks and test for certain conditions, making it possible to separate the content from the HTML tags. Finally, we convert content from a long string to a *list of words* that can later be sorted, indexed, and counted.

The Challenge

To get a clearer picture of the task ahead, open the *obo-t17800628-33.html* file that you created in 'Downloading Web Pages with Python'[149] (or download and save the trial if you do not already have a copy)[150], then look at the HTML source by clicking on *Tools -> Web Developer -> Page Source.* As you scroll through the source code you'll notice that there are a few HTML tags mixed in with the text. Because this is a printable version there is far less HTML than you will find on the other versions of the transcript (see the HTML[151] and XML[152] versions to compare). While not

[146] William J. Turkel and Adam Crymble, 'Manipulating Strings in Python', *The Programming Historian* (2012).

[147] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python' *The Programming Historian* (2012).

[148] 'Trial of Benjamin Bowsey, June 1780, (t17800628-33)', *The Old Bailey Online*: http://www.oldbaileyonline.org/print.jsp?div=t17800628-33

[149] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python' *The Programming Historian* (2012).

[150] 'Trial of Benjamin Bowsey, June 1780, (t17800628-33)', *The Old Bailey Online*: http://www.oldbaileyonline.org/print.jsp?div=t17800628-33

[151] *Old Bailey Proceedings Online* (www.oldbaileyonline.org, version 7.2, 26 February 2016), June 1780, trial of BENJAMIN BOWSEY (t17800628-33): www.oldbaileyonline.org/browse.jsp?id=t17800628-33-defend448&div=t17800628-33

[152]http://www.oldbaileyonline.org/browse.jsp?foo=bar&path=sessionsPapers/17800628.xml&div=t17800628-33&xml=yes

mandatory, we recommend that at this point you take the W3 Schools HTML tutorial[153] to familiarize yourself with HTML markup. If your work often requires that you remove HTML markup, it will certainly help to be able to understand it when you see it.

Files Needed For This Lesson

obo-t17800628-33.html

If you do not have these files, you can download programming-historian-2, the (zip - http://programminghistorian.org/assets/programming-historian2.zip) file from the previous lesson.

Devising an Algorithm

Since the goal is to get rid of the HTML, the first step is to create an *algorithm* that returns only the text (minus the HTML tags) of the article. An algorithm is a procedure that has been specified in enough detail that it can be implemented on a computer. It helps to write your algorithms first in plain English; it's a great way to outline exactly what you want to do before diving into code. To construct this algorithm you are going to use your close reading skills to figure out a way to capture only the textual content of the biography.

Looking at the source code of *obo-t17800628-33.html* you will notice the actual transcript does not start right away. Instead there are a couple of HTML tags and some citation information. In this case:

```
<div style="font-family:serif;"><i>Old Bailey Proceedings Online</i>
(www.oldbaileyonline.org, version 6.0, 01 July 2011), June 1780, trial of BE
NJAMIN BOWSEY (t17800628-33).<hr/><h2>BENJAMIN BOWSEY...
```

We are only interested in the transcript itself, not the extra metadata contained in the tags. However, you will notice that the end of the metadata corresponds with the start of the transcript. This makes the location of the metadata a potentially useful marker for isolating the transcript text.

At a glance, we can see that the metadata ends with two HTML tags: `<hr/><h2>`. We might be able to use those to find the starting point of our transcript text. We are lucky in this case because it turns out that these tags are a reliable way to find the start of transcript text in the printable versions (if you want, take a look at a few other printable trials to check). We are also lucky because other than a few HTML tags at the end of the transcript, there is no further information on the page. Had there been other unrelated content, we would take a similar approach and look for some way of isolating the end of the desired text. Well-formatted websites

[153] 'HTML(5) Tutorial': http://www.w3schools.com/html/

will almost always have some unique way of signalling the end of the content. You often just need to look closely.

The next thing that you want to do is strip out all of the HTML markup that remains mixed in with the content. Since you know HTML tags are always found between matching pairs of angle brackets, it's probably a safe bet that if you remove everything between angle brackets, you will remove the HTML and be left only with the transcript. Note that we are making the assumption that the transcript will not contain the mathematical symbols for "less than" or "greater than." If Bowsey was a mathematician, this assumption would not be as safe.

The following describes our algorithm in words.

To isolate the content:

Download the transcript text
Search the HTML for and store the location of `<hr/><h2>`
Save everything after the `<hr/><h2>` tags to a string: *pageContents*

At this point we have the trial transcript text, plus HTML markup. Next:

Look at every character in the *pageContents* string, one character at a time
If the character is a left angle bracket (<) we are now inside a tag so ignore each following character
If the character is a right angle bracket (>) we are now leaving the tag; ignore the current character, but look at each following character
If we're not inside a tag, append the current character to a new variable: *text*

Finally:

Split the text string into a list of individual words that can later be manipulated further.

Isolating Desired Content

The following step uses Python commands introduced in the 'Manipulating Strings in Python'[154] lesson to implement the first half of the algorithm: removing all content before the `<hr/><h2>` tags. To recap, the algorithm was as follows:

Download the transcript text
Search the HTML for and store the location of `<hr/><h2>`
Save everything after the `<hr/><h2>` tags to a string: *pageContents*

[154] William J. Turkel and Adam Crymble 'Manipulating Strings in Python' *The Programming Historian* (2012).

To achieve this, you will use the find string method and create a new substring containing only the desired content using the index as start point for the substring.

As you work, you will be developing separate files to contain your code. One of these will be called *obo.py* (for “Old Bailey Online”). This file is going to contain all of the code that you will want to re-use; in other words, *obo.py* is a module. We discussed the idea of modules in ‘Code Reuse and Modularity in Python’[155] when we saved our functions to *greet.py*.

Create a new file named *obo.py* and save it to your *programming-historian* directory. We are going to use this file to keep copies of the functions needed to process The Old Bailey Online. Type or copy the following code into your file.

```
# obo.py

def stripTags(pageContents):
    startLoc = pageContents.find("<hr/><h2>")

    pageContents = pageContents[startLoc:]
    return pageContents
```

Create a second file, *trial-content.py*, and save the program shown below.

```
# trial-content.py

import urllib2, obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
HTML = response.read()

print obo.stripTags(HTML)
```

When you run *trial-content.py* it will get the web page for Bowsey’s trial transcript, then look in the *obo.py* module for the *stripTags* function. It will use that function to extract the stuff after the `<hr/><h2>` tags. With any luck, this should be the textual content of the Bowsey transcript, along with some of HTML markup. Don’t worry if your Command Output screen ends in a thick black line. Komodo Edit’s output screen has a maximum number of characters it will display, after which characters start literally writing over one another on the screen, giving the appearance of a black blob. Don’t worry, the text is in there even though you cannot read it; you can cut and paste it to a text file to double check.

Let’s take a moment to make sure we understand how *trial-contents.py* is able to use the functions stored in *obo.py*. The *stripTags* function that we saved to *obo.py* requires one argument. In other words, to run properly it needs one piece of information to be supplied. Recall the trained dog

[155] William J. Turkel and Adam Crymble, ‘Code Reuse and Modularity in Python’, *The Programming Historian* (2012).

example from a previous lesson. In order to bark, the dog needs two things: air and a delicious treat. The *stripTags* function in *obo.py* needs one thing: a string called *pageContents*. But you'll notice that when we call *stripTags* in the final program (*trialcontents.py*) there's no mention of "*pageContents*". Instead the function is given HTML as an argument. This can be confusing to many people when they first start programming. Once a function has been declared, we no longer need to use the same variable name when we call the function. As long as we provide the right type of argument, everything should work fine, no matter what we call it. In this case we wanted *pageContents* to use the contents of our HTML variable. You could have passed it any string, including one you input directly between the parentheses. Try rerunning *trial-content.py*, changing the *stripTags* argument to "I am quite fond of dogs" and see what happens. Note that depending on how you define your function (and what it does) your argument may need to be something other than a string: an *integer* for example.

Suggested Reading

Lutz, Learning Python[156]
Ch. 7: Strings
Ch. 8: Lists and Dictionaries
Ch. 10: Introducing Python Statements
Ch. 15: Function Basics

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your programming-historian directory. At the end of each chapter you can download the programming-historian zip file to make sure you have the correct code. Note we have removed unneeded files from earlier lessons. Your directory may contain more files and that's ok!

programming-historian-2 (zip - http://programminghistorian.org/assets/programming-historian2.zip)

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'From HTML to List of Words (part 2)'.[157]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[156] Mark Lutz, *Learning Python* (5th edition), O'Reilley, 2013.

[157] William J. Turkel and Adam Crymble, 'From HTML to List of Words (part 2)', *The Programming Historian*, 2012.

17. From HTML to List of Words (part 2)

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'From HTML to List of Words (part 1)'.[158]

Lesson Goals

In this lesson, you will learn the Python commands needed to implement the second part of the algorithm begun in the 'From HTML to a List of Words (part 1)'. The first half of the algorithm gets the content of an HTML page and saves only the content that follows the `<hr/><h2>` tags. The second half of the algorithm does the following:

Look at every character in the *pageContents* string, one character at a time
If the character is a left angle bracket (<) we are now inside a tag so ignore each following character
If the character is a right angle bracket (>) we are now leaving the tag; ignore the current character, but look at each following character
If we're not inside a tag, append the current character to a new variable: *text*
Split the *text* string into a list of individual words that can later be manipulated further.

Files Needed For This Lesson

obo.py
trial-content.py

If you do not have these files, you can download programming-historian-2, a (zip - http://programminghistorian.org/assets/programming-historian2.zip) file from the previous lesson.

Repeating and Testing in Python

The next stage in implementing the algorithm is to look at every character in the *pageContents* string, one at a time and decide whether the character belongs to HTML markup or to the content of the trial transcript. Before you can do this you'll have to learn a few techniques for repeating tasks and for testing conditions.

[158] William J. Turkel and Adam Crymble, 'From HTML to List of Words (part 1)', *The Programming Historian* (2012).

Looping

Like many programming languages, Python includes a number of *looping* mechanisms. The one that you want to use in this case is called a *for loop*. The version below tells the interpreter to do something for each character in a string named *pageContents*. The variable *char* will contain each character from *pageContents* in succession. We gave *char* its name; it does not have any special significance and could have been named *jingles* or *k* if we had felt so inclined. You can use the colour-coding in Komodo Edit as a guideline for deciding if a word is a variable with a user-given name (such as “*char*“) or a Python-defined name that serves a specific purpose (such as “for“). It is usually a good idea to give variables names that provide information about what they contain. This will make it much easier to understand a program that you haven’t looked at for a while. With this in mind, “*jingles*” is probably not a very good choice for a variable name in this case.

```
for char in pageContents:
    # do something with char
```

Branching

Next you need a way of testing the contents of a string, and choosing a course of action based on that test. Again, like many programming languages, Python includes a number of *branching* mechanisms. The one that you want to use here is called an *if statement*. The version below tests to see whether the string named *char* consists of a left angle bracket. As we mentioned earlier, indentation is important in Python. If code is indented, Python will execute it when the condition is true.

Note that Python uses a single equals sign (=) for *assignment*, that is for setting one thing equal to something else. In order to test for *equality*, use double equals signs (==) instead. Beginning programmers often confuse the two.

```
if char == '<':
    # do something
```

A more general form of the if statement allows you to specify what to do in the event that your test condition is false.

```
if char == '<':
    # do something
else:
    # do something different
```

In Python you have the option of doing further tests after the first one, by using an *elif statement* (which is shorthand for else if).

```
if char == '<':
    # do something
elif char == '>':
    # do another thing
else:
    # do something completely different
```

Use the Algorithm to Remove HTML Markup

You now know enough to implement the second part of the algorithm: removing all HTML tags. In this part of the algorithm we want to:

Look at every character in the *pageContents* string, one character at a time

If the character is a left angle bracket (<) we are now inside a tag so ignore the character

If the character is a right angle bracket (>) we are now leaving the tag; ignore the character

If we're not inside a tag, append the current character to a new variable: text

To do this, you will use a for loop to look at each successive character in the string. You will then use an if / elif statement to determine whether the character is part of HTML markup or part of the content, then append the content characters to the *text* string. How will we keep track of whether or not we're inside a tag? We can use an integer variable, which will be 1 (true) if the current character is inside a tag and 0 (false) if it's not (in the example below we have named the variable *inside*).

The stripTags Routine

Putting it all together, the final version of the routine is shown below. Note that we are expanding the *stripTags* function created above. Make sure you maintain the indentation as shown when you replace the old *stripTags* routine in *obo.py* with this new one.

Your routine may look slightly different and as long as it works that's fine. If you've elected to experiment, it's probably best to try our version as well to make sure that your program does what ours does.

```
# obo.py
def stripTags(pageContents):
    startLoc = pageContents.find("<hr/><h2>")
    pageContents = pageContents[startLoc:]

    inside = 0
    text = ''

    for char in pageContents:
        if char == '<':
            inside = 1
        elif (inside == 1 and char == '>'):
            inside = 0
        elif inside == 1:
            continue
        else:
            text += char

    return text
```

There are two new Python concepts in this new code: *continue* and *return*.

The Python continue statement tells the interpreter to jump back to the top of the enclosing loop. So if we are processing characters inside of a pair of angle brackets, we want to go get the next character in the *pageContents* string without adding anything to our *text* variable.

In our previous examples we have used `print` extensively. This outputs the result of our program to the screen for the user to read. Often, however, we wish to allow one part of the program to send information to another part. When a function finishes executing, it can return a value to the code which called it. If we were to call *stripTags* using another program, we would do so like this:

```
#understanding the Return statement

import obo

myText = "This is my <h1>HTML</h1> message"

theResult = obo.stripTags(myText)
```

By using `return`, we have been able to save the output of the *stripTags* function directly into a variable, which we can then resume processing as needed using additional code.

Note that in the *stripTags* example from the start of this sub-section, the value that we want to return now is not *pageContents*, but rather the content which has had the HTML markup stripped out.

To test our new *stripTags* routine, you can run *trial-content.py* again. Since we’ve redefined *stripTags*, the *trial-content.py* program now does something different (and closer to what we want). Before you continue, make sure that you understand why the behaviour of *trial-content.py* would change when we only edited *obo.py*.

Python Lists

Now that you have the ability to extract raw text from web pages, you're going to want to get the text in a form that is easy to process. So far, when you've needed to store information in your Python programs, you've usually used strings. There were a couple of exceptions, however. In the *stripTags* routine, you also made use of an `integer`[159] named *inside* to store a 1 when you were processing a tag and a 0 when you weren't. You can do mathematical operations on integers but you cannot store fractions or decimal numbers in integer variables.

```
inside = 1
```

And whenever you've needed to read from or write to a file, you've used a special file handle like *f* in the example below.

```
f = open('helloworld.txt','w')
f.write('hello world')
f.close()
```

One of the most useful `types`[160] of object that Python provides, however, is the *list*, an ordered collection of other objects (including, potentially, other lists). Converting a string into a list of characters or words is straightforward. Type or copy the following program into your text editor to see two ways of achieving this. Save the file as *string-to-list.py* and execute it. Compare the two lists that are printed to the Command Output pane and see if you can figure out how the code works.

```
# string-to-list.py

# some strings
s1 = 'hello world'
s2 = 'howdy world'

# list of characters
charlist = []
for char in s1:
    charlist.append(char)
print charlist

# list of 'words'
wordlist = s2.split()
print wordlist
```

The first routine uses a for loop to step through each character in the string *s1*, and appends the character to the end of *charlist*. The second routine makes use of the split operation to break the string *s2* apart wherever there is whitespace (spaces, tabs, carriage returns and similar characters). Actually, it is a bit of a simplification to refer to the objects in the second

[159] 'Numeric Types – int, float, long, complex' *Python Library Reference*: https://docs.python.org/2.4/lib/typesnumeric.html

[160] 'Types' *Python*: https://docs.python.org/3/library/types.html

list as words. Try changing *s2* in the above program to 'howdy world!' and running it again. What happened to the exclamation mark? Note, that you will have to save your changes before using Run Python again.

Given what you've learned so far, you can now open a URL, download the web page to a string, strip out the HTML and then split the text into a list of words. Try executing the following program.

```
#html-to-list1.py
import urllib2, obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
html = response.read()
text = obo.stripTags(html)
wordlist = text.split()

print wordlist[0:120]
```

You should get something like the following.

```
['BENJAMIN', 'BOWSEY,', 'Breaking', 'Peace', '>',
'riot,', '28th', 'June', '1780.', '324.', 'BENJAMIN',
'BOWSEY', '(a', 'blackmoor', ')', 'was', 'indicted',
'for', 'that', 'he', 'together', 'with', 'five',
'hundred', 'other', 'persons', 'and', 'more,', 'did,',
'unlawfully,', 'riotously,', 'and', 'tumultuously',
'assemble', 'on', 'the', '6th', 'of', 'June', 'to',
'the', 'disturbance', 'of', 'the', 'public', 'peace',
'and', 'did', 'begin', 'to', 'demolish', 'and', 'pull',
'down', 'the', 'dwelling', 'house', 'of', 'Richard',
'Akerman', ',', 'against', 'the', 'form', 'of', 'the',
'statute,', '&c.', 'ROSE', 'JENNINGS', ',', 'Esq.',
'sworn.', 'Had', 'you', 'any', 'occasion', 'to', 'be',
'in', 'this', 'part', 'of', 'the', 'town,', 'on', 'the',
'6th', 'of', 'June', 'in', 'the', 'evening?', '-', 'I',
'dined', 'with', 'my', 'brother', 'who', 'lives',
'opposite', 'Mr.', "Akerman's", 'house.', 'They',
'attacked', 'Mr.', "Akerman's", 'house', 'precisely',
'at', 'seven', "o'clock;", 'they', 'were', 'preceded',
'by', 'a', 'man']
```

Simply having a list of words doesn't buy you much yet. As human beings, we already have the ability to read. You're getting much closer to a representation that your programs can process, however.

Suggested Reading

Lutz, Learning Python[161]
Ch. 7: Strings
Ch. 8: Lists and Dictionaries
Ch. 10: Introducing Python Statements

[161] Mark Lutz, *Learning Python* (5th edition), O'Reilley, 2013.

Ch. 15: Function Basics

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your programming-historian directory. At the end of each chapter you can download the programming-historian zip file to make sure you have the correct code. Note we have removed unneeded files from earlier lessons. Your directory may contain more files and that's ok!

programming-historian-2 (zip - http://programminghistorian.org/assets/programming-historian2.zip)

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Normalizing Textual Data with Python'.[162]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[162] William J. Turkel and Adam Crymble, 'Normalizing Textual Data with Python', *The Programming Historian*, 2012.

18. Normalizing Textual Data with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'From HTML to List of Words (part 2)'.[163]

Lesson Goals

The list that we created in the 'From HTML to a List of Words (2)' needs some *normalizing* before it can be used further. We are going to do this by applying additional string methods, as well as by using *regular expressions*. Once normalized, we will be able to more easily analyze our data.

Files Needed For This Lesson

html-to-list-1.py
obo.py

If you do not have these files from the previous lesson, you can download a zip file from the previous lesson here:

http://programminghistorian.org/lessons/from-html-to-list-of-words-2#code-syncing.

Cleaning up the List

In 'From HTML to a List of Words (2)', we wrote a Python program called *html-to-list-1.py* which downloaded a web page,[164] stripped out the HTML formatting and metadata and returned a list of "words" like the one shown below. Technically, these entities are called "*tokens*" rather than "words". They include some things that are, strictly speaking, not words at all (like the abbreviation &c. for "etcetera"). They also include some things that may be considered composites of more than one word. The possessive "Akerman's," for example, is sometimes analyzed by linguists as two words: "Akerman" plus a possessive marker. Is "o'clock" one word or two? And so on.

[163] William J. Turkel and Adam Crymble, 'From HTML to List of Words (part 2)', *The Programming Historian* (2012).

[164] The page was a criminal trial transcript from 1780: 'Trial of Benjamin Bowsey, June 1780 (t17800628-33)', *the Old Bailey Online*: http://www.oldbaileyonline.org/print.jsp?div=t17800628-33

Turn back to your program *html-to-list-1.py* and make sure that your results look something like this:

```
['BENJAMIN', 'BOWSEY,', 'Breaking', 'Peace', '>',
'riot,', '28th', 'June', '1780.', '324.', 'BENJAMIN',
'BOWSEY', '(a', 'blackmoor', ')', 'was', 'indicted',
'for', 'that', 'he', 'together', 'with', 'five',
'hundred', 'other', 'persons', 'and', 'more,', 'did,',
'unlawfully,', 'riotously,', 'and', 'tumultuously',
'assemble', 'on', 'the', '6th', 'of', 'June', 'to',
'the', 'disturbance', 'of', 'the', 'public', 'peace',
'and', 'did', 'begin', 'to', 'demolish', 'and', 'pull',
'down', 'the', 'dwelling', 'house', 'of', 'Richard',
'Akerman', ',', 'against', 'the', 'form', 'of', 'the',
'statute,', '&c.', 'ROSE', 'JENNINGS', ',', 'Esq.',
'sworn.', 'Had', 'you', 'any', 'occasion', 'to', 'be',
'in', 'this', 'part', 'of', 'the', 'town,', 'on', 'the',
'6th', 'of', 'June', 'in', 'the', 'evening?', '-', 'I',
'dined', 'with', 'my', 'brother', 'who', 'lives',
'opposite', 'Mr.', "Akerman's", 'house.', 'They',
'attacked', 'Mr.', "Akerman's", 'house', 'precisely',
'at', 'seven', "o'clock;", 'they', 'were', 'preceded',
'by', 'a', 'man']
```

By itself, this ability to separate the document into words doesn't buy us much because we already know how to read. We can use the text, however, to do things that aren't usually possible without special software. We're going to start by computing the frequencies of tokens and other linguistic units, a classic measure of a text.

It is clear that our list is going to need some cleaning up before we can use it to count frequencies. In keeping with the practices established in 'From HTML to a List of Words (1)',[165] let's try to describe our algorithm in plain English first. We want to know the frequency of each meaningful word that appears in the trial transcript. So, the steps involved might look like this:

Convert all words to lower case so that "BENJAMIN" and "benjamin" are counted as the same word

Remove any strange or unusual characters

Count the number of times each word appears

Remove overly common words such as "it", "the", "and", etc.

Convert to Lower Case

Typically tokens are *folded* to lower case when counting frequencies, so we'll do that using the string method lower which was introduced in 'Manipulating Strings in Python'.[166] Since this is a string method we will have to apply it to the string: *text* in the *html-to-list1.py* program. Amend

[165] William J. Turkel and Adam Crymble, 'From HTML to a List of Words (part 1)', *The Programming Historian* (2012).

[166] William J. Turkel and Adam Crymble, 'Manipulating Strings in Python', *The Programming Historian* (2012).

html-to-list1.py by adding the string tag `lower()` to the the end of the *text* string.

```
#html-to-list1.py
import urllib2, obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
html = response.read()
text = obo.stripTags(html).lower() #add the string method here.
wordlist = text.split()

print (wordlist)
```

You should now see the same list of words as before, but with all characters changed to lower case.

By calling methods one after another like this, we can keep our code short and make some pretty significant changes to our program.

Like we said before, Python makes it easy to do a lot with very little code!

At this point, we might look through a number of other *Old Bailey Online* entries and a wide range of other potential sources to make sure that there aren't other special characters that are going to cause problems later. We might also try to anticipate situations where we don't want to get rid of punctuation (e.g., distinguishing monetary amounts like "$1629" or "£1295" from dates, or recognizing that "1629-40" has a different meaning than "1629 40".) This is what professional programmers get paid to do: try to think of everything that might go wrong and deal with it in advance.

We're going to take a different approach. Our main goal is to develop techniques that a working historian can use during the research process. This means that we will almost always prefer approximately correct solutions that can be developed quickly. So rather than taking the time now to make our program robust in the face of exceptions, we're simply going to get rid of anything that isn't an accented or unaccented letter or an Arabic numeral. Programming is typically a process of "stepwise refinement". You start with a problem and part of a solution, and then you keep refining your solution until you have something that works better.

Python Regular Expressions

We've eliminated upper case letters. That just leaves all the punctuation to get rid of. Punctuation will throw off our frequency counts if we leave them in. We want "evening?" to be counted as "evening" and "1780." as "1780", of course.

It is possible to use the replace string method to remove each type of punctuation:

```
text = text.replace('[', '')
text = text.replace(']', '')
text = text.replace(',', '')
#etc...
```

But that’s not very efficient. In keeping with our goal of creating short, powerful programs, we’re going to use a mechanism called *regular expressions*. Regular expressions are provided by many programming languages in a range of different forms.

Regular expressions allow you to search for well defined patterns and can drastically shorten the length of your code. For instance, if you wanted to know if a substring matched a letter of the alphabet, rather than use an if/else statement to check if it matched the letter “a” then “b” then “c”, and so on, you could use a regular expression to see if the substring matched a letter between “a” and “z”. Or, you could check for the presence of a digit, or a capital letter, or any alphanumeric character, or a carriage return, or any combination of the above, and more.

In Python, regular expressions are available as a Python module. To speed up processing it is not loaded automatically because not all programs require it. So, you will have to `import` the module (called *re*) in the same way that you imported your *obo.py* module.

Since we’re interested in only alphanumeric characters, we’ll create a regular expression that will isolate only these and remove the rest. Copy the following function and paste it into the *obo.py* module at the end. You can leave the other functions in the module alone, as we’ll continue to use those.

```
# Given a text string, remove all non-alphanumeric
# characters (using Unicode definition of alphanumeric).

def stripNonAlphaNum(text):
    import re
    return re.compile(r'\W+', re.UNICODE).split(text)
```

The regular expression in the above code is the material inside the string, in other words `W+`. The `W` is shorthand for the class of *non-alphanumeric characters*. In a Python regular expression, the plus sign (+) matches one or more copies of a given character. The `re.UNICODE` tells the interpreter that we want to include characters from the world’s other languages in our definition of “alphanumeric”, as well as the A to Z, a to z and 0-9 of English. Regular expressions have to be *compiled* before they can be used, which is what the rest of the statement does. Don’t worry about understanding the compilation part right now.

When we refine our *html-to-list1.py* program, it now looks like this:

```
#html-to-list1.py
import urllib2, obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
html = response.read()
text = obo.stripTags(html).lower()
wordlist = obo.stripNonAlphaNum(text)

print wordlist[0:500]
```

When you execute the program and look through its output in the “Command Output” pane, you’ll see that it has done a pretty good job. This code will split hyphenated forms like “coach-wheels” into two words and turn the possessive “s” or “o’clock” into separate words by losing the apostrophe. But it is a good enough approximation to what we want that we should move on to counting frequencies before attempting to make it better. (If you work with sources in more than one language, you need to learn more about the Unicode standard[167] and about Python support for it.)[168]

For extra practice with Regular Expressions, you may find Chapter 7 of Mark Pilgrim’s “Dive into Python” a useful tutorial.[169]

If you are following along the ‘Intro to Python’ lessons in order, the next lesson in this sequence is ‘Counting Word Frequencies with Python’.[170]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[167] ‘The Unicode Consortium’: http://unicode.org/

[168] ‘Unicode’, *Dive into Python*: http://www.diveintopython.net/xml_processing/unicode.html

[169] ‘Chapter 7. Regular Expressions’, *Dive Into Python*: http://www.diveintopython.net/regular_expressions/index.html

[170] William J. Turkel and Adam Crymble, ‘Counting Word Frequencies with Python’, *The Programming Historian*, 2012.

19. Keywords in Context (Using n-grams) with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Output Data as an HTML File with Python'.[171]

Lesson Goals

Like in 'Output Data as HTML File', this lesson takes the frequency pairs collected in 'Counting Word Frequencies with Python'[172] and outputs them in HTML. This time the focus is on keywords in context (KWIC) which creates n-grams from the original document content – in this case a trial transcript from the *Old Bailey Online*. You can use your program to select a keyword and the computer will output all instances of that keyword, along with the words to the left and right of it, making it easy to see at a glance how the keyword is used.

Once the KWICs have been created, they are then wrapped in HTML and sent to the browser where they can be viewed. This reinforces what was learned in 'Output Data as HTML File with Python',[173] opting for a slightly different output.

At the end of this lesson, you will be able to extract all possible n-grams from the text. In the next lesson, you will be learn how to output all of the n-grams of a given keyword in a document downloaded from the Internet, and display them clearly in your browser window.

Files Needed For This Lesson

`obo.py`

If you do not have these files from the previous lesson, you can download programming-historian-3, a zip file from the previous lesson:

http://programminghistorian.org/assets/programming-historian3.zip

[171] William J. Turkel and Adam Crymble, 'Output Data as an HTML File with Python', *The Programming Historian* (2012).

[172] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

[173] William J. Turkel and Adam Crymble, 'Output Data as an HTML File with Python', *The Programming Historian* (2012).

From Text to N-Grams to KWIC

Now that you know how to harvest the textual content of a web page automatically with Python, and have begun to use strings, lists and dictionaries for text processing, there are many other things that you can do with the text besides counting frequencies. People who study the statistical properties of language have found that studying linear sequences of linguistic units can tell us a lot about a text. These linear sequences are known as *bigrams* (2 units), *trigrams* (3 units), or more generally as *n-grams*.

You have probably seen n-grams many times before. They are commonly used on search results pages to give you a preview of where your keyword appears in a document and what the surrounding context of the keyword is. This application of n-grams is known as keywords in context (often abbreviated as KWIC). For example, if the string in question were "it was the best of times it was the worst of times it was the age of wisdom it was the age of foolishness" then a 7-gram for the keyword "wisdom" would be:

```
the age of wisdom it was the
```

An n-gram could contain any type of linguistic unit you like. For historians you are most likely to use characters as in the bigram "qu" or words as in the trigram "the dog barked"; however, you could also use phonemes, syllables, or any number of other units depending on your research question.

What we're going to do next is develop the ability to display KWIC for any keyword in a body of text, showing it in the context of a fixed number of words on either side. As before, we will wrap the output so that it can be viewed in Firefox and added easily to Zotero.

From Text to N-grams

Since we want to work with words as opposed to characters or phonemes, it will be much easier to create n-grams using a list of words rather than strings. As you already know, Python can easily turn a string into a list using the `split` operation. Once split it becomes simple to retrieve a subsequence of adjacent words in the list by using a *slice*, represented as two indexes separated by a colon. This was introduced when working with strings in ‘Manipulating Strings in Python’.[174]

```
message9 = "Hello World"
message9a = message9[1:8]
print message9a
-> ello Wo
```

[174] William J. Turkel and Adam Crymble, ‘Manipulating Strings in Python’, *The Programming Historian* (2012).

However, we can also use this technique to take a predetermined number of neighbouring words from the list with very little effort. Study the following examples, which you can try out in a Python Shell.

```
wordstring = 'it was the best of times it was the worst of times '
wordstring += 'it was the age of wisdom it was the age of foolishness'
wordlist = wordstring.split()

print wordlist[0:4]
-> ['it', 'was', 'the', 'best']

print wordlist[0:6]
-> ['it', 'was', 'the', 'best', 'of', 'times']

print wordlist[6:10]
-> ['it', 'was', 'the', 'worst']

print wordlist[0:12]
-> ['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst',
'of', 'times']

print wordlist[:12]
-> ['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst',
'of', 'times']

print wordlist[12:]
-> ['it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'o
f', 'foolishness']
```

In these examples we have used the `slice` method to return parts of our list. Note that there are two sides to the colon in a slice. If the right of the colon is left blank as in the last example above, the program knows to automatically continue to the end – in this case, to the end of the list. The second last example above shows that we can start at the beginning by leaving the space before the colon empty. This is a handy shortcut available to keep your code shorter.

You can also use variables to represent the index positions. Used in conjunction with a `for` loop, you could easily create every possible n-gram of your list. The following example returns all 5-grams of our string from the example above.

```
i = 0
for items in wordlist:
    print wordlist[i: i+5]
    i += 1
```

Keeping with our modular approach, we will create a function and save it to the `obo.py` module that can create n-grams for us. Study and type or copy the following code:

```
# Given a list of words and a number n, return a list
# of n-grams.

def getNGrams(wordlist, n):
    return [wordlist[i:i+n] for i in range(len(wordlist)-(n-1))]
```

This function may look a little confusing as there is a lot going on here in not very much code. It uses a *list comprehension* to keep the code compact. The following example does exactly the same thing:

```
def getNGrams(wordlist, n):
    ngrams = []
    for i in range(len(wordlist)-(n-1)):
        ngrams.append(wordlist[i:i+n])
    return ngrams
```

A concept that may still be confusing to you are the two function arguments. Notice that our function has two variable names in the parentheses after its name when we declared it: *wordlist*, *n*. These two variables are the function arguments. When you call (run) this function, these variables will be used by the function for its solution. Without these arguments there is not enough information to do the calculations. In this case, the two pieces of information are the list of words you want to turn into n-grams (wordlist), and the number of words you want in each n-gram (n). For the function to work it needs both, so you call it in like this (save the following as `useGetNGrams.py` and run):

```
#useGetNGrams.py

import obo

wordstring = 'it was the best of times it was the worst of times '
wordstring += 'it was the age of wisdom it was the age of foolishness'
allMyWords = wordstring.split()

print obo.getNGrams(allMyWords, 5)
```

Notice that the arguments you enter do not have to have the same names as the arguments named in the function declaration. Python knows to use *allMyWords* everywhere in the function that *wordlist* appears, since this is given as the first argument. Likewise, all instances of *n* will be replaced by the integer 5 in this case. Try changing the 5 to a string, such as "elephants" and see what happens when you run your program. Note that because *n* is being used as an integer, you have to ensure the argument sent is also an integer. The same is true for strings, floats or any other variable type sent as an argument.

You can also use a Python shell to play around with the code to get a better understanding of how it works. Paste the function declaration for *getNGrams* (either of the two functions above) into your Python shell.

```
test1 = 'here are four words'
test2 = 'this test sentence has eight words in it'

getNGrams(test1.split(), 5)
-> []

getNGrams(test2.split(), 5)
-> [['this', 'test', 'sentence', 'has', 'eight'],
['test', 'sentence', 'has', 'eight', 'words'],
['sentence', 'has', 'eight', 'words', 'in'],
['has', 'eight', 'words', 'in', 'it']]
```

There are two concepts that we see in this example of which you need to be aware. Firstly, because our function expects a list of words rather than a string, we have to convert the strings into lists before our function can handle them. We could have done this by adding another line of code above the function call, but instead we used the `split` method directly in the function argument as a bit of a shortcut.

Secondly, why did the first example return an empty list rather than the n-grams we were after? In *test1*, we have tried to ask for an n-gram that is longer than the number of words in our list. This has resulted in a blank list. In *test2* we have no such problem and get all possible 5-grams for the longer list of words. If you wanted to you could adapt your function to print a warning message or to return the entire string instead of an empty list.

We now have a way to extract all possible n-grams from a body of text. In the next lesson, we can focus our attention on isolating those n-grams that are of interest to us.

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each chapter you can download the "programming-historian" zip file to make sure you have the correct code. If you are following along with the Mac / Linux version you may have to open the `obo.py` file and change "file:///Users/username/Desktop/programming-historian/" to the path to the directory on your own computer.

programming-historian [Mac / Linux]

(zip: http://programminghistorian.org/assets/programming-historian-mac-linux.zip)

programming-historian [Windows]

(zip: http://programminghistorian.org/assets/programming-historian-windows.zip)

If you are following along the ‘Intro to Python’ lessons in order, the next lesson in this sequence is ‘Output Keywords in Context into an HTML File with Python’.[175]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[175] William J. Turkel and Adam Crymble, ‘Output Keywords in Context into an HTML File with Python’, *The Programming Historian*, 2012.

20. Creating New Items in Zotero

Amanda Morton – 2013

Editor's Note: This is the second of three lessons on the 'Zotero API'. You may find it easier to complete this tutorial if you have already completed the previous one: 'Intro to the Zotero API'.[176]

Using Python to Create an New Zotero Item

In 'Intro to the Zotero API', you learned a little bit about Zotero;[177] now you can access some of its functions using Python scripts. In this lesson, you will create a new item in a Zotero library and add some basic metadata such as title and date.

Creating a new Zotero Item

It will be helpful to remember that Zotero began as a citation management system, and that an *item* on Zotero contains only metadata; it's a bit like a library calling card. To upload file contents into Zotero, you would create an *attachment* to that item. But for now you will start by creating a new Zotero Item and assigning some information to metadata fields.

Your first step is to import the python modules that you will need for this program.

```
from libZotero import zotero
import urllib2
import datetime
```

Your next line of code will connect to the Zotero group library for this lesson using the unique group id and API key. (You can also replace the first number in the line with your own group or user ID, but if you are trying to connect to an individual user library, you must change the word `group` to the word `user` and create your own API key.)

```
#Links to zotero group library
zlib = zotero.Library('group', '155975','<null>', 'f4Bfk3OTYb7bukNwfcKXKNLG'
)
```

Now that you have imported the required modules and connected to your Zotero library, you can create a new item and assign it some metadata. Start by using the following code to create a new item of the type *document* and set the title to *Python Lesson Document.*

[176] Amanda Morton, 'Intro to the Zotero API', *the Programming Historian* (2013).

[177] *Zotero*: https://www.zotero.org/

```
#create a new item of type document
newItem = zotero.getTemplateItem('document')

#sets the title of the item to Python Lesson Document
newItem.set('title', 'Python Lesson Document')
```

Next you will add two more types of metadata to your item. First, you will add an abstract note, which is basically a short description of the item you have created. Then you will set the item's creation date to the current date.

```
#adds a new abstract note
newItem.set('abstractNote', 'Created using a zotero python library and the w
rite api')

#sets date to current date
now = datetime.datetime.today().strftime("%Y-%m-%d")
newItem.set('date', now)
```

Now that you have set the important metadata for your item, you can make a request to the API to create that item. This code has set the *writeFailure* property to display an error message if the item is not successfully created.

```
# make the request to the API to create the item
# a Zotero Item object will be returned
# if the creation went okay it will have a writeFailure property set to Fals
e
createdItem = zlib.createItem(newItem)
if createdItem.writeFailure != False:
   print(createdItem.writeFailure['code'])
   print(createdItem.writeFailure['message'])
```

Your last step is to add a *tag* to your new item. The following code will tag your item as *python lesson* and update the item with the new tag. Just as in the last segment, this code contains a *writeFailure* property that will print an error message if the item has not updated correctly.

```
#adds a new tag to the new item
tagname = 'python lesson'

#in the bracket (tagname, '<tag type:0>')
createdItem.addTag(tagname, '0')

#updates the item with the new tag
updatedItem = zlib.writeUpdatedItem(createdItem)
if updatedItem.writeFailure != False:
   print("Error updating item")
   print(updatedItem.writeFailure['code'])
   print(updatedItem.writeFailure['message'])
```

At last, you have created a new item with a title and a tag name. This last line of code will confirm the item you have just created.

```
print 'Created new item <%s> with new tag <%s>' % (createdItem.title, tagnam
e)
```

If all has gone according to plan, your output should look like this:

```
Created new item <Python Lesson Document> with new tag <python lesson>
```

You can also check your Zotero library to find the document that you made using Python. The title, abstract, and date should be filled out, and the tag should appear also.

By editing the program above, you can create items with different types (such as books, journal articles, or newspapers) and specify more precise titles, creation dates, and tags. To see a list of all the Item Types available in the Zotero API, use your browser to navigate to this URL:

```
https://api.zotero.org/itemTypes
```

You can then see the fields available in each Item Type template by navigating to the following URL, replacing `document` with the key for the Item Type that interests you:

```
https://api.zotero.org/items/new?itemType=document
```

For example, the list of Item Types returned by the first URL shows a type called `videoRecording`. In our code above, you could request a template for that type by changing the `document` argument in our `getItemTemplate()` function with `videoRecording`. To see which fields are available in this template, you could navigate in your browser to the appropriate URL:

```
https://api.zotero.org/items/new?itemType=videoRecording
```

For more details, see the documentation on write requests for the Zotero API.[178]

Editorial Note: This is the second of three lessons on the 'Zotero API'. The final lesson in this sequence is 'Counting Frequencies from Zotero Items'.[179]

About the Author

Amanda Morton is a DH Fellow at the Center for History and New Media.

[178] 'Zotero Web API Write Requests', *Zotero*: https://www.zotero.org/support/dev/web_api/v3/write_requests

[179] Spencer Roberts, 'Counting Frequencies from Zotero Items', *The Programming Historian* (2013).

21. Intro to Beautiful Soup

Jeri Wieringa – 2012

What is Beautiful Soup?

This tutorial uses Python 2.7.2 and BeautifulSoup 4.[180]

It assumes basic knowledge of HTML, CSS, and the Document Object Model.[181] It also assumes some knowledge of Python. For a more basic introduction to Python, see Working with Text Files.[182]

Most of the work is done in the terminal. For an introduction to using the terminal, see the Scholar's Lab Command Line Bootcamp tutorial.[183]

"You didn't write that awful page. You're just trying to get some data out of it. Beautiful Soup is here to help." (Opening lines of Beautiful Soup).[184]

Beautiful Soup is a Python library for getting data out of HTML, XML, and other markup languages. Say you've found some webpages that display data relevant to your research, such as date or address information, but that do not provide any way of downloading the data directly. Beautiful Soup helps you pull particular content from a webpage, remove the HTML markup, and save the information. It is a tool for web scraping that helps you clean up and parse the documents you have pulled down from the web.

The Beautiful Soup documentation will give you a sense of variety of things that the Beautiful Soup library will help with, from isolating titles and links, to extracting all of the text from the html tags, to altering the HTML within the document you're working with.

Installing Beautiful Soup

Installing Beautiful Soup is easiest if you have pip or another Python installer already in place. If you don't have pip, run through a quick tutorial by Fred Gibbs (2013) on installing python modules to get it

[180] 'Python 2.7 Release', https://www.python.org/download/releases/2.7/ ; 'Beautiful Soup 4.4.0 Documentation', http://www.crummy.com/software/BeautifulSoup/bs4/doc/

[181] 'HTML', *Wikipedia:* https://en.wikipedia.org/wiki/HTML; 'Cascading Style Sheets', *Wikipedia*: https://en.wikipedia.org/wiki/Cascading_Style_Sheets ; 'Document Object Model', *Wikipedia*: https://en.wikipedia.org/wiki/Document_Object_Model;

[182] William J. Turkel and Adam Crymble, 'Working with Text Files in Python', *Programming Historian* 2012.

[183] 'The Command Line', *The Praxis Program at the Scholar's Lab*: http://praxis.scholarslab.org/scratchpad/bash/

[184] 'Beautiful Soup 4.4.0 Documentation', http://www.crummy.com/software/BeautifulSoup/bs4/doc/

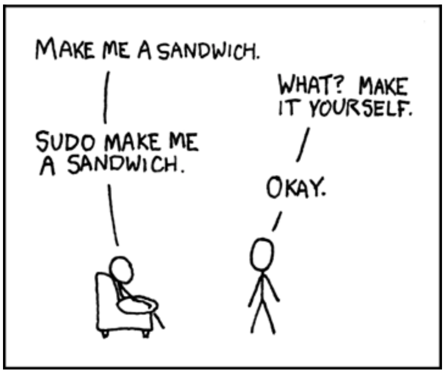
running.[185] Once you have pip installed, run the following command in the terminal to install Beautiful Soup:

```
pip install beautifulsoup4
```

You may need to preface this line with “sudo”, which gives your computer permission to write to your root directories and requires you to re-enter your password. This is the same logic behind you being prompted to enter your password when you install a new program.

With sudo, the command is:

```
sudo pip install beautifulsoup4
```



The Power of Sudo: 'Sandwich' by XKCD.

Application: Extracting names and URLs from an HTML page

Preview: Where we are going

Because I like to see where the finish line is before starting, I will begin with a view of what we are trying to create. We are attempting to go from a search results page where the html page looks like this:

```
<table border="1" cellspacing="2" cellpadding="3">
<tbody>
<tr>
<th>Member Name</th>
<th>Birth-Death</th>
</tr>
<tr>
<td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000035">ADAMS
, George Madison</a></td>
<td>1837-1920</td>
</tr>
<tr>
<td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000074">ALBER
T, William Julian</a></td>
<td>1816-1879</td>
</tr>
```

[185] Fred Gibbs, ‘Installing Python Modules with pip’, *The Programming Historian* (2013).

```
<tr>
<td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000077">ALBRI
GHT, Charles</a></td>
<td>1830-1880</td>
</tr>
</tbody>
</table>
```

to a CSV file with names and urls that looks like this:

```
"ADAMS, George Madison",http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000035
"ALBERT, William Julian",http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000074
"ALBRIGHT, Charles",http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000077
```

using a Python script like this:

```
from bs4 import BeautifulSoup
import csv

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

f = csv.writer(open("43rd_Congress.csv", "w"))
f.writerow(["Name", "Link"])    # Write column headers as the first line

links = soup.find_all('a')
for link in links:
    names = link.contents[0]
    fullLink = link.get('href')

    f.writerow([names,fullLink])
```

This tutorial explains to how to assemble the final code.

Get a webpage to scrape

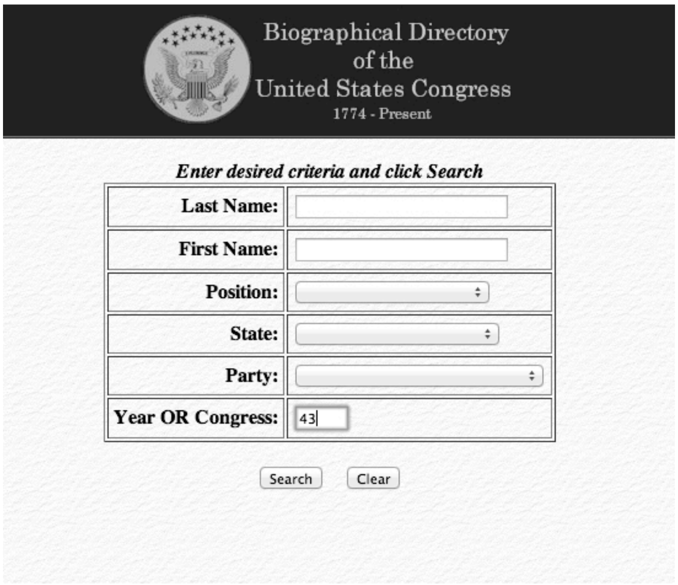
The first step is getting a copy of the HTML page(s) want to scrape. You can combine BeautifulSoup with urllib3 to work directly with pages on the web.[186] This tutorial, however, focuses on using BeautifulSoup with local (downloaded) copies of html files.

The Congressional database that we're using is not an easy one to scrape because the URL for the search results remains the same regardless of what you're searching for. While this can be bypassed programmatically, it is easier for our purposes to go to:

http://bioguide.congress.gov/biosearch/biosearch.asp

And search for Congress number 43, and to save a copy of the results page.

[186] 'urllib3 Documentation' *urllib3*: http://urllib3.readthedocs.org/en/latest/

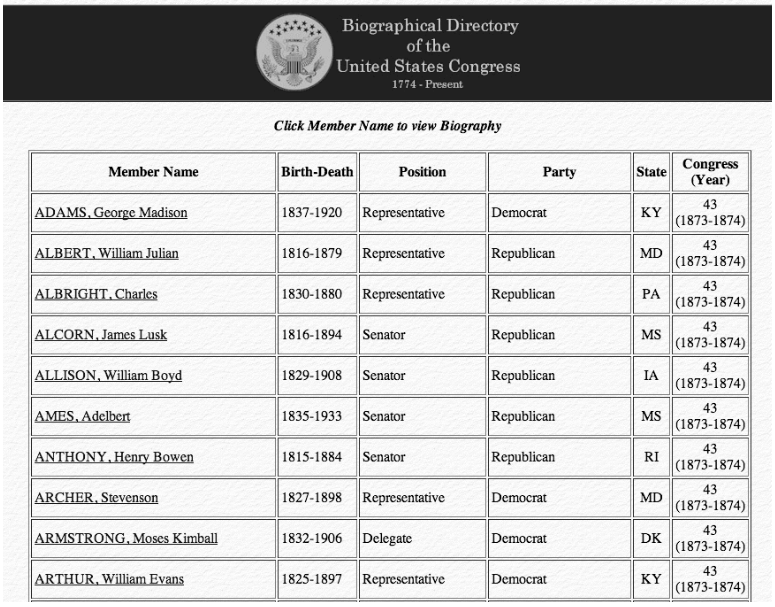


Biographical Directory of the United States Congress
1774 - Present

Enter desired criteria and click Search

Last Name:	
First Name:	
Position:	
State:	
Party:	
Year OR Congress:	43

BioGuide Interface Search for 43rd Congress



Biographical Directory of the United States Congress
1774 - Present

Click Member Name to view Biography

Member Name	Birth-Death	Position	Party	State	Congress (Year)
ADAMS, George Madison	1837-1920	Representative	Democrat	KY	43 (1873-1874)
ALBERT, William Julian	1816-1879	Representative	Republican	MD	43 (1873-1874)
ALBRIGHT, Charles	1830-1880	Representative	Republican	PA	43 (1873-1874)
ALCORN, James Lusk	1816-1894	Senator	Republican	MS	43 (1873-1874)
ALLISON, William Boyd	1829-1908	Senator	Republican	IA	43 (1873-1874)
AMES, Adelbert	1835-1933	Senator	Republican	MS	43 (1873-1874)
ANTHONY, Henry Bowen	1815-1884	Senator	Republican	RI	43 (1873-1874)
ARCHER, Stevenson	1827-1898	Representative	Democrat	MD	43 (1873-1874)
ARMSTRONG, Moses Kimball	1832-1906	Delegate	Democrat	DK	43 (1873-1874)
ARTHUR, William Evans	1825-1897	Representative	Democrat	KY	43 (1873-1874)

BioGuide Results We want to download the HTML behind this page.

Selecting "File" and "Save Page As …" from your browser window will accomplish this (life will be easier if you avoid using spaces in your filename). I have used "43rd-congress.html". Move the file into the folder you want to work in.

(To learn how to automate the downloading of HTML pages using Python, see Ian Milligan's tutorial on 'Automated Downloading with Wget' or Adam Crymble's 'Downloading Multiple Records Using Query Strings'.)[187]

[187] Ian Milligan, 'Automated Downloading with Wget', *The Programming Historian* (2012); Adam Crymble, 'Downloading Multiple Records Using Query Strings', *The Programming Historian* (2012).

Identify content

One of the first things Beautiful Soup can help us with is locating content that is buried within the HTML structure. Beautiful Soup allows you to select content based upon tags (example: `soup.body.p.b` finds the first bold item inside a paragraph tag inside the body tag in the document). To get a good view of how the tags are nested in the document, we can use the method "prettify" on our soup object.

Create a new text file called "`soupexample.py`" in the same location as your downloaded HTML file. This file will contain the Python script that we will be developing over the course of the tutorial.

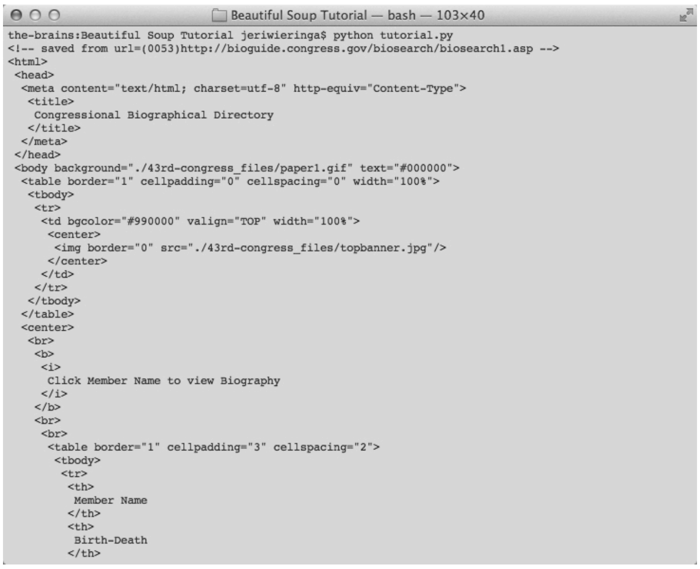
To begin, import the Beautiful Soup library, open the HTML file and pass it to Beautiful Soup, and then print the "pretty" version in the terminal.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(open("43rd-congress.html"))

print(soup.prettify())
```

Save "`soupexample.py`" in the folder with your HTML file and go to the command line. Navigate (use '`cd`') to the folder you're working in and execute the following:

```
python soupexample.py
```

You should see your terminal window fill up with a nicely indented version of the original html text (see Figure below). This is a visual representation of how the various tags relate to one another.



```
the-brains:Beautiful Soup Tutorial jeriwieringa$ python tutorial.py
<!-- saved from url=(0053)http://bioguide.congress.gov/biosearch/biosearch1.asp -->
<html>
 <head>
  <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
   <title>
    Congressional Biographical Directory
   </title>
  </meta>
 </head>
 <body background="./43rd-congress_files/paper1.gif" text="#000000">
  <table border="1" cellpadding="0" cellspacing="0" width="100%">
   <tbody>
    <tr>
     <td bgcolor="#990000" valign="TOP" width="100%">
      <center>
       <img border="0" src="./43rd-congress_files/topbanner.jpg"/>
      </center>
     </td>
    </tr>
   </tbody>
  </table>
  <center>
   <br>
    <b>
     <i>
      Click Member Name to view Biography
     </i>
    </b>
    <br>
     <br>
      <table border="1" cellpadding="3" cellspacing="2">
       <tbody>
        <tr>
         <th>
          Member Name
         </th>
         <th>
          Birth-Death
         </th>
```

'Pretty' print of the BioGuide results

Using BeautifulSoup to select particular content

Remember that we are interested in only the names and URLs of the various member of the 43rd Congress. Looking at the “pretty” version of the file, the first thing to notice is that the data we want is not too deeply embedded in the HTML structure.

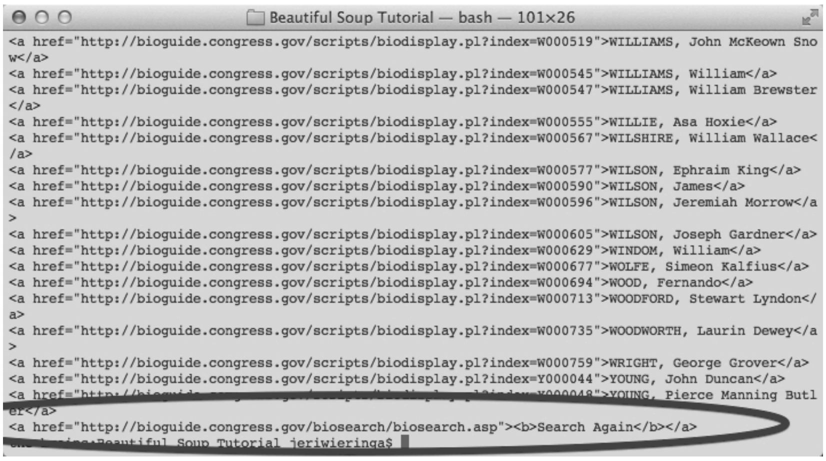
Both the names and the URLs are, most fortunately, embedded in “<a>” tags. So, we need to isolate out all of the “<a>” tags. We can do this by updating the code in “`soupexample.py`” to the following:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

links = soup.find_all('a')

for link in links:
    print link
```

Save and run the script again to see all of the anchor tags in the document.

```
python soupexample.py
```

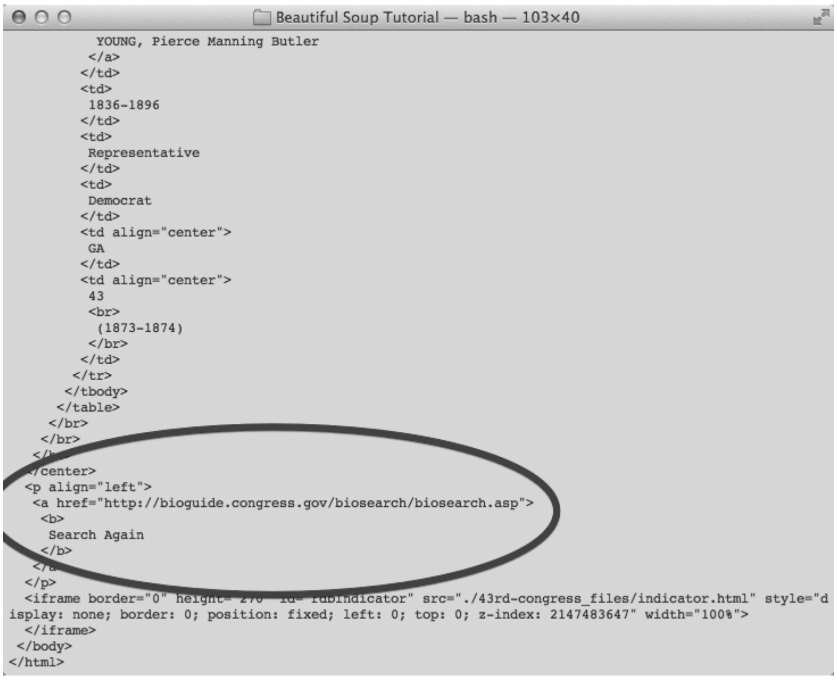
One thing to notice is that there is an additional link in our file – the link for an additional search.



```
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000519">WILLIAMS, John McKeown Sno
w</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000545">WILLIAMS, William</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000547">WILLIAMS, William Brewster
</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000555">WILLIE, Asa Hoxie</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000567">WILSHIRE, William Wallace<
/a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000577">WILSON, Ephraim King</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000590">WILSON, James</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000596">WILSON, Jeremiah Morrow</a
>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000605">WILSON, Joseph Gardner</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000629">WINDOM, William</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000677">WOLFE, Simeon Kalfius</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000694">WOOD, Fernando</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000713">WOODFORD, Stewart Lyndon</
a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000735">WOODWORTH, Laurin Dewey</a
>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000759">WRIGHT, George Grover</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000044">YOUNG, John Duncan</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000048">YOUNG, Pierce Manning Butl
er</a>
<a href="http://bioguide.congress.gov/biosearch/biosearch.asp"><b>Search Again</b></a>
:Beautiful Soup Tutorial jeriwieringa$
```

The URLs and names, plus one addition

We can get rid of this with just a few lines of code. Going back to the pretty version, notice that this last “<a>” tag is not within the table but is within a “<p>” tag.



```
            YOUNG, Pierce Manning Butler
           </a>
          </td>
          <td>
           1836-1896
          </td>
          <td>
           Representative
          </td>
          <td>
           Democrat
          </td>
          <td align="center">
           GA
          </td>
          <td align="center">
           43
           <br>
            (1873-1874)
           </br>
          </td>
         </tr>
        </tbody>
       </table>
      </br>
     </br>
    </br>
   </center>
  <p align="left">
   <a href="http://bioguide.congress.gov/biosearch/biosearch.asp">
    <b>
     Search Again
    </b>
   </a>
  </p>
  <iframe border="0" height="270" id="fdbIndicator" src="./43rd-congress_files/indicator.html" style="d
isplay: none; border: 0; position: fixed; left: 0; top: 0; z-index: 2147483647" width="100%">
  </iframe>
 </body>
</html>
```

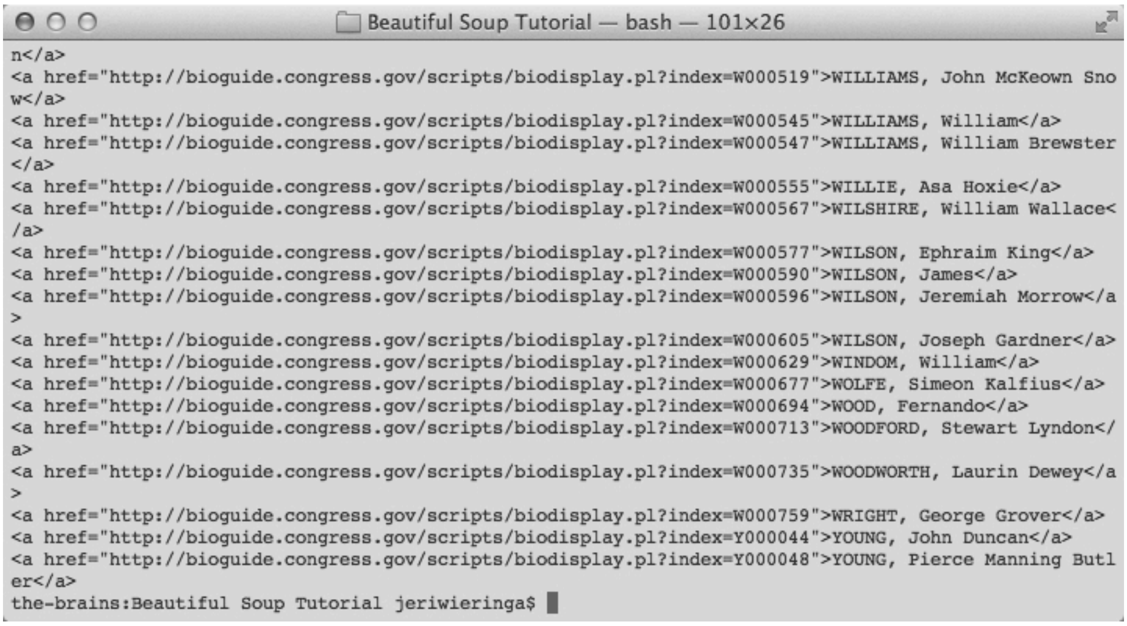
The Rogue Link

Because Beautiful Soup allows us to modify the HTML, we can remove the "<a>" that is under the "<p>" before searching for all the "<a>" tags.

To do this, we can use the "decompose" method, which removes the specified content from the "soup". Do be careful when using "decompose"—you are deleting both the HTML tag and all of the data inside of that tag. If you have not correctly isolated the data, you may be deleting information that you wanted to extract. Update the file as below and run again.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

links = soup.find_all('a')

for link in links:
    print link
```

Success! We have isolated out all of the links we want and none of the links we don't!



```
n</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000519">WILLIAMS, John McKeown Sno
w</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000545">WILLIAMS, William</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000547">WILLIAMS, William Brewster
</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000555">WILLIE, Asa Hoxie</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000567">WILSHIRE, William Wallace<
/a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000577">WILSON, Ephraim King</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000590">WILSON, James</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000596">WILSON, Jeremiah Morrow</a
>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000605">WILSON, Joseph Gardner</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000629">WINDOM, William</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000677">WOLFE, Simeon Kalfius</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000694">WOOD, Fernando</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000713">WOODFORD, Stewart Lyndon</
a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000735">WOODWORTH, Laurin Dewey</a
>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000759">WRIGHT, George Grover</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000044">YOUNG, John Duncan</a>
<a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000048">YOUNG, Pierce Manning Butl
er</a>
the-brains:Beautiful Soup Tutorial jeriwieringa$ █
```

Successfully isolated only names and URLs

Stripping Tags and Writing Content to a CSV file

But, we are not done yet! There are still HTML tags surrounding the URL data that we want. And we need to save the data into a file in order to use it for other projects.

In order to clean up the HTML tags and split the URLs from the names, we need to isolate the information from the anchor tags. To do this, we will use two powerful, and commonly used Beautiful Soup methods: `contents` and `get`.

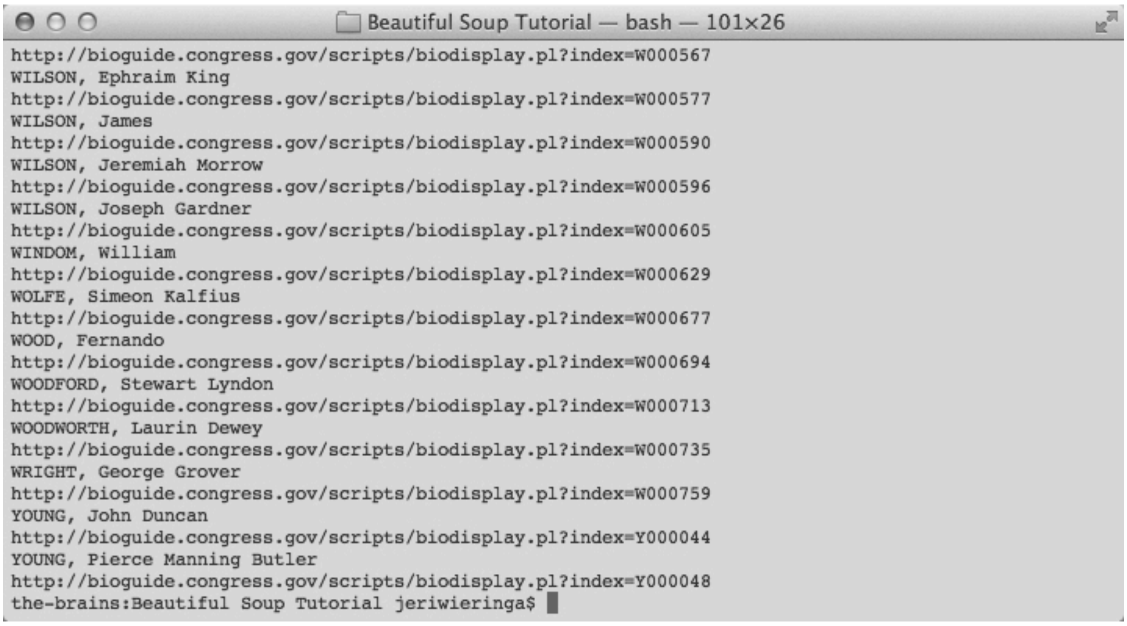
Where before we told the computer to print each link, we now want the computer to separate the link into its parts and print those separately. For the names, we can use `link.contents`. The "contents" method isolates out the text from within html tags. For example, if you started with

```
<h2>This is my Header text</h2>
```

You would be left with "This is my Header text" after applying the contents method. In this case, we want the contents inside the first tag in "link". (There is only one tag in "link", but since the computer doesn't realize that, we must tell it to use the first tag.)

For the URL, however, "contents" does not work because the URL is part of the HTML tag. Instead, we will use "get", which allow us to pull the text associated with (is on the other side of the "=" of) the "href" element.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

links = soup.find_all('a')
for link in links:
    names = link.contents[0]
    fullLink = link.get('href')
    print names
    print fullLink
```



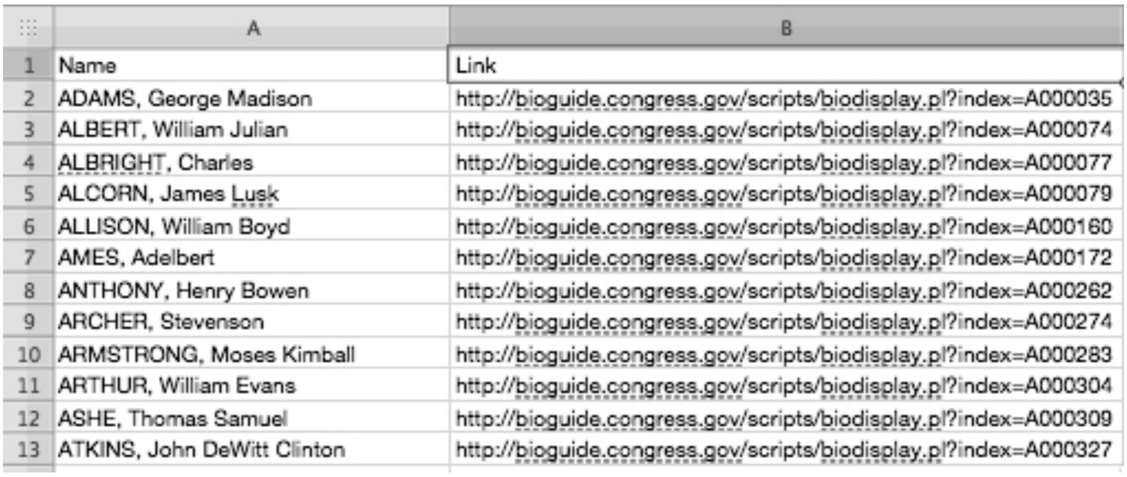
```
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000567
WILSON, Ephraim King
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000577
WILSON, James
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000590
WILSON, Jeremiah Morrow
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000596
WILSON, Joseph Gardner
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000605
WINDOM, William
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000629
WOLFE, Simeon Kalfius
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000677
WOOD, Fernando
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000694
WOODFORD, Stewart Lyndon
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000713
WOODWORTH, Laurin Dewey
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000735
WRIGHT, George Grover
http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000759
YOUNG, John Duncan
http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000044
YOUNG, Pierce Manning Butler
http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000048
the-brains:Beautiful Soup Tutorial jeriwieringa$
```

All HTML tags have been removed.

Finally, we want to use the CSV library to write the file. First, we need to import the CSV library into the script with “import csv.” Next, we create the new CSV file when we “open” it using “csv.writer”. The “w” tells the computer to “write” to the file. And to keep everything organized, let’s write some column headers. Finally, as each line is processed, the name and URL information is written to our CSV file.

```
from bs4 import BeautifulSoup
import csv

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

f = csv.writer(open("43rd_Congress.csv", "w"))
f.writerow(["Name", "Link"]) # Write column headers as the first line

links = soup.find_all('a')
for link in links:
    names = link.contents[0]
    fullLink = link.get('href')

    f.writerow([names, fullLink])
```

When executed, this gives us a clean CSV file that we can then use for other purposes.



	A	B
1	Name	Link
2	ADAMS, George Madison	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000035
3	ALBERT, William Julian	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000074
4	ALBRIGHT, Charles	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000077
5	ALCORN, James Lusk	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000079
6	ALLISON, William Boyd	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000160
7	AMES, Adelbert	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000172
8	ANTHONY, Henry Bowen	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000262
9	ARCHER, Stevenson	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000274
10	ARMSTRONG, Moses Kimball	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000283
11	ARTHUR, William Evans	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000304
12	ASHE, Thomas Samuel	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000309
13	ATKINS, John DeWitt Clinton	http://bioguide.congress.gov/scripts/biodisplay.pl?index=A000327

CSV file of results

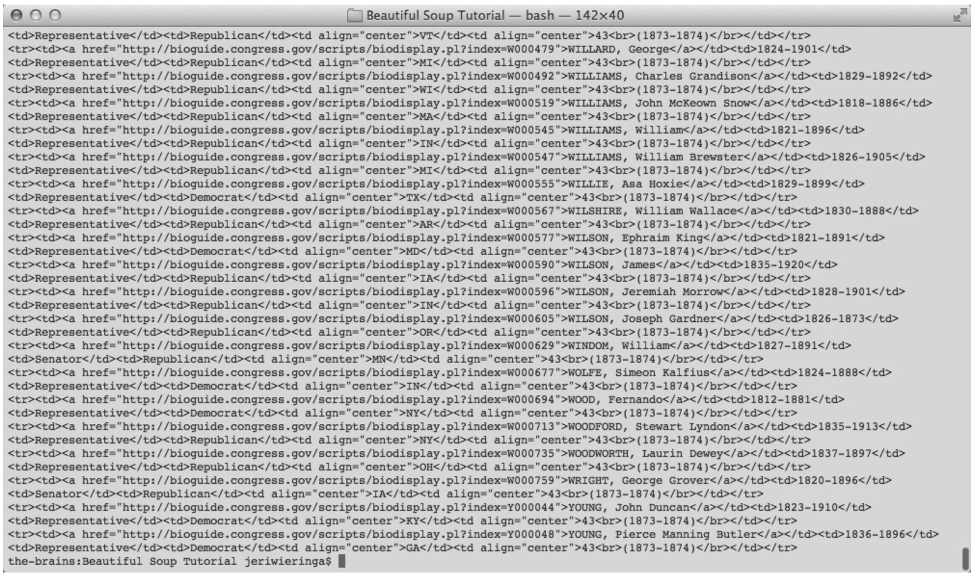
We have solved our puzzle and have extracted names and URLs from the HTML file.

But wait! What if I want ALL of the data?

Let's extend our project to capture all of the data from the webpage. We know all of our data can be found inside a table, so let's use "<tr>" to isolate the content that we want.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

trs = soup.find_all('tr')
for tr in trs:
    print tr
```

Looking at the print out in the terminal, you can see we have selected a lot more content than when we searched for "<a>" tags. Now we need to sort through all of these lines to separate out the different types of data.



```
<td>Representative</td><td>Republican</td><td align="center">VT</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000479">WILLARD, George</a></td><td>1824-1901</td>
<td>Representative</td><td>Republican</td><td align="center">MI</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000492">WILLIAMS, Charles Grandison</a></td><td>1829-1892</td>
<td>Representative</td><td>Republican</td><td align="center">WI</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000519">WILLIAMS, John McKeown Snow</a></td><td>1818-1886</td>
<td>Representative</td><td>Republican</td><td align="center">MA</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000545">WILLIAMS, William</a></td><td>1821-1896</td>
<td>Representative</td><td>Republican</td><td align="center">IN</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000547">WILLIAMS, William Brewster</a></td><td>1826-1905</td>
<td>Representative</td><td>Republican</td><td align="center">MI</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000555">WILLIE, Asa Hoxie</a></td><td>1829-1899</td>
<td>Representative</td><td>Democrat</td><td align="center">TX</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000567">WILSHIRE, William Wallace</a></td><td>1830-1888</td>
<td>Representative</td><td>Republican</td><td align="center">AR</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000577">WILSON, Ephraim King</a></td><td>1821-1891</td>
<td>Representative</td><td>Democrat</td><td align="center">MD</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000590">WILSON, James</a></td><td>1835-1920</td>
<td>Representative</td><td>Republican</td><td align="center">IA</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000596">WILSON, Jeremiah Morrow</a></td><td>1828-1901</td>
<td>Representative</td><td>Republican</td><td align="center">IN</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000605">WILSON, Joseph Gardner</a></td><td>1826-1873</td>
<td>Representative</td><td>Republican</td><td align="center">OR</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000629">WINDOM, William</a></td><td>1827-1891</td>
<td>Senator</td><td>Republican</td><td align="center">MN</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000677">WOLFE, Simeon Kalfius</a></td><td>1824-1888</td>
<td>Representative</td><td>Democrat</td><td align="center">IN</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000694">WOOD, Fernando</a></td><td>1812-1881</td>
<td>Representative</td><td>Democrat</td><td align="center">NY</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000713">WOODFORD, Stewart Lyndon</a></td><td>1835-1913</td>
<td>Representative</td><td>Republican</td><td align="center">NY</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000735">WOODWORTH, Laurin Dewey</a></td><td>1837-1897</td>
<td>Representative</td><td>Republican</td><td align="center">OH</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=W000759">WRIGHT, George Grover</a></td><td>1820-1896</td>
<td>Senator</td><td>Republican</td><td align="center">IA</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000044">YOUNG, John Duncan</a></td><td>1823-1910</td>
<td>Representative</td><td>Democrat</td><td align="center">KY</td><td align="center">43<br>(1873-1874)</br></td></tr>
<tr><td><a href="http://bioguide.congress.gov/scripts/biodisplay.pl?index=Y000048">YOUNG, Pierce Manning Butler</a></td><td>1836-1896</td>
<td>Representative</td><td>Democrat</td><td align="center">GA</td><td align="center">43<br>(1873-1874)</br></td></tr>
the-brains:Beautiful Soup Tutorial jeriwieringa$ 
```

All of the Table Row Data

Extracting the Data

We can extract the data in two moves. First, we will isolate the link information; then, we will parse the rest of the table row data.

For the first, let's create a loop to search for all of the anchor tags and "get" the data associated with "href".

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

trs = soup.find_all('tr')

for tr in trs:
    for link in tr.find_all('a'):
        fulllink = link.get ('href')
        print fulllink #print in terminal to verify results
```

We then need to run a search for the table data within the table rows. (The "print" here allows us to verify that the code is working but is not necessary.)

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

trs = soup.find_all('tr')

for tr in trs:
    for link in tr.find_all('a'):
        fulllink = link.get ('href')
        print fulllink #print in terminal to verify results

    tds = tr.find_all("td")
    print tds
```

Next, we need to extract the data we want. We know that everything we want for our CSV file lives within table data ("td") tags. We also know that these items appear in the same order within the row. Because we are dealing with lists, we can identify information by its position within the list. This means that the first data item in the row is identified by [0], the second by [1], etc.

Because not all of the rows contain the same number of data items, we need to build in a way to tell the script to move on if it encounters an error. This is the logic of the "try" and "except" block. If a particular line fails, the script will continue on to the next line.

```
from bs4 import BeautifulSoup

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()
trs = soup.find_all('tr')

for tr in trs:
    for link in tr.find_all('a'):
        fulllink = link.get ('href')
        print fulllink #print in terminal to verify results

    tds = tr.find_all("td")

    try: #we are using "try" because the table is not well formatted. This allows th
e program to continue after encountering an error.
        names = str(tds[0].get_text()) # This structure isolate the item by its colu
mn in the table and converts it into a string.
        years = str(tds[1].get_text())
        positions = str(tds[2].get_text())
        parties = str(tds[3].get_text())
        states = str(tds[4].get_text())
        congress = tds[5].get_text()

    except:
        print "bad tr string"
        continue #This tells the computer to move on to the next item after it encou
nters an error

    print names, years, positions, parties, states, congress
```

Within this we are using the following structure:

```
years = str(tds[1].get_text())
```

We are applying the "get_text" method to the 2nd element in the row (because computers count beginning with 0) and creating a string from the result. This we assign to the variable "years", which we will use to create the CSV file. We repeat this for every item in the table that we want to capture in our file.

Writing the CSV file

The last step in this file is to create the CSV file. Here we are using the same process as we did in Part I, just with more variables.

As a result, our file will look like:

```
from bs4 import BeautifulSoup
import csv

soup = BeautifulSoup (open("43rd-congress.html"))

final_link = soup.p.a
final_link.decompose()

f= csv.writer(open("43rd_Congress_all.csv", "w"))   # Open the output file for writi
ng before the loop
f.writerow(["Name", "Years", "Position", "Party", "State", "Congress", "Link"]) # Wr
ite column headers as the first line

trs = soup.find_all('tr')

for tr in trs:
    for link in tr.find_all('a'):
        fullLink = link.get ('href')

    tds = tr.find_all("td")

    try: #we are using "try" because the table is not well formatted. This allows th
e program to continue after encountering an error.
        names = str(tds[0].get_text()) # This structure isolate the item by its colu
mn in the table and converts it into a string.
        years = str(tds[1].get_text())
        positions = str(tds[2].get_text())
        parties = str(tds[3].get_text())
        states = str(tds[4].get_text())
        congress = tds[5].get_text()

    except:
        print "bad tr string"
        continue #This tells the computer to move on to the next item after it encou
nters an error

    f.writerow([names, years, positions, parties, states, congress, fullLink])
```

You've done it! You have created a CSV file from all of the data in the table, creating useful data from the confusion of the html page.

About the Author

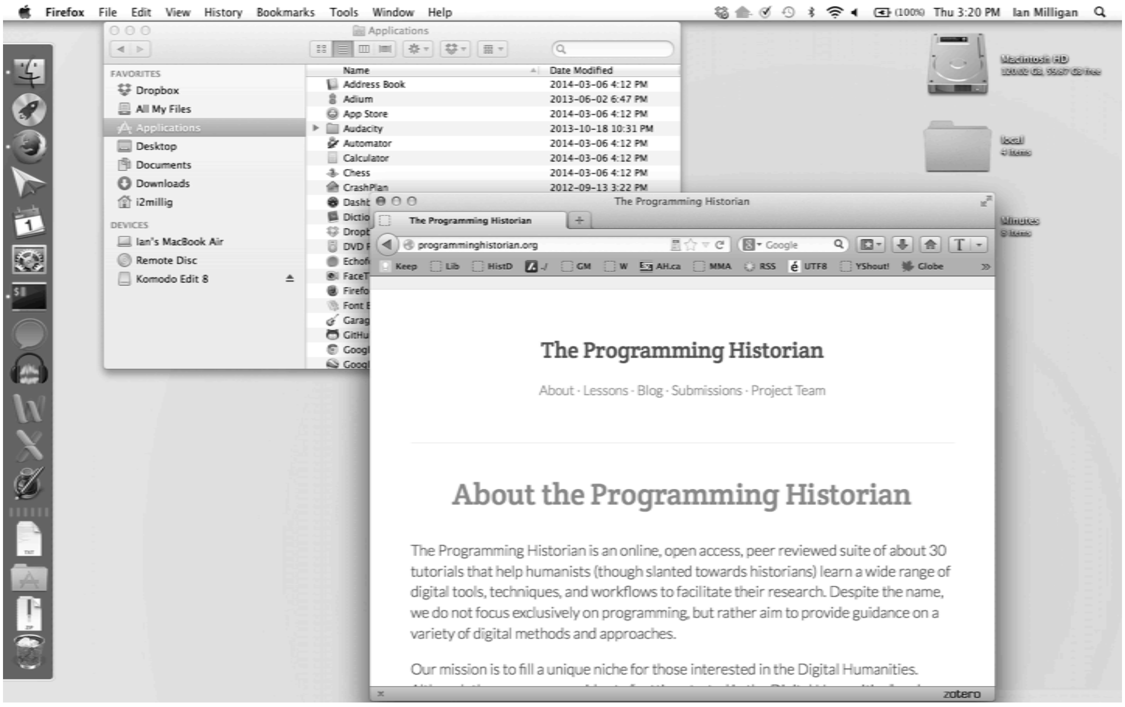
Jeri Wieringa is the digital publishing production lead for the George Mason University Libraries and a doctoral candidate in history at George Mason University.

22. Introduction to the Bash Command Line

Ian Milligan and James Baker – 2014

Introduction

Many of the lessons at the *Programming Historian* require you to enter commands through a **Command-Line Interface**. The usual way that computer users today interact with their system is through a **Graphical-User Interface**, or GUI. This means that when you go into a folder, you click on a picture of a file folder; when you run a program, you click on it; and when you browse the web, you use your mouse to interact with various elements on a webpage. Before the rise of GUIs in the late 1980s, however, the primary way to interact with a computer was through a command-line interface.



Name	Date Modified
Address Book	2014-03-06 4:12 PM
Adium	2013-06-02 6:47 PM
App Store	2014-03-06 4:12 PM
Audacity	2013-10-18 10:31 PM
Automator	2014-03-06 4:12 PM
Calculator	2014-03-06 4:12 PM
Chess	2014-03-06 4:12 PM
CrashPlan	2012-09-13 3:22 PM

The Programming Historian

About · Lessons · Blog · Submissions · Project Team

About the Programming Historian

The Programming Historian is an online, open access, peer reviewed suite of about 30 tutorials that help humanists (though slanted towards historians) learn a wide range of digital tools, techniques, and workflows to facilitate their research. Despite the name, we do not focus exclusively on programming, but rather aim to provide guidance on a variety of digital methods and approaches.

Our mission is to fill a unique niche for those interested in the Digital Humanities.

GUI of Ian Milligan's Computer

Command-line interfaces have advantages for computer users who need more precision in their work -- such as digital historians. They allow for more detail when running some programs, as you can add modifiers to specify *exactly* how you want your program to run. Furthermore, they can be easily automated through scripts,[188] which are essentially recipes of text-based commands.

There are two main command-line interfaces, or 'shells,' that many digital historians use. On OS X or many Linux installations, the shell is known as `bash`, or the 'Bourne-again shell.' For users on Windows-based systems, the command-line interface is by default `MS-DOS-based`, which uses different commands and syntax,[189] but can often achieve similar tasks. This tutorial provides a basic introduction to the `bash` terminal, and Windows users can follow along by installing popular shells such as Cygwin[190] or Git Bash (see below).

This lesson uses a `Unix shell`,[191] which is a command-line interpreter that provides a user interface for the Unix operating system[192] and for Unix-like systems. This lesson will cover a small number of basic commands. By the end of this tutorial you will be able to navigate through your file system and find files, open them, perform basic data manipulation tasks such as combining and copying files, as well as both reading them and making relatively simple edits. These commands constitute the building blocks upon which more complex commands can be constructed to fit your research data or project. Readers wanting a reference guide that goes beyond this lesson are recommended to read Deborah S. Ray and Eric J. Ray, *Unix and Linux: Visual Quickstart Guide*, 4th edition (2009).

Windows Only: Installing Git Bash

For those on OS X, and most Linux installations, you're in luck — you already have a bash shell installed. For those of you on Windows, you'll need to take one extra step and install Git Bash. This can be installed by downloading the most recent 'Full installer' at the the msysgit featured downloads list.[193] Instructions for installation are available at Open Hatch.[194]

Opening Your Shell

Let's start up the shell. In Windows, run Git Bash from the directory that you installed it in. You will have to run it as an administrator - to do so, right click on the program and select 'Run as Administrator.' In OS X, by default the shell is located in:

```
Applications -> Utilities -> Terminal
```

[188] See 'Chapter 1. Bash and Bash scripts', *Bash Guide for Beginners*: http://www.tldp.org/LDP/Bash-Beginners-Guide/html/chap_01.html

[189] 'Syndax', *Wikipedia*: https://en.wikipedia.org/wiki/Syntax

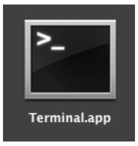
[190] 'Cygwin': https://www.cygwin.com/

[191] 'Unix Shell', *Wikipedia*: https://en.wikipedia.org/wiki/Unix_shell

[192] 'Unix', *Wikipedia*: https://en.wikipedia.org/wiki/Unix

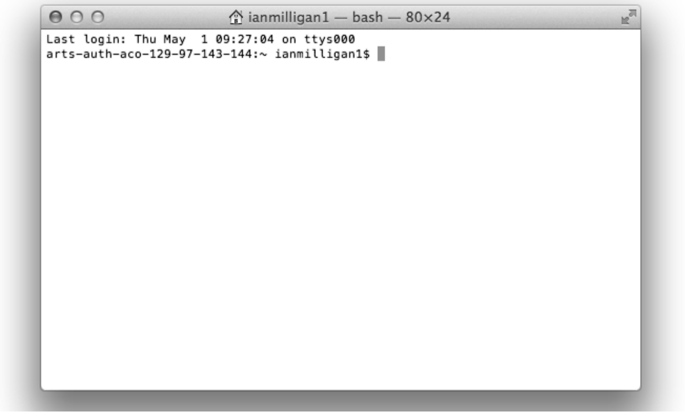
[193] 'Git-1.9.5': https://github.com/msysgit/msysgit/releases/

[194] 'Install Git Bash', *Open Hatch*: https://openhatch.org/missions/windows-setup/install-git-bash



Termina.app program on OS X

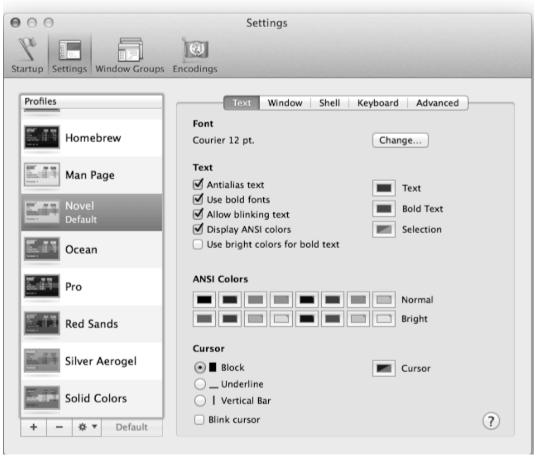
When you run it, you will see this window.



```
Last login: Thu May  1 09:27:04 on ttys000
arts-auth-aco-129-97-143-144:~ ianmilligan1$ 
```

A blank terminal screen on our OS X workstation

You might want to change the default visual appearance of the terminal, as eyes can strain at repeatedly looking at black text on a white background. In the default OS X application, you can open the 'Settings' menu in 'Preferences' under Terminal. Click on the 'Settings' tab and change it to a new colour scheme. We personally prefer something with a bit less contrast between background and foreground, as you'll be staring at this a great deal. 'Novel' is a soothing one as is the popular Solarized suite of colour palettes.[195] For Windows users, a similar effect can be achieved using the Git Bash `Properties` tab. To reach this, right-click anywhere in the top bar and select `Properties`.



Font

Courier 12 pt.

Text

☑ Antialias text
☑ Use bold fonts
☑ Allow blinking text
☑ Display ANSI colors
□ Use bright colors for bold text

ANSI Colors

Cursor

● Block
○ Underline
○ Vertical Bar
□ Blink cursor

The Settings Screen on the OS X Terminal Shell Application

[195] 'Solarized': http://ethanschoonover.com/solarized

Once you are happy with the interface, let's get started.

Moving Around Your Computer's File System

If, when opening a command window, you are unsure of where you are in a computer's file system, the first step is to find out what directory you are in. Unlike in a graphical system, when in a shell you cannot be in multiple directories at once. When you open up your file explorer on your desktop, it's revealing files that are within a directory. You can find out what directory you are in through the `pwd` command, which stands for "print working directory." Try inputting:

```
pwd
```

and hitting enter. If you're on OS X or Linux, your computer will probably display `/users/USERNAME` with your own user name in place of USERNAME. For example, Ian's path on OS X is `/users/ianmilligan1/`.

Here is where you realize that those on Windows and those on OS X/Linux will have slightly different experiences. On Windows, James is at:

```
c/users/jbaker
```

There are minor differences, but fear not; once you're moving and manipulating files, these platform divergences can fade into the background.

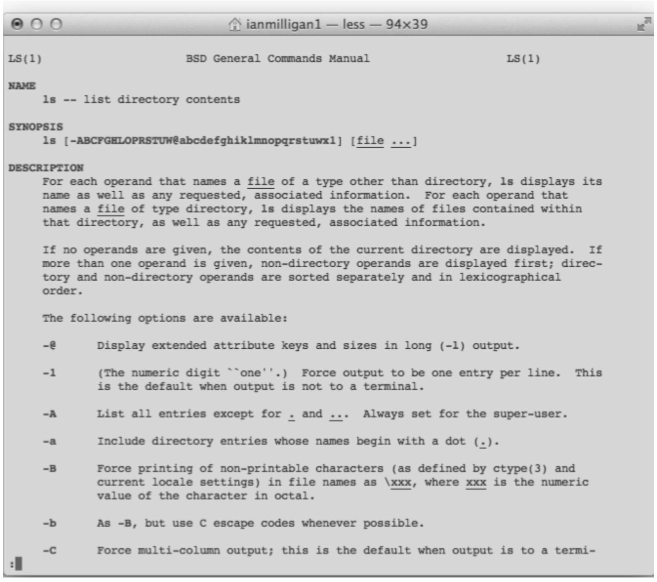
To orient ourselves, let's get a listing of what files are in this directory. Type

```
ls
```

and you will see a list of every file and directory within your current location. Your directory may be cluttered or it may be pristine, but you will at a minimum see some familiar locations. On OS X, for example, you'll see `Applications`, `Desktop`, `Documents`, `Downloads`, `Library`, `Pictures`, etc.

You may want more information than just a list of files. You can do this by specifying various *flags* to go with our basic commands. These are additions to a command that provide the computer with a bit more guidance of what sort of output or manipulation you want. To get a list of these, OS X/Linux users can turn to the built-in help program. OS X/Linux users type

```
man ls
```



```
LS(1)                     BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls displays its
     name as well as any requested, associated information.  For each operand that
     names a file of type directory, ls displays the names of files contained within
     that directory, as well as any requested, associated information.

     If no operands are given, the contents of the current directory are displayed.  If
     more than one operand is given, non-directory operands are displayed first; direc-
     tory and non-directory operands are sorted separately and in lexicographical
     order.

     The following options are available:

     -@      Display extended attribute keys and sizes in long (-l) output.

     -1      (The numeric digit ``one''.)  Force output to be one entry per line.  This
             is the default when output is not to a terminal.

     -A      List all entries except for . and ...  Always set for the super-user.

     -a      Include directory entries whose names begin with a dot (.).

     -B      Force printing of non-printable characters (as defined by ctype(3) and
             current locale settings) in file names as \xxx, where xxx is the numeric
             value of the character in octal.

     -b      As -B, but use C escape codes whenever possible.

     -C      Force multi-column output; this is the default when output is to a termi-
:
```

The Manual page for the LS command

Here, you see a listing of the name of the command, the way that you can format this command and what it does. **Many of these will not make sense at this stage, but don't worry; over time you will become more familiar with them.** You can explore this page in a variety of ways: the spacebar moves down a page, or you can arrow down and arrow up throughout the document.

To leave the manual page, press

```
q
```

and you will be brought back to the command line where you were before entering the manual page.

Try playing around with the `man` page for the other command you have learned so far, `pwd`.

Windows users can use the `help` command, though this command has fewer features than `man` on OS X/Linux. Enter `help` to see the help available, and `help pwd` for an example of the command's output.

Let's try using a few of those options you saw in the `man` page for ls. Perhaps you only want to see TXT files that are in our home directory. Type

```
ls *.txt
```

which returns a list of text files, if you have any in your home directory (you may not, and that is OK as well). The * command is a **wildcard** — it stands for 'anything.' So, in this case, you're indicating that anything that fits the pattern:

[anything.txt]

will be displayed. Try out different combinations. If, for example, you had several files in the format `1-Canadian.txt`, `2-Canadian.txt`, and so forth, the command `ls *-Canadian.txt` would display them all but exclude all other files (those that do not match the pattern).

Say you want more information. In that long `man` page, you saw an option that might be useful:

```
-l      (The lowercase letter ``ell''.)  List in long format.  (See below.)
If
        the output is to a terminal, a total sum for all the file sizes is
out-
        put on a line before the long listing.
```

So, if you type

```
ls -l
```

the computer returns a long list of files that contains information similar to what you'd find in your finder or explorer: the size of the files in bites, the date it was created or last modified, and the file name. However, this can be a bit confusing: you see that a file test.html is '6020' bits large. In commonplace language, you are more used to units of measurement like bytes, kilobytes, megabytes, and gigabytes.

Luckily, there's another flag:

```
-h      When used with the -l option, use unit suffixes: Byte, Kilobyte,
        Megabyte, Gigabyte, Terabyte and Petabyte in order to reduce the
number
        of digits to three or less using base 2 for sizes.
```

When you want to use two flags, you can just run them together. So, by typing

```
ls -lh
```

you receive output in a human-readable format; you learn that that 6020 bits is also 5.9KB, that another file is 1 megabyte, and so forth.

These options are *very* important. In other lessons within the *Programming Historian*, you'll see them. 'Wget', 'MALLET", and 'Pandoc' all use the same syntax.[196] Luckily, you do not need to memorize syntax; instead, keep these lessons handy so you can take a quick peek if you need to tweak something. These lessons can all be done in any order.

You've now spent a great deal of time in your home directory. Let's go somewhere else. You can do that through the `cd` or Change Directory command.

[196] See: Kellen Kurchinski, 'Applied Archival Downloading with Wget' (2013); Shawn Graham, Scott Weingart, and Ian Milligan, 'Getting Started with Topic Modeling and MALLET' (2012); and Dennis Tenen and Grant Wythoff, 'Sustainable Authorship in Plain Text using Pandoc and Markdown' *The Programming Historian* (2014).

If you type

```
cd desktop
```

you are now on your desktop. This is akin to you 'double-clicking' on the 'desktop' folder within a file explorer. To double check, type `pwd` and you should see something like:

```
/Users/ianmilligan1/desktop
```

Try playing around with those earlier commands: explore your current directory using the `ls` command.

If you want to go back, you can type

```
cd ..
```

This moves us 'up' one directory, putting us back in `/Users/ianmilligan1/`. If you ever get completely lost, the command

```
cd --
```

will bring you right back to the home directory, right where you started.

Try exploring: visit your documents directory, your pictures, folders you might have on your desktop. Get used to moving in and out of directories. Imagine that you are navigating a tree structure.[197] If you're on the desktop, you won't be able to `cd documents` as it is a 'child' of your home directory, whereas your Desktop is a 'sibling' of the Documents folder. To get to a sibling, you have to go back to the common parent. To do this, you will have to back up to your home directory (`cd ..`) and then go forward again to `cd documents`.

Being able to navigate your file system using the bash shell is very important for many of the lessons at the *Programming Historian*. As you become more comfortable, you'll soon find yourself skipping directly to the directory that you want. In our case, from anywhere on our system, you could type

```
cd /users/ianmilligan1/mallet-2.0.7
```

or, on Windows, something like

```
cd c:\mallet-2.0.7\
```

and be brought to our MALLET directory for topic modeling.[198]

Finally, try

```
open .
```

[197] 'Tree Structure', *Wikipedia*: https://en.wikipedia.org/wiki/Tree_structure

[198] Shawn Graham, Scott Weingart, and Ian Milligan, 'Getting Started with Topic Modeling and MALLET' *The Programming Historian* (2012).

in OS X or

```
explorer .
```

in Windows. That command will open up your GUI at the current directory. Make sure to leave a space between `open` or `explorer` and the period.

Interacting with Files

As well as navigating directories, you can interact with files on the command line: you can read them, open them, run them, and even edit them, often without ever having to leave the interface. There is some debate over why one would do this. The primary reason is the seamless experience of working on the command line: you never have to pick up your mouse or touch your track pad, and, although it has a steep learning curve it can eventually become a sole writing environment. Furthermore, many programs require you to use the command line to operate with them. Since you'll be using programs on the command line, it can often be quicker to make small edits without switching into a separate program. For some of these arguments, see Jon Beltran de Heredia's "Why, oh WHY, do those #?@! nutheads use vi?".[199]

Here's a few basic ways to do interact with files.

First, you can create a new directory so you can engage with text files. We will create it on your desktop, for convenience's sake. You can always move it later. Navigate to your desktop using your shell, and type:

```
mkdir ProgHist-Text
```

This creates a directory named, you guessed it, '`ProgHist-Text`.' In general, it's good to avoid putting spaces in your filenames and directories when using the command line (there are workarounds, of course, but this approach is simpler). You can look at your desktop to verify it has worked. Now, move into that directory (remember, that would be `cd ProgHist-Text`).

But wait! There's a trick to make things a bit quicker. Go up one directory (`cd ..` - which will take you back to the Desktop). To navigate to the `ProgHist-Text` directory you could type `cd ProgHist-Text`. Alternatively, you could type `cd Prog` and then hit tab. You will notice that the interface completes the line to `cd ProgHist-Text`. Hitting tab at any time within the shell will prompt it to attempt to auto-complete the line based on the files or sub-directories in the current directory. This is case sensitive, however (i.e. in the previous example, `cd prog` would not auto complete to `ProgHist-Text`. Where two or more files have the same characters, the auto-complete will only fill up to the first point of difference. We would encourage using this method throughout the lesson to see how it behaves.

[199] Jon Beltran de Heredia, 'Why, oh WHY do those #?@! Nutheads use vi?', *ViEmu* (16 May 2007): http://www.viemu.com/a-why-vi-vim.html

Now you need to find a basic text file to help us with the example. Why don't you use a book that you know is long, such as Leo Tolstoy's epic *War and Peace*. The text file is availiable via Project Gutenberg.[200] If you have already installed wget,[201] you can just type:

```
wget http://www.gutenberg.org/cache/epub/2600/pg2600.txt
```

If you do not have wget installed, download the text itself using your browser. Go to the link above, and, in your browser, use the 'Save Page as..' command in your 'file menu.' Save it in your new 'ProgHist-Text directory.' Now, when you type

```
ls -lh
```

you see

```
-rw-r--r--+ 1 ianmilligan1 staff 3.1M 1 May 10:03 pg2600.txt
```

You can read the text within this file in a few different ways. First, you can tell our computer that you want to read it using the standard program that you use to open text files. By default, this may be TextEdit on OS X or Notepad in Windows. To open a file, just type

```
open pg2600.txt
```

on OS X, or

```
explorer pg2600.txt
```

in Windows.

This selects the default program to open that type of file, and opens it.

However, you often want to just work on the command line without leaving it. You can read files within this environment as well. To try this, type:

```
cat pg2600.txt
```

The terminal window erupts and *War and Peace* cascades by. That's great, in theory, but you can't really make any sense of that amount of text? Instead, you may want to just look at the first or the last bit of the file.

```
head pg2600.txt
```

Provides a view of the first ten lines, whereas

```
tail pg2600.txt
```

provides a perspective on the last ten lines. This is a good way to quickly determine the contents of the file. You could add a command to change the

[200] graf Leo Tolstoy, 'War and Peace', *Project Gutenberg*: http://www.gutenberg.org/ebooks/2600?msg=welcome_stranger

[201] Kellen Kurchinski, 'Applied Archival Downloading with Wget', *The Programming Historian* (2013).

amount of lines displayed: `head -20 pg2600.txt`, for example, would show the first twenty lines.

You may also want to change the file name to something more descriptive. You can 'move' it to a new name by typing

```
mv pg2600.txt tolstoy.txt
```

Afterwards, when you perform a `ls` command, you will see that it is now `tolstoy.txt`. Had you wanted to duplicate it, you could also have run the copy command by typing

```
cp pg2600.txt tolstoy.txt
```

you will revisit these commands shortly.

Now that you have used several new commands, it's time for another trick. Hit the up arrow on your keyboard. Notice that `cp pg2600.txt tolstoy.txt` appears before your cursor. You can continue pressing the up arrow to cycle through your previous commands. The down arrow cycles back toward your most recent command.

After having read and renamed several files, you may wish to bring their text together into one file. To combine, or concatenate, two or more files, you can use the `cat` command. First, let's duplicate the Tolstoy file (`cp tolstoy.txt tolstoy2.txt`). Now that you have two copies of *War and Peace*, let's put them together to make an **`even longer`** book.

To combine, or concatenate, two or more files use the `cat` command. Type

```
cat tolstoy.txt tolstoy2.txt
```

and press enter. This prints, or displays, the combined files within the shell. However, it is too long to read on this window! Luckily, by using the `>` command, you can send the output to a new file, rather than the terminal window. Type

```
cat tolstoy.txt tolstoy2.txt > tolstoy-twice.txt.
```

Now, when you type `ls` you'll see `tolstoy-twice.txt` appear in your directory.

When combining more than two files, using a wildcard can help avoid having to write out each filename individually. As you have seen above, *, is a place holder for zero or more characters or numbers. So, if you type

```
cat *.txt > everything-together.txt
```

and hit enter, a combination of all the .txt files in the current directory are combined in alphabetical order as `everything-together.txt`. This can be very useful if you need to combine a large number of smaller files within a directory so that you can work with them in a text analysis program.

Another wildcard worth remembering is ? which is a place holder for a single character or number.

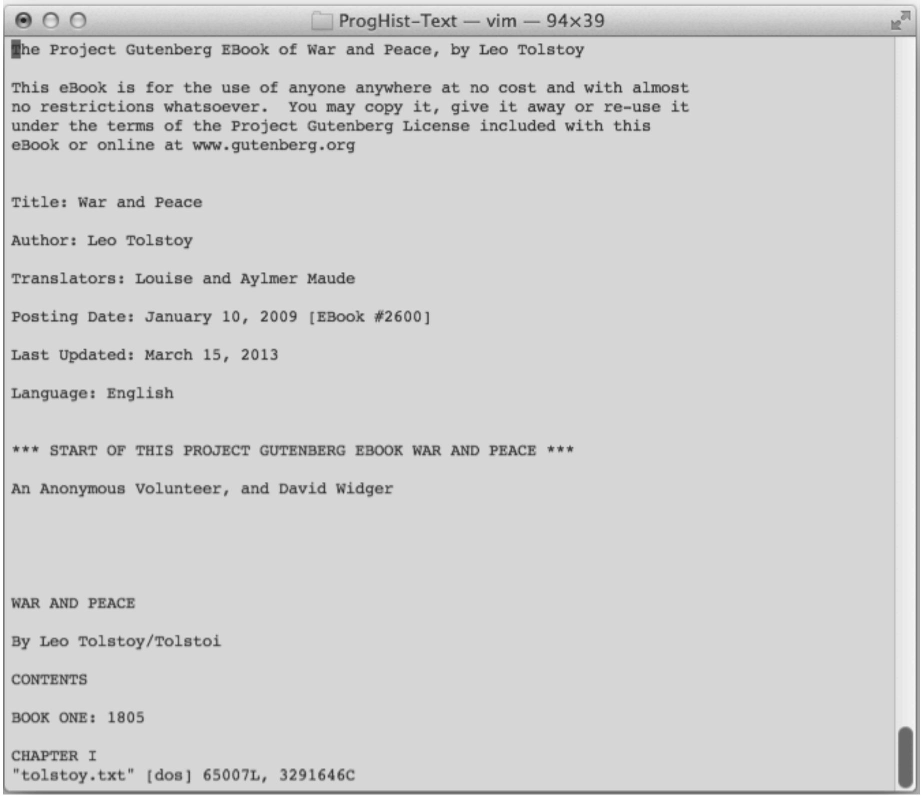
Editing Text Files Directly on the Command Line

If you want to read a file in its entirety without leaving the command line, you can fire up vim.[202] Vim is a very powerful text editor, which is perfect for using with programs such as Pandoc[203] to do word processing, or for editing your code without having to switch to another program. Best of all, it comes included with bash on both OS X and Windows. Vim has a fairly steep learning curve, so we will just touch on a few minor points.

Type

```
vim tolstoy.txt
```

You should see vim come to life before you, a command-line based text editor.



```
The Project Gutenberg EBook of War and Peace, by Leo Tolstoy

This eBook is for the use of anyone anywhere at no cost and with almost
no restrictions whatsoever.  You may copy it, give it away or re-use it
under the terms of the Project Gutenberg License included with this
eBook or online at www.gutenberg.org


Title: War and Peace

Author: Leo Tolstoy

Translators: Louise and Aylmer Maude

Posting Date: January 10, 2009 [EBook #2600]

Last Updated: March 15, 2013

Language: English


*** START OF THIS PROJECT GUTENBERG EBOOK WAR AND PEACE ***

An Anonymous Volunteer, and David Widger





WAR AND PEACE

By Leo Tolstoy/Tolstoi

CONTENTS

BOOK ONE: 1805

CHAPTER I
"tolstoy.txt" [dos] 65007L, 3291646C
```

Vim

If you really want to get into Vim, there is a good Vim guide available.[204]

[202] 'Vim (text editor)', *Wikipedia*: https://en.wikipedia.org/wiki/Vim_%28text_editor%29

[203] 'Pandoc': http://pandoc.org/

[204] 'Vim Documentation: quickref': http://vimdoc.sourceforge.net/htmldoc/quickref.html

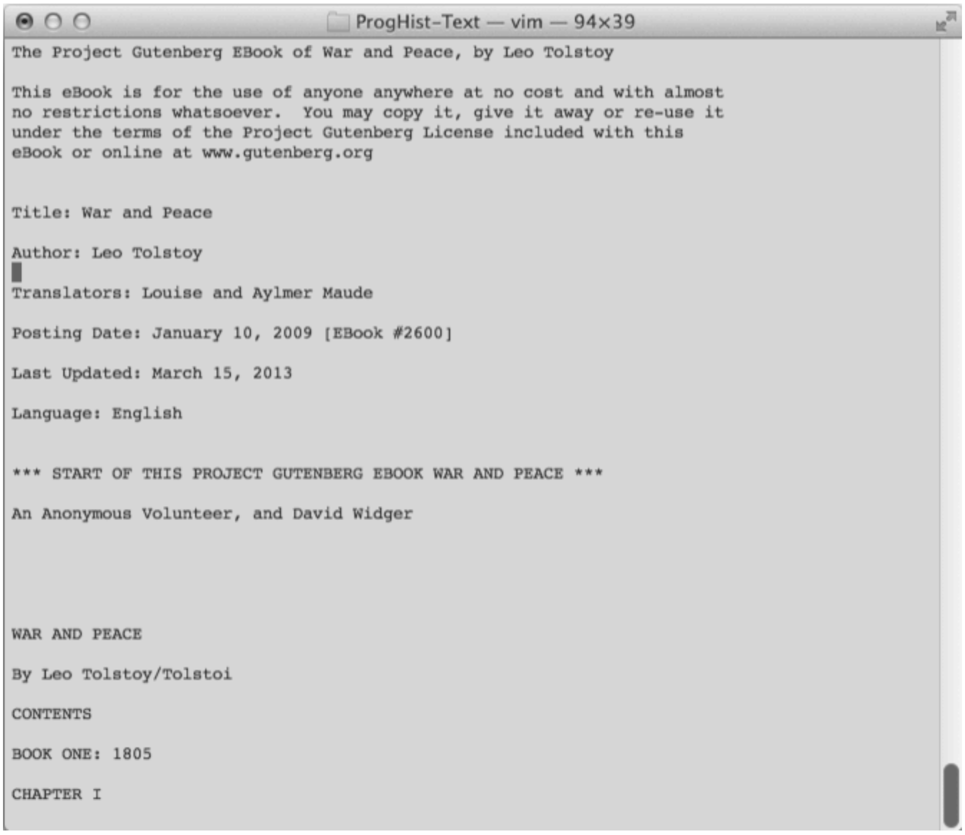
Using Vim to read files is relatively simple. You can use the arrow keys to navigate around and could theoretically read *War and Peace* through the command line (one should get an achievement for doing that). Some quick basic navigational commands are as follows:

`Ctrl+F` (that is, holding down your 'control key' and pressing the letter F) will move you down a page (`Shift+UpArrow` for Windows).

`Ctrl+B` will move you up a page. (`Shift+DownArrow` for Windows users).

If you want to rapidly move to the end of a line, you can press: `$` and to move to the start of one, `0`. You can also move between sentences by typing `)` (forward) or `(` (backwards). For paragraphs, use `}` and `{`. Since you are doing everything with your keyboard, rather than having to hold your arrow key down to move around a document, this lets you zip quickly back and forth.

Let's scroll to the top and do a minor change, such as adding a `Reader` field in the heading. Move your cursor in between **Author:** and **Translators:**, like so:



```
The Project Gutenberg EBook of War and Peace, by Leo Tolstoy

This eBook is for the use of anyone anywhere at no cost and with almost
no restrictions whatsoever.  You may copy it, give it away or re-use it
under the terms of the Project Gutenberg License included with this
eBook or online at www.gutenberg.org


Title: War and Peace

Author: Leo Tolstoy
█
Translators: Louise and Aylmer Maude

Posting Date: January 10, 2009 [EBook #2600]

Last Updated: March 15, 2013

Language: English


*** START OF THIS PROJECT GUTENBERG EBOOK WAR AND PEACE ***

An Anonymous Volunteer, and David Widger




WAR AND PEACE

By Leo Tolstoy/Tolstoi

CONTENTS

BOOK ONE: 1805

CHAPTER I
```

About to Insert a Field

If you just start typing, you'll get an error message or the cursor will begin jumping around. This is because you have to specify that you want to do an edit. Press the letter

```
a
```

At the bottom of the screen, you will see

```
-- INSERT --
```

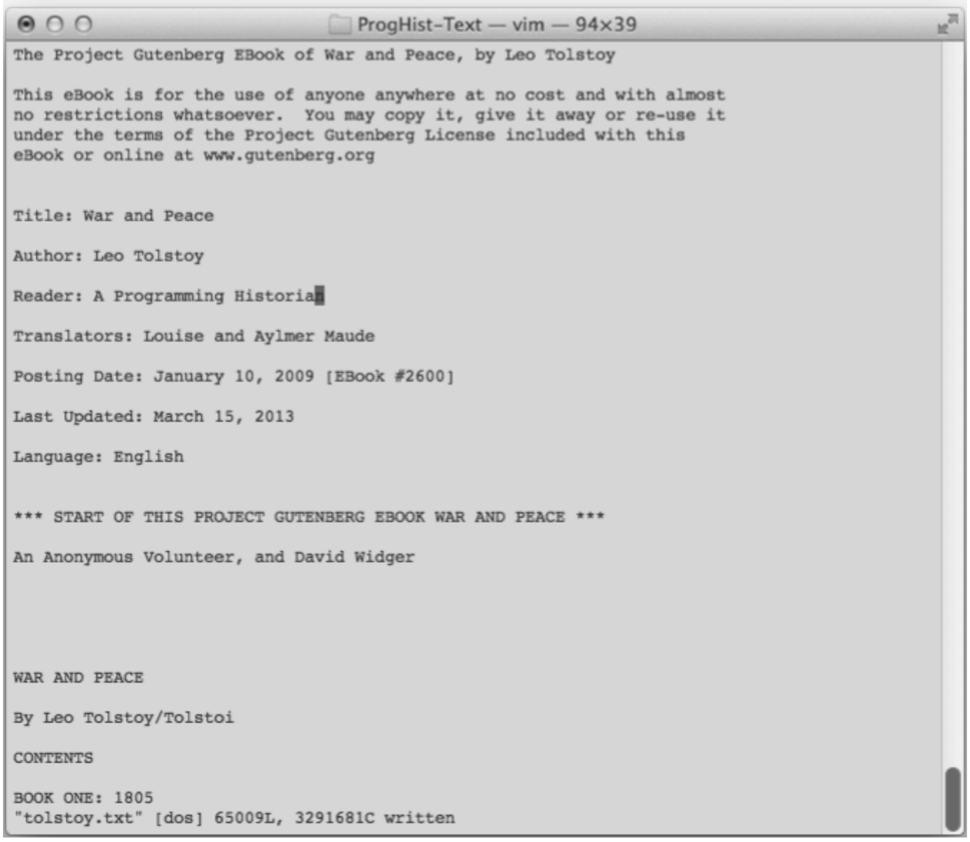
This means you are in insert mode. You can now type and edit text as if you are in a standard text editor. Press `enter` twice, then `arrow up`, and type

```
Reader: A Programming Historian
```

When you are done, press `ESC` to return to reading mode.

To leave vim or to make saves, you have to enter a series of commands. Press `:` and you'll move to the command input line of Vim. you can enter a variety of commands here. If you want to save the file, type `w` to 'write' the file. If you execute that command, you will see

"tolstoy.txt" [dos] 65009L, 3291681C written



```
The Project Gutenberg EBook of War and Peace, by Leo Tolstoy

This eBook is for the use of anyone anywhere at no cost and with almost
no restrictions whatsoever.  You may copy it, give it away or re-use it
under the terms of the Project Gutenberg License included with this
eBook or online at www.gutenberg.org


Title: War and Peace

Author: Leo Tolstoy

Reader: A Programming Historian

Translators: Louise and Aylmer Maude

Posting Date: January 10, 2009 [EBook #2600]

Last Updated: March 15, 2013

Language: English


*** START OF THIS PROJECT GUTENBERG EBOOK WAR AND PEACE ***

An Anonymous Volunteer, and David Widger




WAR AND PEACE

By Leo Tolstoy/Tolstoi

CONTENTS

BOOK ONE: 1805
"tolstoy.txt" [dos] 65009L, 3291681C written
```

After Writing the File, with our minor change

If you want to quit, type `:` again and then `q`. It will return you to the command line. As with the rest of bash, you could have also combined the two commands. Pressing `:` and then typing `wq` would have written the file and then quit. Or, if you wanted to exit **`without`** saving, `q!` would have quit vim and overriden the default preference to save your changes.

Vim is different than you are likely used to and will require more work and practice to become fluent with it. But if you are tweaking minor things in files, it is a good way to get started. As you become more comfortable, you might even find yourself writing term papers with it, by harnessing the footnoting and formatting power of Pandoc and Markdown.[205]

Moving, Copying, and Deleting Files

Let's say you are done with this directory, and you would like to move `tolstoy.txt` somewhere else. First, you should create a backup copy. The shell is quite unforgiving with mistakes, and backing up is even more important than with GUIs. If you delete something here, there's no recycling bin to fish it out of. To create a backup, you can type

```
cp tolstoy.txt tolstoy-backup.txt
```

Now when you run a `ls` command you will see five files, two of which are the same: `tolstoy.txt` and `tolstoy-backup.txt`.

Let's move the first of these somewhere else. By way of example, let's create a second directory on your desktop. Move up to your desktop (`cd ..`) and `mkdir` another directory. Let's call it `proghist-dest`.

To copy `tolstoy.txt` you have a few different options. you could run these commands from anywhere in the shell, or you could visit either the origin or destination directories. For this example, let's just run it from here. The basic format of the copy command is `cp [source] [destination]`. That is, you type `cp` first, and then enter the file or files that you want to copy followed by where they should go.

In this case, the command

```
cp /users/ianmilligan1/desktop/proghist-text/tolstoy.txt
/users/ianmilligan1/desktop/proghist-dest/
```

will copy Tolstoy from the first directory to the second directory. You will have to insert your own username in place of 'ianmilligan1'. This means you now have three copies of the novel on our computer. The original, the backup and the new copy in the second directly. If you wanted to **move** the file, that is, not leave a copy behind, you could run the command again, swapping `cp` for `mv`; let's not do this yet.

You can also copy multiple files with a single command. If you wanted to copy **both** the original and the backup file, you could use the wildcard command.

```
cp /users/ianmilligan1/desktop/proghist-text/*.txt
/users/ianmilligan1/desktop/proghist-dest/
```

[205] See: Dennis Tenen and Grant Wythoff, 'Sustainable Authorship in Plain Text using Pandoc and Markdown', *The Programming Historian* (2014).

This command copies **all** the text files from the origin directory into the destination directory.

Note: If you are in the directory that you either want to move things to or from, you do not have to type out the whole directory structure. Let's do two quick examples. Change your directory to the `proghist-text` directory. From this location, if you wanted to copy these two files to `proghist-dest`, this command would work:

```
cp *.txt /users/ianmilligan1/desktop/proghist-dest/
```

(on OS X, substitute the directory on Windows)

Alternatively, if you were in the `proghist-dest` directory, this command would work:

```
cp /users/ianmilligan1/desktop/proghist-text/*.txt ./
```

The `./` command refers to the **current** directory you're in. **This is a really valuable command.**

Finally, if you want to delete a file, for whatever reason, the command is `rm`, or remove. **Be careful with the `rm` command**, as you don't want to delete files that you do not mean to. Unlike deleting from within your GUI, there is **no** recycling bin or undo options. For that reason, if you are in doubt, you may want to exercise caution or maintain a regular backup of your data.

Move to `proghist-text` and delete the original file by typing

```
rm tolstoy.txt
```

Check that the file is gone using the `ls` command.

If you wanted to delete an entire directory, you have two options. you can use `rmdir`, the opposite of `mkdir`, to delete an **empty** directory. To delete a directory with files, you could use from the desktop:

```
rm -r proghist-text
```

Conclusions

You may want to take a break from the terminal at this point. To do so, enter `exit` and you'll close your session.

There are more commands to try as you get more comfortable with the command line. Some of our other favourites are `du`, which is a way to find out how much memory is being used (`du -h` makes it human readable — as with other commands). For those of you on OS X, `top` provides an overview of what processes are running (`mem` on Windows) and `touch FILENAME` can create a basic text file on both systems

By this point, we hope you have a good, basic understanding of how to move around using the command line, move basic files, and make minor edits here and there. This beginner-level lesson is designed to give you some basic fluency and confidence. In the future, you may want to get involved with scripting.

Have fun! Before you know it, you may find yourself liking the convenience and precision of the command line - for certain applications, at least - far more than the bulkier GUI that your system came with.

Reference Guide

For your convenience, here are the commands that you have learned in this lesson:

Command	What it does
pwd	Prints the 'present working directory,' letting you know where you are.
ls	Lists the files in the current directory
`man *`	Lists the manual for the command, substituted for the *
`cd *`	Changes the current directory to *
`mkdir *`	Makes a directory named *
`open` or `explorer`	On OS X, `open` followed by a file opens it; in Windows, the command `explorer` followed by a file name does the same thing.
`cat *`	`cat` is a versatile command. It will read a file to you if you substitute a file for *, but can also be used to combine files.
`head *`	Displays the first ten lines of *
`tail *`	Displays the last ten lines of *
`mv`	Moves a file
`cp`	Copies a file

`rm`	Deletes a file
`vim`	Opens up the `vim` document editor.

About the Authors

Ian Milligan is an assistant professor of history at the University of Waterloo. James Baker is a lecturer of digital history at the University of Sussex.

23. Counting and mining research data with Unix

James Baker and Ian Milligan – 2014

Introduction

This lesson will look at how research data, when organised in a clear and predictable manner, can be counted and mined using the Unix shell. The lesson builds on the lessons "Preserving Your Research Data: Documenting and Structuring Data"[206] and "Introduction to the Bash Command Line".[207] Depending on your confidence with the Unix shell, it can also be used as a standalone lesson or refresher.

Having accumulated research data for one project, a historian might ask different questions of that same data when returning to it during a subsequent project. If this data is spread across multiple files - a series of tabulated data, a set of transcribed text, a collection of images - it can be counted and mined using simple Unix commands.

The Unix shell gives you access to a range of powerful commands that can transform how you count and mine research data. This lesson will introduce you to a series of commands that use counting and mining of tabulated data, though they only scratch the surface of what the Unix shell can do. By learning just a few simple commands you will be able to undertake tasks that are impossible in Libre Office Calc, Microsoft Excel, or other similar spreadsheet programs. These commands can be easily extended for use with non-tabulated data.

This lesson will also demonstrate that the options for manipulating, counting and mining data available to you will often depend on the amount of metadata, or descriptive text, contained in the filenames of the data you are using as much as the range of Unix commands you have learnt to use. Thus, even if it is not a prerequisite of working with the Unix shell, taking the time to structure your research data and filenaming conventions in a consistent and predictable manner is certainly a significant step towards getting the most out of Unix commands and being able to count and mine your research data. For the value of taking the time to make your data

[206] James Baker, 'Preserving Your Research Data', *The Programming Historian* (2014).

[207] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historian* (2014).

consistent and predictable beyond matters of preservation, see "Preserving Your Research Data".[208]

Software and setup

Windows users will need to install Git Bash. This can be installed by downloading the most recent installer at the git for windows webpage.[209] Instructions for installation are available at Open Hatch.[210]

OS X and Linux users will need to use their terminal shells to follow this lesson, as discussed in "Introduction to the Bash Command Line."[211]

This lesson was written using Git Bash 1.9.0 and the Windows 7 operating system. Equivalent file paths for OS X/Linux have been included where possible. Nonetheless, as commands and flags can change slightly between operating systems OS X/Linux users are referred to Deborah S. Ray and Eric J. Ray, "*Unix and Linux: Visual Quickstart Guide*", 4th edition (2009)[212] which covers interoperability in greater detail.

The files used in this lesson are available on "Figshare".[213] The data contains the metadata for journal articles categorised under 'History' in the British Library ESTAR database. The data is shared under a CC0 copyright waiver.

Download the required files, save them to your computer, and unzip them. If you do not have default software installed to interact with .zip files, we recommend 7-zip for this purpose.[214] On Windows, we recommend unzipping the folder provided to your c: drive so the files are at `c:\proghist\`. However, any location will work fine, but you may have to adjust your commands as you are following along with this lesson if you use a different location. On OS X or Linux, we similarly recommend unzipping them to your user directory, so that they appear at `/user/USERNAME/proghist/`. In both cases, this means that when you open up a new terminal window, you can just type `cd proghist` to move to the correct directory.

[208] James Baker, 'Preserving Your Research Data', *The Programming Historian* (2014).

[209] 'Git for Windows': https://git-for-windows.github.io/

[210] 'Install Git Bash', *Open Hatch*: https://openhatch.org/missions/windows-setup/install-git-bash

[211] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historian* (2014).

[212] Deborah S. Ray and Eric J. Ray, 'Unix and Linux' (4th Edition), Peachpit Press (2009).

[213] 'Data for 'Counting and mining research data with Unix', The Programming Historian' *Figshare*: https://figshare.com/articles/Data_for_Counting_and_mining_research_data_with_Unix_The_Programming_Historian_/1172094

[214] '7-Zip': http://www.7-zip.org/

Counting files

You will begin this lesson by counting the contents of files using the Unix shell. The Unix shell can be used to quickly generate counts from across files, something that is tricky to achieve using the graphical user interfaces (GUI) of standard office suites.

In Unix the `wc` command is used to count the contents of a file or of a series of files.

Open the Unix shell and navigate to the directory that contains our data, the `data` subdirectory of the `proghist` directory. Remember, if at any time you are not sure where you are in your directory structure, type `pwd` and use the `cd` command to move to where you need to be. The directory structure here is slightly different between OS X/Linux and Windows: on the former, the directory is in a format such as `~/users/USERNAME/proghist/data` and on Windows in a format such as `c:\proghist\data`.

Type `ls` and then hit enter. This prints, or displays, a list that includes two files and a subdirectory.

The files in this directory are the dataset `2014-01_JA.csv` that contains journal article metadata and a file containing documentation about `2014-01_JA.csv` called `2014-01_JA.txt`.

The subdirectory is named `derived_data`. It contains four .tsv files[215] derived from `2014-01_JA.csv`. Each of these includes all data where a keyword such as `africa` or `america` appears in the 'Title' field of `2014-01_JA.csv`. The `derived_data` directory also includes a subdirectory called `results`.

Note: *CSV* files[216] are those in which the units of data (or cells) are separated by commas (comma-separated-values) and TSV files are those in which they are separated by tabs. Both can be read in simple text editors or in spreadsheet programs such as Libre Office Calc or Microsoft Excel.

Before you begin working with these files, you should move into the directory in which they are stored. Navigate to `c:\proghist\data\derived_data` on Windows or `~/users/USERNAME/proghist/data/derived_data` on OS X.

Now that you are here you can count the contents of the files.

The Unix command for counting is `wc`. Type `wc -w 2014-01-31_JA_africa.tsv` and hit enter. The flag `-w` combined with `wc` instructs

[215] 'Tab-separated values', *Wikipedia*: https://en.wikipedia.org/wiki/Tab-separated_values

[216] 'Comma-separated values', *Wikipedia*: https://en.wikipedia.org/wiki/Comma-separated_values

the computer to print a word count, and the name of the file that has been counted, into the shell.

As was seen in "Introduction to the Bash Command Line",[217] flags such as `-w` are an essential part of getting the most out of the Unix shell as they give you better control over commands.

If your research is more concerned with the number of entries (or lines) than the number of words, you can use the line count flag. Type `wc -l 2014-01-31_JA_africa.tsv` and hit enter. Combined with `wc` the flag `-l` prints a line count and the name of the file that has been counted.

Finally, type `wc -c 2014-01-31_JA_africa.tsv` and hit enter. This uses the flag `-c` in combination with the command `wc` to print a character count for `2014-01-31_JA_africa.tsv`.

Note: OS X and Linux users should replace the *`-c`* flag with *`-m`*.

With these three flags, the most obvious thing historians can use `wc` for is to quickly compare the shape of sources in digital format - for example word counts per page of a book, the distribution of characters per page across a collection of newspapers, the average line lengths used by poets. You can also use `wc` with a combination of wildcards and flags to build more complex queries. Type `wc -l 2014-01-31_JA_a*.tsv` and hit enter. This prints the line counts for `2014-01-31_JA_africa.tsv` and `2014-01-31_JA_america.tsv`, offering a simple means of comparing these two sets of research data. Of course, it may be faster to compare the line count for the two documents in Libre Office Calc, Microsoft Excel, or a similar spreadsheet program. But when wishing to compare the line count for tens, hundreds, or thousands of documents, the Unix shell has a clear speed advantage.

Moreover, as our datasets increase in size you can use the Unix shell to do more than copy these line counts by hand, by the use of print screen, or by copy and paste methods. Using the `>` redirect operator you can export your query results to a new file. Type `wc -l 2014-01-31_JA_a*.tsv > results/2014-01-31_JA_a_wc.txt` and hit enter. This runs the same query as before, but rather than print the results within the Unix shell it saves the results as `2014-01-31_JA_a_wc.txt`. By prefacing this with `results/` it moves the .txt file to the `results` sub-directory. To check this, navigate to the `results` subdirectory, hit enter, type `ls`, and hit enter again to see this file listed within `c:\proghist\data\derived_data\results` on Windows or `/users/USERNAME/proghist/data/derived_data/results` on OS X/Linux.

[217] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historian* (2014).

Mining files

The Unix shell can do much more than count the words, characters, and lines within a file. The `grep` command (meaning 'global regular expression print') is used to search across multiple files for specific strings of characters. It is able to do so much faster than the graphical search interface offered by most operating systems or office suites. And combined with the `>` operator, the `grep` command becomes a powerful research tool can be used to mine your data for characteristics or word clusters that appear across multiple files and then export that data to a new file. The only limitations here are your imagination, the shape of your data, and - when working with thousands or millions of files - the processing power at your disposal.

To begin using `grep`, first navigate to the `derived_data` directory (`cd ..`). Here type `grep 1999 *.tsv` and hit enter. This query looks across all files in the directory that fit the given criteria (the .tsv files) for instances of the string, or character cluster, '1999'. It then prints them within the shell.

Note: there is a large amount of data to print, so if you get bored hit *`ctrl+c`* to cancel the action. Ctrl+c is used to cancel any process in the Unix shell.

Press the up arrow once in order to cycle back to your most recent action. Amend `grep 1999 *.tsv` to `grep -c 1999 *.tsv` and hit enter. The shell now prints the number of times the string 1999 appeared in each .tsv file. Cycle to the previous line again and amend this to `grep -c 1999 2014-01-31_JA_*.tsv > results/2014-01-31_JA_1999.txt` and hit enter. This query looks for instances of the string '1999' across all documents that fit the criteria and saves them as `2014-01-31_JA_1999.txt` in the `results` subdirectory.

Strings need not be numbers. `grep -c revolution 2014-01-31_JA_america.tsv 2014-02-02_JA_britain.tsv`, for example, counts the instances of the string `revolution` within the defined files and prints those counts to the shell. Run this and then amend it to `grep -ci revolution 2014-01-31_JA_america.tsv 2014-02-02_JA_britain.tsv`. This repeats the query, but prints a case insensitive count (including instances of both `revolution` and `Revolution`). Note how the count has increased nearly 30 fold for those journal article titles that contain the keyword 'revolution'. As before, cycling back and adding `> results/`, followed by a filename (ideally in .txt format), will save the results to a data file.

You can also use `grep` to create subsets of tabulated data. Type `grep -i revolution 2014-01-31_JA_america.tsv 2014-02-02_JA_britain.tsv > YEAR-MONTH-DAY_JA_america_britain_i_revolution.tsv` (where `YEAR-MONTH-DAY` is the date you are completing this lesson) and hit enter. This command looks in both of the defined files and exports any lines containing `revolution` (without regard to case) to the specified .tsv file.

The data has not been saved to to the `results` directory because it isn't strictly a result; it is derived data. Depending on your research project it may be easier to save this to another subdirectory. For now have a look at this file to verify its contents and when you are happy, delete it using the `rm` command. *Note: the `rm` common is very powerful and should be used with caution. Please refer to "Introduction to the Bash Command Line" for instructions on how to use this command correctly.*[218]

Finally, you can use another flag, `-v`, to exclude data elements when using the `grep` command. Type `grep -iv revolution 2014*_JA_a*.tsv > 2014_JA_iv_revolution.csv` and hit enter. This query looks in the defined files (three in total) and exports all lines that do not contain `revolution` or `Revolution` to `c:\proghist\data\derived_data\2014_JA_iv_revolution.csv`.

Note that you have transformed the data from one format to another - from .tsv to .csv. Often there is a loss of data structure when undertaking such transformations. To observe this for yourself, run `grep -iv revolution 2014*_JA_a*.tsv > 2014_JA_iv_revolution.tsv` and open both the .csv and .tsv files in Libre Office Calc, Microsoft Excel, or a similar spreadsheet program. Note the differences in column delineation between the two files.

Summary

Within the Unix shell you can now:

use the `wc` command with the flags `-w` and `-l` to count the words and lines in a file or a series of files.

use the redirector and structure `> subdirectory/filename` to save results into a subdirectory.

use the `grep` command to search for instances of a string.

use with `grep` the `-c` flag to count instances of a string, the `-i` flag to return a case insensitive search for a string, and the `-v` flag to exclude a string from the results.

combine these commands and flags to build complex queries in a way that suggests the potential for using the Unix shell to count and mine your research data and research projects.

Conclusion

In this lesson you have learnt to undertake some basic file counting, to query across research data for common strings, and to save results and derived data. Though this lesson is restricted to using the Unix shell to count and mine tabulated data, the processes can be easily extended to free text. For this we recommend two guides written by William Turkel:

[218] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historian* (2014).

William Turkel, 'Basic Text Analysis with Command Line Tools in Linux' (15 June 2013).[219]

William Turkel, 'Pattern Matching and Permuted Term Indexing with Command Line Tools in Linux' (20 June 2013).[220]

As these recommendations suggest, the present lesson only scratches the surface of what the Unix shell environment is capable of. It is hoped, however, that this lesson has provided a taster sufficient to prompt further investigation and productive play.

For many historians, the full potential of these tools may only emerge upon embedding these skills into a real research project. Once your research grows, and, with it, your research data, being able to manipulate, count and mine thousands of files will be extremely useful. For if you choose to build on this lesson and investigate the Unix shell further you will find that even a large collection of files which do not contain any alpha-numeric data elements, such as image files, can be easily sorted, selected and queried in the Unix shell.

About the Authors

Ian Milligan is an assistant professor of history at the University of Waterloo. James Baker is a lecturer of digital history at the University of Sussex.

[219] William J. Turkel, 'Basic Text Analysis with Command Line Tools in Linux', *William J. Turkel* (2013): http://williamjturkel.net/2013/06/15/basic-text-analysis-with-command-line-tools-in-linux/

[220] William J. Turkel, 'Pattern Matching and Permuted Term Indexing with Command Line Tools in Linux', *William J. Turkel* (2013): http://williamjturkel.net/2013/06/20/pattern-matching-and-permuted-term-indexing-with-command-line-tools-in-linux/

24. Cleaning Data with OpenRefine

Seth van Hooland, Ruben Verborgh, and Max De Wilde – 2013

Lesson goals

Don't take your data at face value. That is the key message of this tutorial which focuses on how scholars can diagnose and act upon the accuracy of data. In this lesson, you will learn the principles and practice of data cleaning, as well as how *OpenRefine*[221] can be used to perform four essential tasks that will help you to clean your data:

Remove duplicate records
Separate multiple values contained in the same field
Analyse the distribution of values throughout a data set
Group together different representations of the same reality

These steps are illustrated with the help of a series of exercises based on a collection of metadata from the Powerhouse museum,[222] demonstrating how (semi-)automated methods can help you correct the errors in your data.

Why should historians care about data quality?

Duplicate records, empty values and inconsistent formats are phenomena we should be prepared to deal with when using historical data sets. This lesson will teach you how to discover inconsistencies in data contained within a spreadsheet or a database. As we increasingly share, aggregate and reuse data on the web, historians will need to respond to data quality issues which inevitably pop up. Using a program called *OpenRefine*, you will be able to easily identify systematic errors such as blank cells, duplicates, spelling inconsistencies, etc. *OpenRefine* not only allows you to quickly diagnose the accuracy of your data, but also to act upon certain errors in an automated manner.

Description of the tool: OpenRefine

In the past, historians had to rely on information technology specialists to diagnose data quality and to run cleaning tasks. This required custom computer programs when working with sizeable data sets. Luckily, the advent of Interactive Data Transformation tools (IDTs) now allows for rapid and inexpensive operations on large amounts of data, even by professionals lacking in-depth technical skills.

[221] 'Open Refine': http://openrefine.org/

[222] 'Museum of Applied Arts and Sciences': https://maas.museum/

IDTs resemble the desktop spreadsheet software we are all familiar with, and they share some common functionalities. You can for example use an application such as Microsoft Excel to sort your data based on numerical, alphabetical and custom-developed filters, which allows you to detect errors more easily. Setting up these filters in a spreadsheet can be cumbersome, as they are a secondary functionality. On a more general level, we could say that spreadsheets are designed to work on individual rows and cells, whereas IDTs operate on large ranges of data at once. These 'spreadsheets on steroids' offer an integrated and user-friendly interface through which end users can detect and correct errors.

Several general purpose tools for interactive data transformation have been developed in recent years, such as *Potter's Wheel ABC*[223] and *Wrangler*.[224] Here we want to focus specifically on *OpenRefine*[225] (formerly Freebase Gridworks and Google Refine), as in the opinion of the authors, it is the most user-friendly tool to efficiently process and clean large amounts of data in a browser-based interface.

On top of data profiling[226] and cleaning operations, *OpenRefine* extensions allow users to identify concepts in unstructured text, a process referred to as named-entity recognition (NER)[227], and can also reconcile their own data with existing knowledge bases. By doing so, *OpenRefine* can be a practical tool to link data with concepts and authorities which have already been declared on the Web by parties such as Library of Congress[228] or OCLC.[229] Data cleaning is a prerequisite to these steps; the success rate of NER and a fruitful matching process between your data and external authorities depends on your ability to make your data as coherent as possible.

Description of the exercise: Powerhouse Museum

The Powerhouse Museum in Sydney provides a freely available metadata export of its collection on its website.[230] The museum is one of the largest science and technology museums worldwide, providing access to almost 90,000 objects, ranging from steam engines to fine glassware and from haute couture to computer chips.

[223] 'Potter's Wheel A-B-C: An Interactive Tool for Data Analysis, Cleansing, and Transformation': http://control.cs.berkeley.edu/abc/

[224] Sean Kandel, Andreaas Paepcke, Joseph Hellerstein, Jeffrey Heer, 'Wrangler: Interactive Visual Specification of Data Transformation Scripts': http://vis.stanford.edu/papers/wrangler/

[225] 'Open Refine': http://openrefine.org/

[226] 'Data profiling', *Wikipedia*: https://en.wikipedia.org/wiki/Data_profiling

[227] 'Named-entity recognition', *Wikipedia*: https://en.wikipedia.org/wiki/Named-entity_recognition

[228] 'Library of Congress': https://www.loc.gov/

[229] 'OCLC': http://www.oclc.org/home.en.html

[230] 'API, Data Access and 3D Scans', *Powerhouse Museum Collection Search*: http://www.powerhousemuseum.com/collection/database/download.php

The Powerhouse has been very actively disclosing its collection online and making most of its data freely available. From the museum website, a tab-separated text file under the name *phm-collection.tsv* can be downloaded, which you can open as a spreadsheet. The unzipped file (58MB) contains basic metadata (17 fields) for 75,823 objects, released under a Creative Commons Attribution Share Alike (CCASA) license. In this tutorial we will be using a copy of the data that we have archived for you to download (in a moment). This ensures that if the Powerhouse Museum updates the data, you will still be able to follow along with the Lesson.

Throughout the data profiling and cleaning process, the case study will specifically focus on the `Categories` field, which is populated with terms from the Powerhouse museum Object Names Thesaurus (PONT). PONT recognizes Australian usage and spelling, and reflects in a very direct manner the strengths of the collection. In the collection you will find better representations of social history and decorative arts, and comparably few object names relating to fine arts and natural history.

The terms in the Categories field comprise what we call a Controlled vocabulary.[231] A controlled vocabulary consists of keywords describing the content of a collection using a limited number of terms, and is often a key entry point into data sets used by historians in libraries, archives and museums. That is why we will give particular attention to the 'Categories' field. Once the data has been cleaned, it should be possible to reuse the terms in the controlled vocabulary to find additional information about the terms elsewhere online, which is known as creating Linked Data.[232]

Getting started: installing OpenRefine and importing data

Download OpenRefine and follow the installation instructions.[233] OpenRefine works on all platforms: Windows, Mac, and Linux. *OpenRefine* will open in your browser, but it is important to realise that the application is run locally and that your data won't be stored online. The data files are available on our FreeYourMetadata website,[234] which will be used throughout this tutorial. Please download the *phm-collection.tsv* file before continuing (also archived on the Programming Historian site: http://programminghistorian.org/images/phm-collection.tsv).

On the *OpenRefine* start page, create a new project using the downloaded data file and click **Next**. By default, the first line will be correctly parsed as the name of a column, but you need to unselect the 'Quotation marks are used to enclose cells containing column separators' checkbox, since the quotes inside the file do not have any meaning to *OpenRefine*. Now click on '**Create project**'. If all goes well, you will see 75,814 rows. Alternatively,

[231] 'Controlled vocabulary', *Wikipedia*: https://en.wikipedia.org/wiki/Controlled_vocabulary

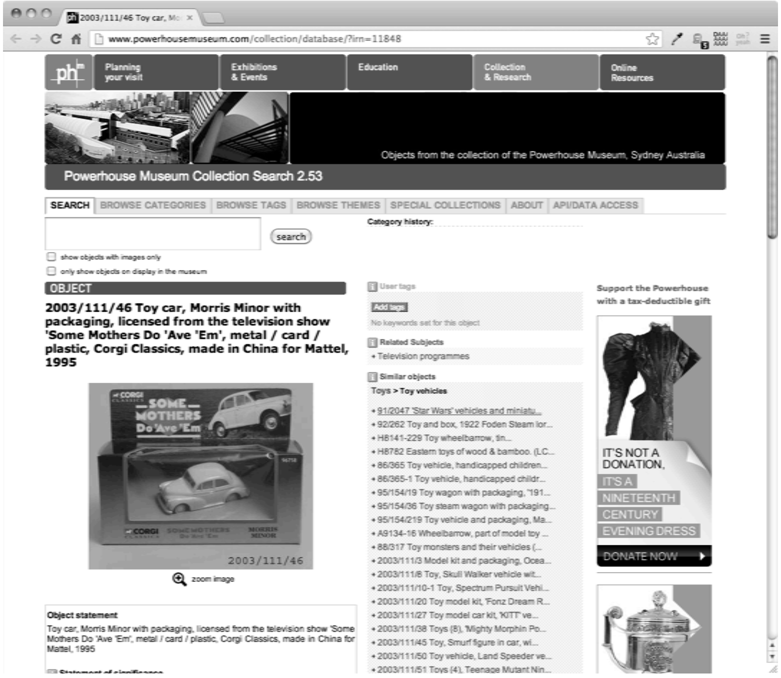
[232] 'Linked data', *Wikipedia*: https://en.wikipedia.org/wiki/Linked_data

[233] Download Open Refine': http://openrefine.org/#download_openrefine

[234] 'Index of /powerhouse-museum/': http://data.freeyourmetadata.org/powerhouse-museum/

you can download the initial OpenRefine project directly (http://data.freeyourmetadata.org/powerhouse-museum/phm-collection.google-refine.tar.gz).

The Powerhouse museum data set consists of detailed metadata on all the collection objects, including title, description, several categories the item belongs to, provenance information, and a persistent link to the object on the museum website. To get an idea of what object the metadata corresponds to, simply click the persistent link and the website will open.



□ show objects with images only

□ only show objects on display in the museum

2003/111/46 Toy car, Morris Minor with packaging, licensed from the television show 'Some Mothers Do 'Ave 'Em', metal / card / plastic, Corgi Classics, made in China for Mattel, 1995

No keywords set for this object

- Television programmes

Toys > Toy vehicles

- 91/2047 'Star Wars' vehicles and miniatu...
- 92/262 Toy and box, 1922 Foden Steam lor...
- H8141-229 Toy wheelbarrow, tin...
- H8782 Eastern toys of wood & bamboo. (LC...
- 86/365 Toy vehicle, handicapped children...
- 86/365-1 Toy vehicle, handicapped childr...
- 95/154/19 Toy wagon with packaging, "191...
- 95/154/36 Toy steam wagon with packaging...
- 95/154/219 Toy vehicle and packaging, Ma...
- A9134-16 Wheelbarrow, part of model toy ...
- 88/317 Toy monsters and their vehicles (...
- 2003/111/3 Model kit and packaging, Ocea...
- 2003/111/8 Toy, Skull Walker vehicle wit...
- 2003/111/10-1 Toy, Spectrum Pursuit Vehi...
- 2003/111/20 Toy model kit, 'Fonz Dream R...
- 2003/111/27 Toy model car kit, 'KITT' ve...
- 2003/111/38 Toys (8), 'Mighty Morphin Po...
- 2003/111/45 Toy, Smurf figure in car, wi...
- 2003/111/50 Toy vehicle, Land Speeder ve...
- 2003/111/51 Toys (4), Teenage Mutant Nin...

Object statement

Toy car, Morris Minor with packaging, licensed from the television show 'Some Mothers Do 'Ave 'Em', metal / card / plastic, Corgi Classics, made in China for Mattel, 1995

Screenshot of a Sample Object on the Powerhouse Museum Website

Get to know your data

The first thing to do is to look around and get to know your data. You can inspect the different data values by displaying them in `facets`. You could consider a facet[235] like a lense through which you view a specific subset of the data, based on a criterion of your choice. Click the triangle in front of the column name, select Facet, and create a facet. For instance, try a `Text` facet or a `Numeric` facet, depending on the nature of the values contained in the fields (numeric values are in green). Be warned, however, that text facets are best used on fields with redundant values (Categories for instance); if you run into a 'too many to display' error, you can choose to raise the choice count limit above the 2,000 default, but too high a limit can slow down the application (5,000 is usually a safe choice). Numeric facets do not have this restriction. For more options, select Customized facets: facet by blank, for instance, comes handy to find out how many values were filled in for each field. We'll explore these further in the following exercises.

[235] 'faceted search': http://en.wikipedia.org/wiki/Faceted_search

Remove blank rows

One thing you notice when creating a numeric facet for the Record ID column, is that three rows are empty. You can find them by unselecting the Numeric checkbox, leaving only Non-numeric values. Actually, these values are not really blank but contain a single whitespace character, which can be seen by moving your cursor to where the value should have been and clicking the '**edit**' button that appears. To remove these rows, click the triangle in front of the first column called '**All**', select '**Edit rows**', and then '**Remove all matching rows**'. Close the numeric facet to see the remaining 75,811 rows.

Removing duplicates

A second step is to detect and remove duplicates. These can be spotted by sorting them by a unique value, such as the Record ID (in this case we are assuming the Record ID should in fact be unique for each entry). The operation can be performed by clicking the triangle left of Record ID, then choosing '**Sort**'… and selecting the '**numbers**' bullet. In *OpenRefine*, sorting is only a visual aid, unless you make the reordering permanent. To do this, click the Sort menu that has just appeared at the top and choose '**Reorder rows permanently**'. If you forget to do this, you will get unpredictable results later in this tutorial.

Identical rows are now adjacent to each other. Next, blank the Record ID of rows that have the same Record ID as the row above them, marking them duplicates. To do this, click on the Record ID triangle, choose **Edit cells** > **Blank down**. The status message tells you that 84 columns were affected (if you forgot to reorder rows permanently, you will get only 19; if so, undo the blank down operation in the 'Undo/Redo' tab and go back to the previous paragraph to make sure that rows are reordered and not simply sorted). Eliminate those rows by creating a facet on '**blank cells**' in the Record ID column ('**Facet**' > '**Customized facets**' > '**Facet by blank**'), selecting the 84 blank rows by clicking on '**true**', and removing them using the '**All**' triangle ('**Edit rows**' > '**Remove all matching rows**'). Upon closing the facet, you see 75,727 unique rows.

Be aware that special caution is needed when eliminating duplicates. In the above mentioned step, we assume the dataset has a field with unique values, indicating that the entire row represents a duplicate. This is not necessarily the case, and great caution should be taken to manually verify wether the entire row represents a duplicate or not.

Atomization

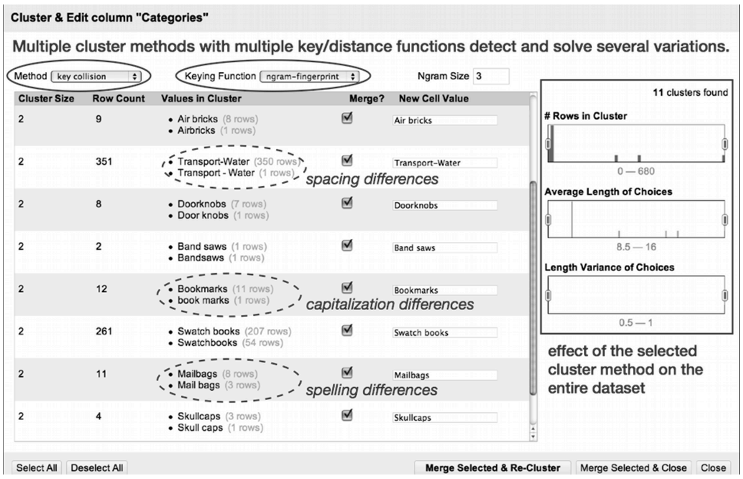
Once the duplicate records have been removed, we can have a closer look at the *Categories* field. On average each object has been attributed 2.25 categories. These categories are contained within the same field, separated by a pipe character '|'. Record 9, for instance, contains three: 'Mineral samples|Specimens|Mineral Samples-Geological'. In order to analyze in detail the use of the keywords, the values of the Categories field need to be split up into individual cells on the basis of the pipe character , expanding the 75,727 records into 170,167 rows. Choose '**Edit cells**', '**Split multi-valued cells**', entering '|' as the value separator. OpenRefine informs you that you now have 170,167 rows.

It is important to fully understand the rows/records paradigm. Make the Record ID column visible to see what is going on. You can switch between 'rows' and 'records' view by clicking on the so-labelled links just above the column headers. In the 'rows view', each row represents a couple of Record IDs and a single Category, enabling manipulation of each one individually. The 'records view' has an entry for each Record ID, which can have different categories on different rows (grouped together in grey or white), but each record is manipulated as a whole. Concretely, there now are 170,167 category assignments (rows), spread over 75,736 collection items (records). You maybe noticed that we are 9 records up from the original 75,727, but don't worry about that for the time being, we will come back to this small difference later.

Facetting and clustering

Once the content of a field has been properly atomized, filters, facets, and clusters can be applied to give a quick and straightforward overview of classic metadata issues. By applying the customized facet 'Facet by blank', one can immediately identify the 461 records that do not have a category, representing 0.6% of the collection. Applying a text facet to the Categories field allows an overview of the 4,934 different categories used in the collection (the default limit being 2,000, you can click '**Set choice count limit**' to raise it to 5,000). The headings can be sorted alphabetically or by frequency ('count'), giving a list of the most used terms to index the collection. The top three headings are 'Numismatics' (8,041), 'Ceramics' (7,390) and 'Clothing and dress' (7,279).

After the application of a facet, *OpenRefine* proposes to cluster facet choices together based on various similarity methods. As the Figure below illustrates, the clustering allows you to solve issues regarding case inconsistencies, incoherent use of either the singular or plural form, and simple spelling mistakes. *OpenRefine* presents the related values and proposes a merge into the most recurrent value. Select values you wish to cluster by selecting their boxes individually or by clicking '**Select all**' at the bottom, then chose '**Merge Selected and Re-Cluster**'.



Cluster & Edit column "Categories"

Multiple cluster methods with multiple key/distance functions detect and solve several variations.

Method: key collision   Keying Function: ngram-fingerprint   Ngram Size: 3

Cluster Size	Row Count	Values in Cluster	Merge?	New Cell Value
2	9	• Air bricks (8 rows) • Airbricks (1 rows)	☑	Air bricks
2	351	• Transport-Water (350 rows) • Transport - Water (1 rows)	☑	Transport-Water
2	8	• Doorknobs (7 rows) • Door knobs (1 rows)	☑	Doorknobs
2	2	• Band saws (1 rows) • Bandsaws (1 rows)	☑	Band saws
2	12	• Bookmarks (11 rows) • book marks (1 rows)	☑	Bookmarks
2	261	• Swatch books (207 rows) • Swatchbooks (54 rows)	☑	Swatch books
2	11	• Mailbags (8 rows) • Mail bags (3 rows)	☑	Mailbags
2	4	• Skullcaps (3 rows) • Skull caps (1 rows)	☑	Skullcaps

spacing differences
capitalization differences
spelling differences
11 clusters found
# Rows in Cluster
0 — 680
Average Length of Choices
8.5 — 16
Length Variance of Choices
0.5 — 1
effect of the selected cluster method on the entire dataset
Select All
Deselect All
Merge Selected & Re-Cluster
Merge Selected & Close
Close

Overview of some clusters

The default clustering method is not too complicated, so it does not find all clusters yet. Experiment with different methods to see what results they yield. Be careful though: some methods are too aggressive, so you might end up clustering values that do not belong together. Now that the values have been clustered individually, we can put them back together in a single cell. Click the Categories triangle and choose **Edit cells**, **Join multi-valued cells**, **OK**. Choose the pipe character (|) as a separator. The rows now look like before, with a multi-valued Categories field.

Applying ad-hoc transformations through the use of regular expressions

You may remember there was an increase in the number of records after the splitting process: nine records appeared out of nowhere. In order to find the cause of this disparity, we need to go back in time before we split the categories into separate rows. To do so, toggle the Undo/Redo tab right of the Facet/Filter tab, and you will get a history of all the actions that you performed since the project was created. Select the step just before 'Split multi-valued cells in column Categories' (if you followed our example this should be 'Remove 84 rows') then go back to the Facet/Filter tab.

The issue arose during the splitting operation on the pipe character, so there is a strong chance that whatever went wrong is linked to this character. Let's apply a filter on the Categories column by selecting '**Text filter**' in the menu. First type a single **|** in the field on the left: *OpenRefine* informs you that there are 71,064 matching records (i.e. records containing a pipe) out of a total of 75,727. Cells that do not contain a pipe can be blank ones, but also cells containing a single category with no separator, such as record 29 which only has 'Scientific instruments'.

Now enter a second **|** after the first one to get || (double pipe): you can see that 9 records are matching this pattern. These are likely the 9 records guilty of our discrepancy: when *OpenRefine* splits these up, the double pipe is interpreted as a break between two records instead of a meaningless double separator. Now how do we correct these values? Go to the menu of the 'Categories' field, and choose '**Edit cells**' > '**Transform**'.... Welcome to the custom text tranform interface, a powerful functionality of *OpenRefine* using the *OpenRefine* Expression Language (GREL).

The word 'value' in the text field represents the current value of each cell, which you can see below. We can modify this value by applying functions to it (see the GREL documentation for a full list).[236] In this case, we want to replace double pipes with a single pipe. This can be achieved by entering the following regular expression[237] (be sure not to forget the quotes):

```
value.replace('||', '|')
```

Under the 'Expression' text field, you get a preview of the modified values, with double pipes removed. Click **OK** and try again to split the categories with '**Edit cells**' > '**Split multi-valued cells**', the number of records will now stay at 75,727 (click the '**records**' link to double-check).

Another issue that can be solved with the help of GREL is the problem of records for which the same category is listed twice. Take record 41 for instance, whose categories are 'Models|Botanical specimens|Botanical Specimens|Didactic Displays|Models'. The category 'Models' appears twice without any good reason, so we want to remove this duplicate. Click the Categories triangle and choose Edit cells, Join multi-valued cells, OK. Choose the pipe character as a separator. Now the categories are listed as before. Then select '**Edit cells**' > '**Transform**', also on the categories column. Using GREL we can successively split the categories on the pipe character, look for unique categories and join them back again. To achieve this, just type the following expression:

```
value.split('|').uniques().join('|')
```

You will notice that 32,599 cells are affected, more than half the collection.

Exporting your cleaned data

Since you first loaded your data into *OpenRefine*, all cleaning operations have been performed in the software memory, leaving your original data set untouched. If you want to save the data that you have been cleaning, you need to export them by clicking on the '**Export**' menu top-right of the screen. *OpenRefine* supports a large variety of formats, such as CSV, HTML or Excel: select whatever suits you best or add your own export template by clicking 'Templating'. You can also export your project in the internal *OpenRefine* format in order to share it with others.

[236] Joe Wicentowski, 'GREL Functions': https://github.com/OpenRefine/OpenRefine/wiki/GREL-Functions

[237] 'Regular expression', *Wikipedia*: https://en.wikipedia.org/wiki/Regular_expression

Building on top of your cleaned data

Once your data has been cleaned, you can take the next step and explore other exciting features of *OpenRefine*. The user community of *OpenRefine* has developed two particularly interesting extensions which allow you to link your data to data that has already been published on the Web. The RDF Refine extension[238] transforms plaintext keywords into URLs. The NER extension[239] allows you to apply named-entity recognition (NER), which identifies keywords in flowing text and gives them a URL.

Conclusions

If you only remember on thing from this lesson, it should be this: *all data is dirty, but you can do something about it.* As we have shown here, there is already a lot you can do yourself to increase data quality significantly. First of all, you have learned how you can get a quick overview of how many empty values your dataset contains and how often a particular value (e.g. a keyword) is used throughout a collection. This lessons also demonstrated how to solve recurrent issues such as duplicates and spelling inconsistencies in an automated manner with the help of *OpenRefine*. Don't hesitate to experiment with the cleaning features, as you're performing these steps on a copy of your data set, and *OpenRefine* allows you to trace back all of your steps in the case you have made an error.

About the Authors

Seth van Hooland is an associate professor at at the Information and Communication Science department of the Université libre de Bruxelles. Ruben Verborgh is a post-doc researcher at the Multimedia Lab of the Universtiteit Gent. Max De Wilde is a PhD student at the Information and Communication Science department of the Université libre de Bruxelles.

[238] 'RDF Refine': http://refine.deri.ie/docs

[239] RubenVerborgh, 'Refine-NER-Extension': https://github.com/RubenVerborgh/Refine-NER-Extension

25. Using Gazetteers to Extract Sets of Keywords from Free-Flowing Texts

Adam Crymble – 2015

Lesson Goals

If you have a copy of a text in electronic format stored on your computer, it is relatively easy to keyword search for a single term. Often you can do this by using the built-in search features in your favourite text editor. However, scholars are increasingly needing to find instances of many terms within a text or texts. For example, a scholar may want to use a gazetteer[240] to extract all mentions of English placenames within a collection of texts so that those places can later be plotted on a map. Alternatively, they may want to extract all male given names, all pronouns, stop words,[241] or any other set of words. Using those same built-in search features to achieve this more complex goal is time consuming and clunky. This lesson will teach you how to use Python to extract a set of keywords very quickly and systematically from a set of texts.

It is expected that once you have completed this lesson, you will be able to generalise the skills to extract custom sets of keywords from any set of locally saved files.

For Whom is this Useful?

This lesson is useful for anyone who works with historical sources that are stored locally on their own computer, and that are transcribed into mutable electronic text (eg, .txt, .xml, .rtf, .md). It is particularly useful for people interested in identifying subsets of documents containing one or more of a fairly large number of keywords. This might be useful for identifying a relevant subset for closer reading, or for extracting and structuring the keywords in a format that can be used in another tool: as input for a mapping exercise, for example.

The present tutorial will show users how to extract all mentions of English and Welsh county names from a series of 6,692 mini-biographies of individuals who began their studies at the University of Oxford during the reign of James I of England (1603-1625). These records were transcribed by

[240] 'Gazetteer', *Wikipedia*: http://programminghistorian.org/lessons/extracting-keywords

[241] 'Stop words', *Wikipedia*: https://en.wikipedia.org/wiki/Stop_words

British History Online,[242] from the printed version of *Alumni Oxonienses, 1500-1714*. These biographies contain information about each graduate, which includes the date of their studies and the college(s) they attended. Often entries contain additional information when known, including date or birth and death, the name or occupation of their father, where they originated, and what they went on to do in later life. The biographies are a rich resource, providing reasonably comparable data about a large number of similar individuals (rich men who went to Oxford). The 6,692 entries have been pre-processed by the author and saved to a CSV file[243] with one entry per row.

In this tutorial, the dataset involves geographical keywords. Once extracted, these placenames could be geo-referenced to their place on the globe and then mapped using digital mapping. This might make it possible to discern which colleges attracted students from what parts of the country, or to determine if these patterns changed over time. For a practical tutorial on taking this next step, see the lesson by Fred Gibbs mentioned at the end of this lesson. Readers may also be interested in 'Georeferencing in QGIS 2.0', also available from the *Programming Historian*.[244]

This approach is not limited to geographical keywords, however. It could also be used to extract given names, prepositions, food words, or any other set of terms defined by the user. This process could therefore be useful for someone seeking to isolate individual entries containing any of these keywords, or for someone looking to calculate the frequency of their keywords within a corpus of texts. This tutorial provides pathways into textual or geospatial analyses, rather than research answers in its own right.

Data management skills are increasingly crucial for historians and textual scholars, and anyone working with particularly messy or difficult texts might consider looking into 'Cleaning Data with OpenRefine' by Seth van Hooland et al.[245] The approach outlined in this tutorial is not optimised for working with poorly transcribed texts such as text converted through Optical Character Recognition[246] if the software has a high error rate. Readers working with early modern texts with non-standardised spelling may also find this approach challenging, as the gazetteer one uses must contain exact matches of the words sought. However, with a good enough gazetteer, this approach could prove quite useful for early modernites, and

[242] 'Alumni Oxonienses 1500-1714', *British History Online*: http://www.british-history.ac.uk/alumni-oxon/1500-1714

[243] 'Comma-separated values', *Wikipedia*: https://en.wikipedia.org/wiki/Comma-separated_values

[244] Jim Clifford, Josh MacFadyen, and Daniel Macfarlane, 'Georeferencing in QGIS 2.0', *The Programming Historian* (2013).

[245] 'Seth van Hooland, Ruben Verborgh, and Max De Wilde, 'Cleaning Data with OpenRefine', *The Programming Historian* (2013).

[246] 'Optical Character Recognition', *Wikipedia*: https://en.wikipedia.org/wiki/Optical_character_recognition

may exceed what's practical with traditional keyword searching by making fuzzy searching possible.[247]

This tutorial assumes that you have already installed Python version 2 on your computer. The lesson will use the Command Line to run Python, as this is more flexible and makes it possible for use in classrooms or computer labs in which students do not have the ability to download and install interactive development environments (IDEs) like Komodo Edit. Readers who would prefer to use an IDE might like to first read Python Introduction and Installation, but this is optional.[248] The tutorial also makes some basic assumptions about your Python skills. It assumes you know what the following Python data structures are (though not knowing will not prevent the code from working should you follow all of the steps in the tutorial):

List[249]
For Loop[250]
String[251]

The lesson touches on Regular Expressions, so some readers may find it handy to have the relevant Programming Historian lessons by Doug Knox or Laura Turner O'Hara open to consult as needed.[252]

Familiarising yourself with the data

The first step of this process is to take a look at the data that we will be using in the lesson. As mentioned, the data includes biographical details of approximately 6,692 graduates who began study at the University of Oxford in the early seventeenth century.

The_Dataset_-_Alumni_Oxonienses-Jas1.csv (1.4MB)

http://programminghistorian.org/assets/The_Dataset_-_Alumni_Oxonienses-Jas1.csv

[247] 'Approximate string matching', *Wikipedia*: https://en.wikipedia.org/wiki/Approximate_string_matching

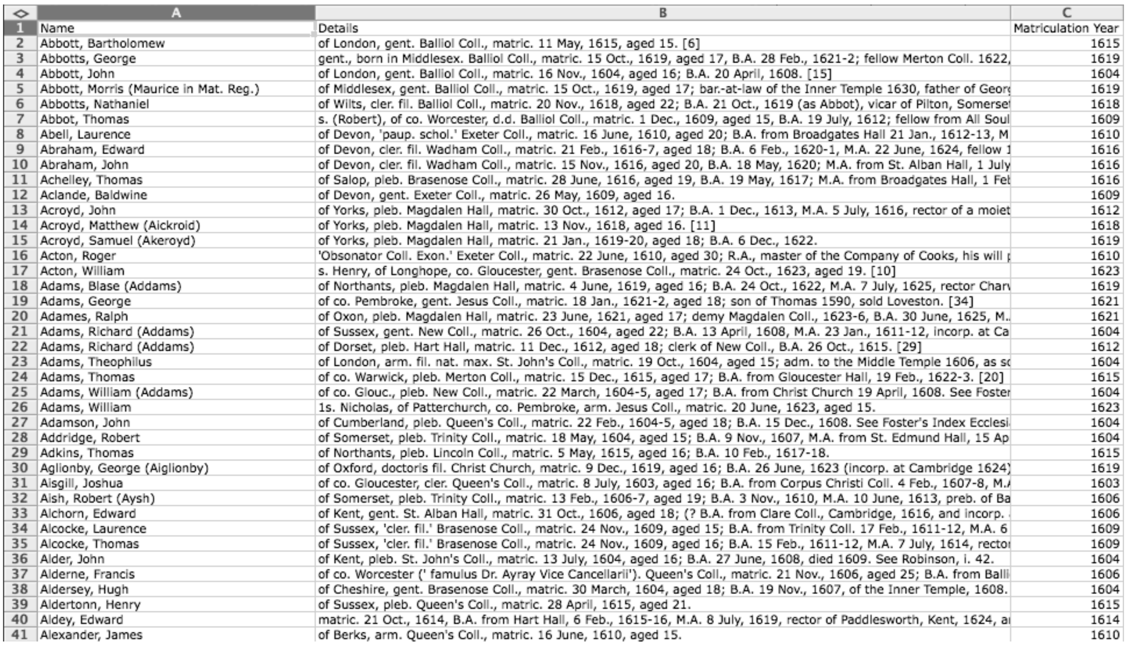
[248] William J. Turkel and Adam Crymble, 'Python Introduction and Installation' *The Programming Historian* (2012).

[249] 'Data Structures', *Python*: https://docs.python.org/2/tutorial/datastructures.html

[250] 'More Control Flow Tools', *Python*: https://docs.python.org/2/tutorial/controlflow.html

[251] 'String', *Python*: https://docs.python.org/2/library/string.html

[252] Doug Knox, 'Understanding Regular Expressions' (2013); Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).



	A	B	C
1	Name	Details	Matriculation Year
2	Abbott, Bartholomew	of London, gent. Balliol Coll., matric. 11 May, 1615, aged 15. [6]	1615
3	Abbotts, George	gent., born in Middlesex. Balliol Coll., matric. 15 Oct., 1619, aged 17, B.A. 28 Feb., 1621-2; fellow Merton Coll. 1622,	1619
4	Abbott, John	of London, gent. Balliol Coll., matric. 16 Nov., 1604, aged 16; B.A. 20 April, 1608. [15]	1604
5	Abbott, Morris (Maurice in Mat. Reg.)	of Middlesex, gent. Balliol Coll., matric. 15 Oct., 1619, aged 17; bar.-at-law of the Inner Temple 1630, father of Geor	1619
6	Abbotts, Nathaniel	of Wilts, cler. fil. Balliol Coll., matric. 20 Nov., 1618, aged 22; B.A. 21 Oct., 1619 (as Abbot), vicar of Pilton, Somerse	1618
7	Abbot, Thomas	s. (Robert), of co. Worcester, d.d. Balliol Coll., matric. 1 Dec., 1609, aged 15, B.A. 19 July, 1612; fellow from All Soul	1609
8	Abell, Laurence	of Devon, 'paup. schol.' Exeter Coll., matric. 16 June, 1610, aged 20; B.A. from Broadgates Hall 21 Jan., 1612-13, M	1610
9	Abraham, Edward	of Devon, cler. fil. Wadham Coll., matric. 21 Feb., 1616-7, aged 18; B.A. 6 Feb., 1620-1, M.A. 22 June, 1624, fellow	1616
10	Abraham, John	of Devon, cler. fil. Wadham Coll., matric. 15 Nov., 1616, aged 20, B.A. 18 May, 1620; M.A. from St. Alban Hall, 1 July	1616
11	Achelley, Thomas	of Salop, pleb. Brasenose Coll., matric. 28 June, 1616, aged 19, B.A. 19 May, 1617; M.A. from Broadgates Hall, 1 Fe	1616
12	Aclande, Baldwine	of Devon, gent. Exeter Coll., matric. 26 May, 1609, aged 16.	1609
13	Acroyd, John	of Yorks, pleb. Magdalen Hall, matric. 30 Oct., 1612, aged 17; B.A. 1 Dec., 1613, M.A. 5 July, 1616, rector of a moiet	1612
14	Acroyd, Matthew (Aickroid)	of Yorks, pleb. Magdalen Hall, matric. 13 Nov., 1618, aged 16. [11]	1618
15	Acroyd, Samuel (Akeroyd)	of Yorks, pleb. Magdalen Hall, matric. 21 Jan., 1619-20, aged 18; B.A. 6 Dec., 1622.	1619
16	Acton, Roger	'Obsonator Coll. Exon.' Exeter Coll., matric. 22 June, 1610, aged 30; R.A., master of the Company of Cooks, his will	1610
17	Acton, William	s. Henry, of Longhope, co. Gloucester, gent. Brasenose Coll., matric. 24 Oct., 1623, aged 19. [10]	1623
18	Adams, Blase (Addams)	of Northants, pleb. Magdalen Hall, matric. 4 June, 1619, aged 16; B.A. 24 Oct., 1622, M.A. 7 July, 1625, rector Char	1619
19	Adams, George	of co. Pembroke, gent. Jesus Coll., matric. 18 Jan., 1621-2, aged 18; son of Thomas 1590, sold Loveston. [34]	1621
20	Adames, Ralph	of Oxon, pleb. Magdalen Hall, matric. 23 June, 1621, aged 17; demy Magdalen Coll., 1623-6, B.A. 30 June, 1625, M..	1621
21	Adams, Richard (Addams)	of Sussex, gent. New Coll., matric. 26 Oct., 1604, aged 22; B.A. 13 April, 1608, M.A. 23 Jan., 1611-12, incorp. at Ca	1604
22	Adams, Richard (Addams)	of Dorset, pleb. Hart Hall, matric. 11 Dec., 1612, aged 18; clerk of New Coll., B.A. 26 Oct., 1615. [29]	1612
23	Adams, Theophilus	of London, arm. fil. nat. max. St. John's Coll., matric. 19 Oct., 1604, aged 15; adm. to the Middle Temple 1606, as s	1604
24	Adams, Thomas	of co. Warwick, pleb. Merton Coll., matric. 15 Dec., 1615, aged 17; B.A. from Gloucester Hall, 19 Feb., 1622-3. [20]	1615
25	Adams, William (Addams)	of co. Glouc., pleb. New Coll., matric. 22 March, 1604-5, aged 17; B.A. from Christ Church 19 April, 1608. See Foster	1604
26	Adams, William	1s. Nicholas, of Patterchurch, co. Pembroke, arm. Jesus Coll., matric. 20 June, 1623, aged 15.	1623
27	Adamson, John	of Cumberland, pleb. Queen's Coll., matric. 22 Feb., 1604-5, aged 18; B.A. 15 Dec., 1608. See Foster's Index Ecclesi	1604
28	Addridge, Robert	of Somerset, pleb. Trinity Coll., matric. 18 May, 1604, aged 15; B.A. 9 Nov., 1607, M.A. from St. Edmund Hall, 15 Ap	1604
29	Adkins, Thomas	of Northants, pleb. Lincoln Coll., matric. 5 May, 1615, aged 16; B.A. 10 Feb., 1617-18.	1615
30	Aglionby, George (Aiglionby)	of Oxford, doctoris fil. Christ Church, matric. 9 Dec., 1619, aged 16; B.A. 26 June, 1623 (incorp. at Cambridge 1624)	1619
31	Aisgill, Joshua	of co. Gloucester, cler. Queen's Coll., matric. 8 July, 1603, aged 16; B.A. from Corpus Christi Coll. 4 Feb., 1607-8, M.	1603
32	Aish, Robert (Aysh)	of Somerset, pleb. Trinity Coll., matric. 13 Feb., 1606-7, aged 19; B.A. 3 Nov., 1610, M.A. 10 June, 1613, preb. of Ba	1606
33	Alchorn, Edward	of Kent, gent. St. Alban Hall, matric. 31 Oct., 1606, aged 18; (? B.A. from Clare Coll., Cambridge, 1616, and incorp.	1606
34	Alcocke, Laurence	of Sussex, 'cler. fil.' Brasenose Coll., matric. 24 Nov., 1609, aged 15; B.A. from Trinity Coll. 17 Feb., 1611-12, M.A. 6	1609
35	Alcocke, Thomas	of Sussex, 'cler. fil.' Brasenose Coll., matric. 24 Nov., 1609, aged 16; B.A. 15 Feb., 1611-12, M.A. 7 July, 1614, recto	1609
36	Alder, John	of Kent, pleb. St. John's Coll., matric. 13 July, 1604, aged 16; B.A. 27 June, 1608, died 1609. See Robinson, i. 42.	1604
37	Alderne, Francis	of co. Worcester (' famulus Dr. Ayray Vice Cancellarii'). Queen's Coll., matric. 21 Nov., 1606, aged 25; B.A. from Balli	1606
38	Aldersey, Hugh	of Cheshire, gent. Brasenose Coll., matric. 30 March, 1604, aged 18; B.A. 19 Nov., 1607, of the Inner Temple, 1608.	1604
39	Aldertonn, Henry	of Sussex, pleb. Queen's Coll., matric. 28 April, 1615, aged 21.	1615
40	Aldey, Edward	matric. 21 Oct., 1614, B.A. from Hart Hall, 6 Feb., 1615-16, M.A. 8 July, 1619, rector of Paddlesworth, Kent, 1624, a	1614
41	Alexander, James	of Berks, arm. Queen's Coll., matric. 16 June, 1610, aged 15.	1610

Screenshot of the first forty entries in the dataset

Download the dataset and spend a couple of minutes looking at the types of information available. You should notice three columns of information. The first, 'Name', contains the name of the graduate. The second: 'Details', contains the biographical information known about that person. The final column, 'Matriculation Year', contains the year in which the person matriculated (began their studies). This final column was extracted from the details column in the pre-processing stage of this tutorial. The first two columns are as you would find them on the British History Online version of the *Alumni Oxonienses*. If you notice more than three columns then your spreadsheet programme has incorrectly set the delimiter[253] between columns. It should be set to "," (double quotes, comma). How you do this depends on your spreadsheet programme, but you should be able to find the solution online.

Most (but not all) of these bibliographic entries contain enough information to tell us what county the graduate came from. Notice that a large number of entries contain placenames that correspond to either major cities ('of London', in the first entry) or English counties ('of Middlesex' in entry 5 or 'of Wilts' - short for Wiltshire in entry 6). If you are not British you may not be familiar with these county names. You can find a list of historic counties of England on Wikipedia.[254]

Unfortunately, the information is not always available in the same format. Sometimes it's the first thing mentioned in an entry. Sometimes it's in the middle. Our challenge is to extract those counties of origin from within this messy text, and store it in a new column next to that person's entry.

[253] 'Delimiter', *Wikipedia*: https://en.wikipedia.org/wiki/Delimiter

[254] 'Historic counties of England', *Wikipedia*: https://en.wikipedia.org/wiki/Historic_counties_of_England

Build your gazetteer

In order to extract the relevant place names, we first have to decide what they are. We need a list of places, often called a gazetteer. Many of the place names mentioned in the records are shortforms, such as 'Wilts' instead of 'Wiltshire', or 'Salop' instead of 'Shropshire'. Getting all of these variations may be tricky. For a start, let's build a basic gazetteer of English counties.

Make a new directory (folder) on your computer where you will save all of your work. Create a text file called `gazetteer.txt` and using the entries listed on the Wikipedia page listed above, add each county to a new line on the text file. It should look something like this:

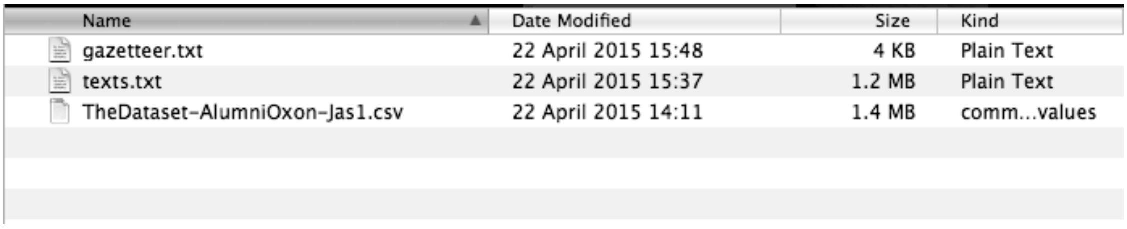
```
Bedfordshire
Berkshire
Buckinghamshire
Cambridgeshire
Cheshire
Cornwall
Cumberland
Derbyshire
Devon
Dorset
Durham
Essex
Gloucestershire
Hampshire
Herefordshire
Hertfordshire
Huntingdonshire
Kent
Lancashire
Leicestershire
Lincolnshire
Middlesex
Norfolk
Northamptonshire
Northumberland
Nottinghamshire
Oxfordshire
Rutland
Shropshire
Somerset
Staffordshire
Suffolk
Surrey
Sussex
Warwickshire
Westmorland
Wiltshire
Worcestershire
Yorkshire
```

Make sure that there are no blank lines in the gazetteer file. If there are, your program will think all spaces are a matching keyword. Some text editing programs (particularly in Linux) will want to add a blank line at the end of your file. If this is the case, try another text editor. It's best to use software that puts you in control.

If you ever need to add to this set of keywords, you can open this file in your text editor and add new words, each on their own line. Komodo Edit is a good text editor for this task, especially if you have set it up to run with Python, but you can also use any plain text editor as long as it is *not* a word processor such as Microsoft Word or Open Office.[255] Word processors are inappropriate for writing code because of how they stylise apostrophes and quotes, causing havoc for your code.

Loading your texts

The next step is to put the texts that you want to search into another text file, with one entry per line. The easiest way to do that is to open the spreadsheet and select all of the second (details) column, then paste the contents into a .txt file. Call the file `texts.txt` and save it to the same directory as your `gazetteer.txt` file. Your directory should look something like this:



Name	Date Modified	Size	Kind
gazetteer.txt	22 April 2015 15:48	4 KB	Plain Text
texts.txt	22 April 2015 15:37	1.2 MB	Plain Text
TheDataset-AlumniOxon-Jas1.csv	22 April 2015 14:11	1.4 MB	comm...values

Contents of your working directory

The reason we do this is to keep the original data (the .CSV file) away from the Python program we are about to write, on the off-chance that we accidentally change something without noticing. In my opinion, this approach also makes for easier to read code, which is important when learning. It is not strictly necessary to use a .txt file for this step, as Python does have ways of opening and reading CSV files. At the end of this lesson we will look at how to use the CSV reading and writing features in Python, but this is an optional advanced step.

Write your Python program

The last step is to write a program that will check each of the texts for each of the keywords in the gazetteer, and then to provide an output that will tell us which entries in our spreadsheet contain which of those words. There are lots of ways that this could be achieved. When planning to write a program, it is always a good idea to devise an algorithm. An algorithm is a set of human-readable steps that will solve the problem. It's a list of what you are going to do, which you then convert into the appropriate

[255] 'Word processor', *Wikipedia*: https://en.wikipedia.org/wiki/Word_processor

programmatic instructions. The approach we will take here uses the following algorithm:

Load the list of keywords that you've created in `gazetteer.txt` and save them each to a Python list

Load the texts from `texts.txt` and save each one to another Python list

Then for each biographical entry, remove the unwanted punctuation (periods, commas, etc)

Then check for the presence of one or more of the keywords from your list. If it finds a match, store it while it checks for other matches. If it finds no match, move on to the next entry

Finally, output the results in a format that can be easily transferred back to the CSV file.

Step 1: Load the Keywords

Using your text editor, create a new empty file, and add the following lines:

```
#Import the keywords
f = open('gazetteer.txt', 'r')
allKeywords = f.read().lower().split("\n")
f.close()

print allKeywords
print len(allKeywords)
```

The first line is a comment for our own benefit, to tells us (the human) what the code does. All Python lines beginning with a # are a comment. When the code runs it will ignore the comments. They are for human use only. A well commented piece of code is easier to return to later because you will have the means of decyphering your earlier creation.

The second line opens the `gazetteer.txt` file, and reads it, which is signified by the 'r' (as opposed to 'w' for write, or 'a' for append). That means we will not be changing the contents of the file. Only reading it.

The third line reads everything in that file, converts it to `lower()` case, and splits the contents into a Python list, using the newline character as the delimiter.[256] Effectively that means each time the program comes across a new line, it stores it as a new entry. We then save that Python list containing the 39 counties into a variable that we have called `allKeywords`.

The fourth line closes the open text file. The fifth line prints out the results, and the sixth line tells us how many results were found.

Save this file as `extractKeywords.py`, again to the same folder as the other files, and then run it with Python. To do this from the command line, first you need to launch your command line terminal. On Mac OS X, this is found in the `Applications` folder and is called `Terminal`. On Windows it is

[256] 'New line Python', *Stack Overflow*: http://stackoverflow.com/questions/11497376/new-line-python

called `Command Prompt`. Instructions here are given for Mac Terminal users, but Windows users should be able to follow along.

Once the Terminal window is open, you need to point your Terminal at the directory that contains all of the files you have just created. I have called my directory 'ExtractingKeywordSets' and I have it on my computer's Desktop. To change the Terminal to this directory, I use the following command:

```
cd Desktop/ExtractingKeywordSets
```

You would need to change the above to reflect the name you gave your directory, and where you put it on your machine. Note that Windows uses '' instead of '/' in file paths. If you get stuck, rename your directory to `ExtractingKeywordSets` and place it on the Desktop so that you can follow along.

You can now run the program you've written with the following command:

```
python extractKeywords.py
```

Once you have run the program you should see your gazetteer printed as a Python list in the command output, along with the number of entries in your list (39). If you can, great! Move on to step 2. If the last line of your output tells you that there was 1 result, that means the code has not worked properly, since we know that there should be 39 keywords in your gazetteer. Double check your code to make sure you havn't included any typos. If you still can't solve the problem, try changing "" to "" on line three. Some text editors will use carriage returns[257] instead of 'newline characters' when creating a new line. The means 'carriage return' and should solve your problem if you're experiencing one.

Step 2: Load the texts

The second step is very similar to the first. Except this time we will load the `texts.txt` in addition to the `gazetteer.txt` file

Add the following lines to the end of your code:

```
#Import the texts you want to search
f = open('texts.txt', 'r')
allTexts = f.read().lower().split("\n")
f.close()

print allTexts
```

If you got step 1 to work, you should understand this bit as well. Before you run the code, make sure that you have saved your program or you may accidentally run the OLD version and will be confused with the result. Once you've done that, rerun the code. As a shortcut, instead of writing out

[257] 'Carriage return', *Wikipedia*: https://en.wikipedia.org/wiki/Carriage_return

the command again in the Terminal, you can press the up arrow, which should display the last command you entered. If you keep pressing the up or down arrows, you can scroll through previous commands, saving yourself the time needed to retype them. Once you've found the command for running the program, press the return key to run the code.

If the code worked, you should see a big wall of text. Those are the texts we input into the program. As long as you see them, you're ok. Before moving on to the next step, delete the three lines from your code beginning with 'print'. Now that we know they are printing the contents of these files properly we do not need to continue to check. Move on to step 3.

Step 3: Remove unwanted punctuation

When matching strings, you have to make sure the punctuation doesn't get in the way. Technically, 'London.' is a different string than 'London' or ';London' because of the added punctuation. These three strings which all mean the same thing to us as human readers will be viewed by the computer as distinct entities. To solve that problem, the easiest thing to do is just to remove all of the punctuation. You can do this with regular expressions,[258] and Doug Knox and Laura Turner O'Hara have provided great introductions at *Programming Historian* for doing so.[259]

To keep things simple, this program will just replace the most common types of punctuation with nothing instead (effectively deleting punctuation).

Add the following lines to your program:

```
for entry in allTexts:
    #for each entry:
    allWords = entry.split(' ')
    for words in allWords:

        #remove punctuation that will interfere with matching
        words = words.replace(',', '')
        words = words.replace('.', '')
        words = words.replace(';', '')
```

The 'allTexts' list variable contains all of our texts. Using a for loop, we will look at each entry in turn.

Since we are interested in single words, we will split the text into single words by using the .split() method, looking explicitly for spaces: `entry.split(' ')`. Note that there is a single space between those quotation marks. Since words are generally separated by spaces, this should work fairly well. This means we now have a Python list called 'allWords' that contains each word in a single bibliographic entry.

[258] 'Regular expressions', *Wikipedia*: https://en.wikipedia.org/wiki/Regular_expression

[259] Doug Knox, 'Understanding Regular Expressions' (2013); Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).

We use another for loop to look through each word in that list, and wherever we find a comma, period, or semi-colon, we replace it with nothing, effectively deleting it. Note that there is no space between those quotation marks in the last three lines.

We now have a clean set of words that we can compare against our gazetteer entries, looking for matches.

Step 4: Look for matching keywords

As the words from our text are already in a list called 'allWords', and all of our keywords are in a list called 'allKeywords', all we have to do now is check our texts for the keywords.

First, we need somewhere to store details of any matches we have. Immediately after the 'for entry in allTexts:' line, at one level of indentation, add the following two lines of code:

```
    matches = 0
    storedMatches = []
```

Indentation is important in Python. The above two lines should be indented one tab deeper than the for loop above it. That means the code is to run every time the for loop runs - it is part of the loop. If your text editor does not allow tabs, you can use spaces instead.

The 'storedMatches' variable is a blank list, where we can store our matching keywords. The 'matches' variable is known as a 'flag', which we will use in the next step when we start printing the output.

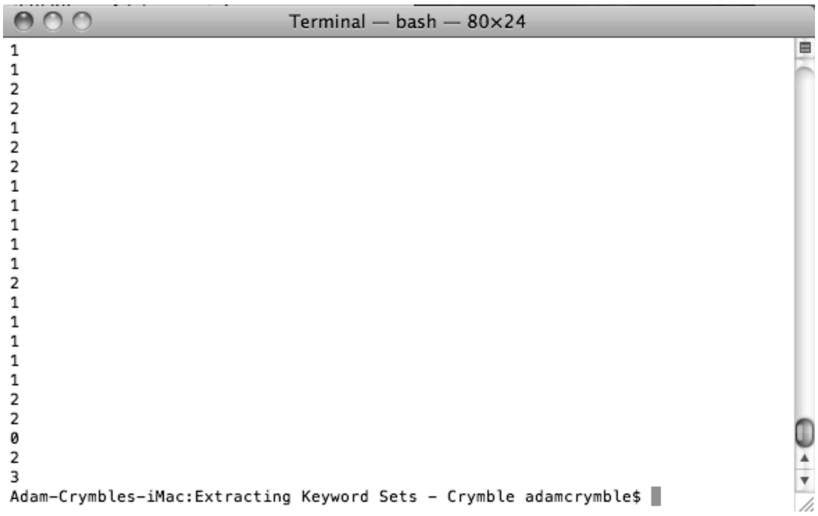
To do the actual matching, add the following lines of code to the bottom of your program, again minding the indentation (2 levels from the left margin), making sure you save:

```
        #if a keyword match is found, store the result.
        if words in allKeywords:
            if words in storedMatches:
                continue
            else:
                storedMatches.append(words)
            matches += 1
    print matches
```

If you are worried that you have your indentation wrong, scroll ahead towards the bottom of the lesson and check the finished code.

Take a look at your whole program. These lines follow immediately after the last section in which you removed the punctuation. So each time a word had its punctuation removed (if it had punctuation to remove in the first place) it was immediately checked to see if it was in the list of keywords in your gazetteer file. If it was a match, we check that we do not already have this word recorded in our 'storedMatches' variable. If we do, we skip ahead to the next word. If it is not already recorded, we append it to the 'storedMatches' list. This is keeping track of the matching words for us for each text. When we find a match, we also increase our 'matches' flag by 1. This lets us know how many matches we have found for that entry.

This code will automatically check each word in a text, keeping track of matches in the 'storedMatches' list. When it gets to the end of a text, it will empty out the 'storedMatches' variable and start again. Printing out the 'matches' variable just lets us see how many matches we got for each text. When you run this code you should see somewhere between 0 and 2 for most entries. If it says 0 for everything then check your code again. If you only have one entry outputting then go back to step one and make sure your program is identifying the right number of keywords (39).



```
1
1
2
2
1
2
2
1
1
1
1
1
2
1
1
1
1
1
2
2
0
2
3
Adam-Crymbles-iMac:Extracting Keyword Sets - Crymble adamcrymble$
```

Correct output of the code to this point

If it looks like it worked, delete the 'print matches' line and move to the next step.

Step 5: Output results

If you have got to this stage, then your Python program is already finding the matching keywords from your gazetteer. All we need to do now is print them out to the command output pane in a format that's easy to work with.

Add the following lines to your program, minding the indentation as always:

```
    #if there is a stored result, print it out
    if matches == 0:
        print ' '
    else:
        matchString = ''
        for matches in storedMatches:
            matchString = matchString + matches + "\t"

        print matchString
```

This code checks if the number of matches is equal to 0. If so, then we havn't found any keywords and we don't need to print them out. However, we are going to print a blank space, because we want our output to contain the same number of lines as did our input (we want 1 line of output per line of text that we searched). This will make it easier to paste the output directly into our CSV file and have all of the entries line up properly with their corresponding text.

If there IS a match, then the program creates a new variable called 'matchString' (it could have been called just about anything. That's just the name I chose because it's a string of matches). Then for each of the matching keywords that were kept in 'storedMatches', it appends the keyword to 'matchString', along with a tab ("") character. The tab character is useful for CSV files because when you paste it into a spreadsheet, content separated by a tab will automatically go into an adjacent cell. This means that if a single text has more than one match, we'll be able to automatically paste one match per cell. This makes it easier to keep the keywords separate once we have them back in our CSV file.

When all of the matching keywords have been added to 'matchString', the program prints it out to the command output before moving on to the next text.

If you save your work and run the program, you should now have code that achieves all of the steps from the algorithm and outputs the results to your command output.

The finished code should look like this:

Finished Code

```
#Import the keywords
f = open('gazetteer.txt', 'r')
allKeywords = f.read().lower().split("\n")
f.close()

#Import the texts you want to search
f = open('texts.txt', 'r')
allTexts = f.read().lower().split("\n")
f.close()

#Our programme:
for entry in allTexts:
    matches = 0
    storedMatches = []

    #for each entry:
    allWords = entry.split(' ')
    for words in allWords:

        #remove punctuation that will interfere with matching
        words = words.replace(',', '')
        words = words.replace('.', '')
        words = words.replace(';', '')


        #if a keyword match is found, store the result.
        if words in allKeywords:
            if words in storedMatches:
                continue
            else:
                storedMatches.append(words)
            matches += 1

    #if there is a stored result, print it out
    if matches == 0:
        print ' '
    else:
        matchString = ''
        for matches in storedMatches:
            matchString = matchString + matches + "\t"

        print matchString
```

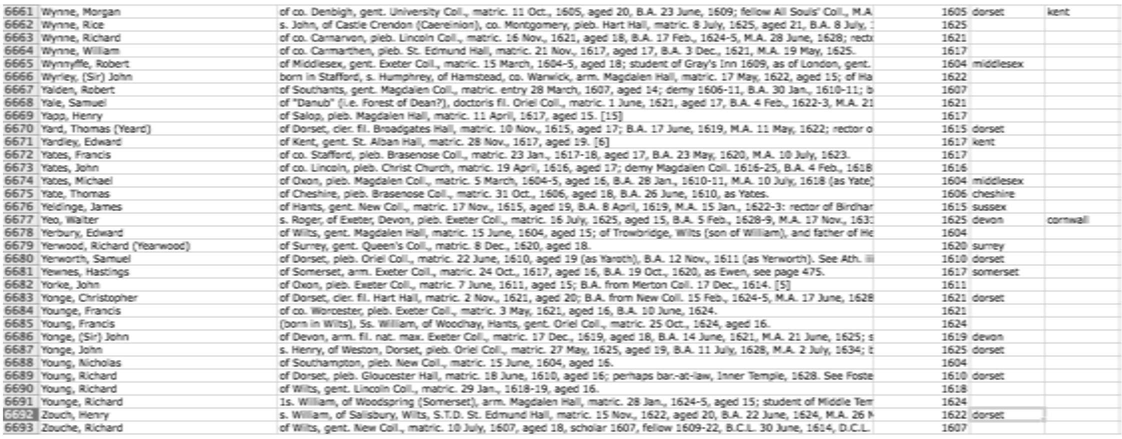
If you do not like the output format, you can change it by adjusting the second last line of code. For example, you could save each entry to a new line in a .txt file rather than to the screen. To do that you would replace 'print matchString' with the following code (make sure it is at one level of indentation, just as was the replaced line):

```
    f = open('output.txt', 'a')
    f.write(matchString)
    f.close()
```

Note the 'a' instead of the 'r' we used earlier. This 'appends' the text to the file called `output.txt`, which will be saved in your working directory. You will have to take care, because running the program several times will continue to append all of the outputs to this file, creating a very long file. There are ways around this, which we will cover in a moment, and you might consider looking into how the 'w' (write) feature works, and experimenting with output formats. There is more information related to these features in 'Working with Text Files in Python'.[260]

Refining the Gazetteer

You can copy and paste that output directly into your spreadsheet next to the first entry. Check that the matches lined up properly. Your last entry of your spreadsheet should correspond to the correctly extracted keywords. In this case, the last entry should be blank, but the second last one should read 'dorset'.



6661	Wynne, Morgan	of co. Denbigh, gent. University Coll., matric. 11 Oct., 1605, aged 20, B.A. 23 June, 1609; fellow All Souls' Coll., M.A	1605	dorset	kent
6662	Wynne, Rice	s. John, of Castle Crendon (Caereinion), co. Montgomery, pleb. Hart Hall, matric. 8 July, 1625, aged 21, B.A. 8 July, 1	1625		
6663	Wynne, Richard	of co. Carnarvon, pleb. Lincoln Coll., matric. 16 Nov., 1621, aged 18, B.A. 17 Feb., 1624-5, M.A. 28 June, 1628; rect	1621		
6664	Wynne, William	of co. Carmarthen, pleb. St. Edmund Hall, matric. 21 Nov., 1617, aged 17, B.A. 3 Dec., 1621, M.A. 19 May, 1625.	1617		
6665	Wynnyffe, Robert	of Middlesex, gent. Exeter Coll., matric. 15 March, 1604-5, aged 18; student of Gray's Inn 1609, as of London, gent.	1604	middlesex	
6666	Wyrley, (Sir) John	born in Stafford, s. Humphrey, of Hamstead, co. Warwick, arm. Magdalen Hall, matric. 17 May, 1622, aged 15; of Ha	1622		
6667	Yalden, Robert	of Southants, gent. Magdalen Coll., matric. entry 28 March, 1607, aged 14; demy 1606-11, B.A. 30 Jan., 1610-11; b	1607		
6668	Yale, Samuel	of "Danub" (i.e. Forest of Dean?), doctoris fil. Oriel Coll., matric. 1 June, 1621, aged 17, B.A. 4 Feb., 1622-3, M.A. 21	1621		
6669	Yapp, Henry	of Salop, pleb. Magdalen Hall, matric. 11 April, 1617, aged 15. [15]	1617		
6670	Yard, Thomas (Yeard)	of Dorset, cler. fil. Broadgates Hall, matric. 10 Nov., 1615, aged 17; B.A. 17 June, 1619, M.A. 11 May, 1622; rector o	1615	dorset	
6671	Yardley, Edward	of Kent, gent. St. Alban Hall, matric. 28 Nov., 1617, aged 19. [6]	1617	kent	
6672	Yates, Francis	of co. Stafford, pleb. Brasenose Coll., matric. 23 Jan., 1617-18, aged 17, B.A. 23 May, 1620, M.A. 10 July, 1623.	1617		
6673	Yates, John	of co. Lincoln, pleb. Christ Church, matric. 19 April, 1616, aged 17; demy Magdalen Coll. 1616-25, B.A. 4 Feb., 1618	1616		
6674	Yates, Michael	of Oxon, pleb. Magdalen Coll., matric. 5 March, 1604-5, aged 16, B.A. 28 Jan., 1610-11, M.A. 10 July, 1618 (as Yate)	1604	middlesex	
6675	Yate, Thomas	of Cheshire, pleb. Brasenose Coll., matric. 31 Oct., 1606, aged 18, B.A. 26 June, 1610, as Yates.	1606	cheshire	
6676	Yeldinge, James	of Hants, gent. New Coll., matric. 17 Nov., 1615, aged 19, B.A. 8 April, 1619, M.A. 15 Jan., 1622-3: rector of Birdhar	1615	sussex	
6677	Yeo, Walter	s. Roger, of Exeter, Devon, pleb. Exeter Coll., matric. 16 July, 1625, aged 15, B.A. 5 Feb., 1628-9, M.A. 17 Nov., 1631	1625	devon	cornwall
6678	Yerbury, Edward	of Wilts, gent. Magdalen Hall, matric. 15 June, 1604, aged 15; of Trowbridge, Wilts (son of William), and father of He	1604		
6679	Yerwood, Richard (Yearwood)	of Surrey, gent. Queen's Coll., matric. 8 Dec., 1620, aged 18.	1620	surrey	
6680	Yerworth, Samuel	of Dorset, pleb. Oriel Coll., matric. 22 June, 1610, aged 19 (as Yaroth), B.A. 12 Nov., 1611 (as Yerworth). See Ath. iii	1610	dorset	
6681	Yewnes, Hastings	of Somerset, arm. Exeter Coll., matric. 24 Oct., 1617, aged 16, B.A. 19 Oct., 1620, as Ewen, see page 475.	1617	somerset	
6682	Yorke, John	of Oxon, pleb. Exeter Coll., matric. 7 June, 1611, aged 15; B.A. from Merton Coll. 17 Dec., 1614. [5]	1611		
6683	Yonge, Christopher	of Dorset, cler. fil. Hart Hall, matric. 2 Nov., 1621, aged 20; B.A. from New Coll. 15 Feb., 1624-5, M.A. 17 June, 1628	1621	dorset	
6684	Younge, Francis	of co. Worcester, pleb. Exeter Coll., matric. 3 May, 1621, aged 16, B.A. 10 June, 1624.	1621		
6685	Young, Francis	(born in Wilts), 5s. William, of Woodhay, Hants, gent. Oriel Coll., matric. 25 Oct., 1624, aged 16.	1624		
6686	Yonge, (Sir) John	of Devon, arm. fil. nat. max. Exeter Coll., matric. 17 Dec., 1619, aged 18, B.A. 14 June, 1621, M.A. 21 June, 1625; s	1619	devon	
6687	Yonge, John	s. Henry, of Weston, Dorset, pleb. Oriel Coll., matric. 27 May, 1625, aged 19, B.A. 11 July, 1628, M.A. 2 July, 1634; t	1625	dorset	
6688	Young, Nicholas	of Southampton, pleb. New Coll., matric. 15 June, 1604, aged 16.	1604		
6689	Young, Richard	of Dorset, pleb. Gloucester Hall, matric. 18 June, 1610, aged 16; perhaps bar.-at-law, Inner Temple, 1628. See Foste	1610	dorset	
6690	Young, Richard	of Wilts, gent. Lincoln Coll., matric. 29 Jan., 1618-19, aged 16.	1618		
6691	Younge, Richard	1s. William, of Woodspring (Somerset), arm. Magdalen Hall, matric. 28 Jan., 1624-5, aged 15; student of Middle Tem	1624		
6692	Zouch, Henry	s. William, of Salisbury, Wilts, S.T.D. St. Edmund Hall, matric. 15 Nov., 1622, aged 20, B.A. 22 June, 1624, M.A. 26 M	1622	dorset	
6693	Zouche, Richard	of Wilts, gent. New Coll., matric. 10 July, 1607, aged 18, scholar 1607, fellow 1609-22, B.C.L. 30 June, 1614, D.C.L.	1607		

The output pasted back into the CSV file

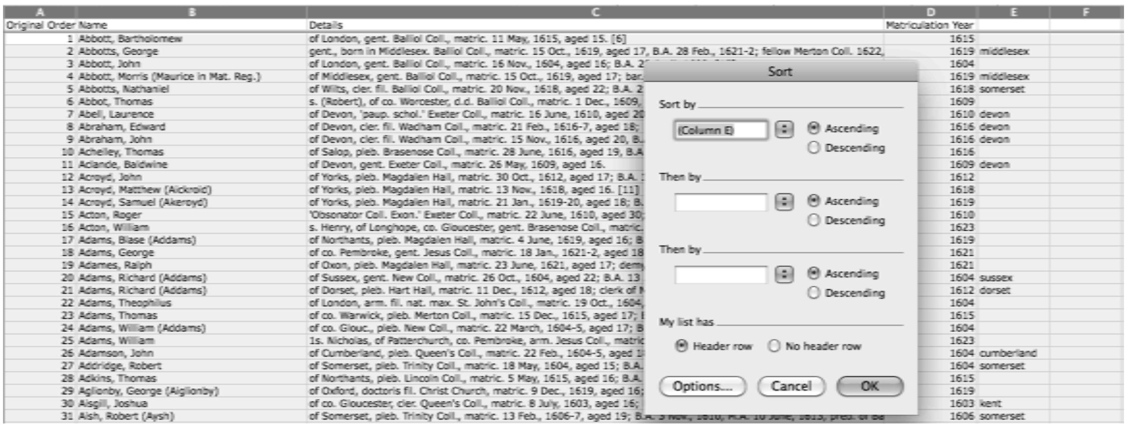
At this point, you might like to refine the gazetteer, as a lot of place names have been missed. Many of them are shortforms, or archaic spellings (Wilts, Salop, Sarum, Northants, etc). You could go through looking at all the empty cells and seeing if you can find keywords that you've missed. It may help to know that you can find the next empty cell in a column in Excel by pressing CTRL + down arrow (CMD + down arrow on Mac).

One of the easiest ways to find all of the missing entries is to sort your spreadsheet by the new columns you've just added. If you sort the matches alphabetically for each of the new columns, then the entries at the bottom of the spreadsheet will all be unclassified. You can do this by selecting the whole spreadsheet and clicking on the Data -> Sort menu item. You can then sort a-z for each of the new columns.

Before you sort a spreadsheet, it's often a good idea to add an 'original order' column in case you want to sort them back. To do this, add a new column, and in the first 3 rows, type 1, 2, and 3 respectively. Then highlight the three cells and put your cursor over the bottom right corner.

[260] William J. Turkel and Adam Crymble, 'Working with Text Files in Python', *The Programming Historian* (2012).

If you are using Microsoft Excel your cursor will change into a black cross. When you see this, click and hold the mouse button and drag the cursor down until you reach the bottom of the spreadsheet (down to the last entry) before you let go. This should automatically number the rows consecutively so that you can always re-sort your entries back to the original order.



A	B	C	D	E	F
Original Order	Name	Details	Matriculation Year		
1	Abbott, Bartholomew	of London, gent. Balliol Coll., matric. 11 May, 1615, aged 15. [6]	1615		
2	Abbotts, George	gent., born in Middlesex. Balliol Coll., matric. 15 Oct., 1619, aged 17, B.A. 28 Feb., 1621-2; fellow Merton Coll. 1622,	1619	middlesex	
3	Abbott, John	of London, gent. Balliol Coll., matric. 16 Nov., 1604, aged 16; B.A. 2	1604		
4	Abbott, Morris (Maurice in Mat. Reg.)	of Middlesex, gent. Balliol Coll., matric. 15 Oct., 1619, aged 17; bar.	1619	middlesex	
5	Abbotts, Nathaniel	of Wilts, cler. fil. Balliol Coll., matric. 20 Nov., 1618, aged 22; B.A. 2	1618	somerset	
6	Abbot, Thomas	s. (Robert), of co. Worcester, d.d. Balliol Coll., matric. 1 Dec., 1609,	1609		
7	Abell, Laurence	of Devon, 'paup. schol.' Exeter Coll., matric. 16 June, 1610, aged 2	1610	devon	
8	Abraham, Edward	of Devon, cler. fil. Wadham Coll., matric. 21 Feb., 1616-7, aged 18;	1616	devon	
9	Abraham, John	of Devon, cler. fil. Wadham Coll., matric. 15 Nov., 1616, aged 20, B.	1616	devon	
10	Achelley, Thomas	of Salop, pleb. Brasenose Coll., matric. 28 June, 1616, aged 19, B.A	1616		
11	Aclande, Baldwine	of Devon, gent. Exeter Coll., matric. 26 May, 1609, aged 16.	1609	devon	
12	Acroyd, John	of Yorks, pleb. Magdalen Hall, matric. 30 Oct., 1612, aged 17; B.A.	1612		
13	Acroyd, Matthew (Aickroid)	of Yorks, pleb. Magdalen Hall, matric. 13 Nov., 1618, aged 16. [11]	1618		
14	Acroyd, Samuel (Akeroyd)	of Yorks, pleb. Magdalen Hall, matric. 21 Jan., 1619-20, aged 18; B.	1619		
15	Acton, Roger	'Obsonator Coll. Exon.' Exeter Coll., matric. 22 June, 1610, aged 30;	1610		
16	Acton, William	s. Henry, of Longhope, co. Gloucester, gent. Brasenose Coll., matric.	1623		
17	Adams, Blase (Addams)	of Northants, pleb. Magdalen Hall, matric. 4 June, 1619, aged 16; B	1619		
18	Adams, George	of co. Pembroke, gent. Jesus Coll., matric. 18 Jan., 1621-2, aged 18	1621		
19	Adames, Ralph	of Oxon, pleb. Magdalen Hall, matric. 23 June, 1621, aged 17; demy	1621		
20	Adams, Richard (Addams)	of Sussex, gent. New Coll., matric. 26 Oct., 1604, aged 22; B.A. 13	1604	sussex	
21	Adams, Richard (Addams)	of Dorset, pleb. Hart Hall, matric. 11 Dec., 1612, aged 18; clerk of M	1612	dorset	
22	Adams, Theophilus	of London, arm. fil. nat. max. St. John's Coll., matric. 19 Oct., 1604,	1604		
23	Adams, Thomas	of co. Warwick, pleb. Merton Coll., matric. 15 Dec., 1615, aged 17;	1615		
24	Adams, William (Addams)	of co. Glouc., pleb. New Coll., matric. 22 March, 1604-5, aged 17; B	1604		
25	Adams, William	1s. Nicholas, of Patterchurch, co. Pembroke, arm. Jesus Coll., matric	1623		
26	Adamson, John	of Cumberland, pleb. Queen's Coll., matric. 22 Feb., 1604-5, aged 1	1604	cumberland	
27	Addridge, Robert	of Somerset, pleb. Trinity Coll., matric. 18 May, 1604, aged 15; B.A.	1604	somerset	
28	Adkins, Thomas	of Northants, pleb. Lincoln Coll., matric. 5 May, 1615, aged 16; B.A.	1615		
29	Aglionby, George (Aiglionby)	of Oxford, doctoris fil. Christ Church, matric. 9 Dec., 1619, aged 16;	1619		
30	Aisgill, Joshua	of co. Gloucester, cler. Queen's Coll., matric. 8 July, 1603, aged 16;	1603	kent	
31	Aish, Robert (Aysh)	of Somerset, pleb. Trinity Coll., matric. 13 Feb., 1606-7, aged 19; B.A. 3 Nov., 1610, M.A. 10 June, 1613, preb. of Ba	1606	somerset	

Sort

Sort by: (Column E) ● Ascending ○ Descending

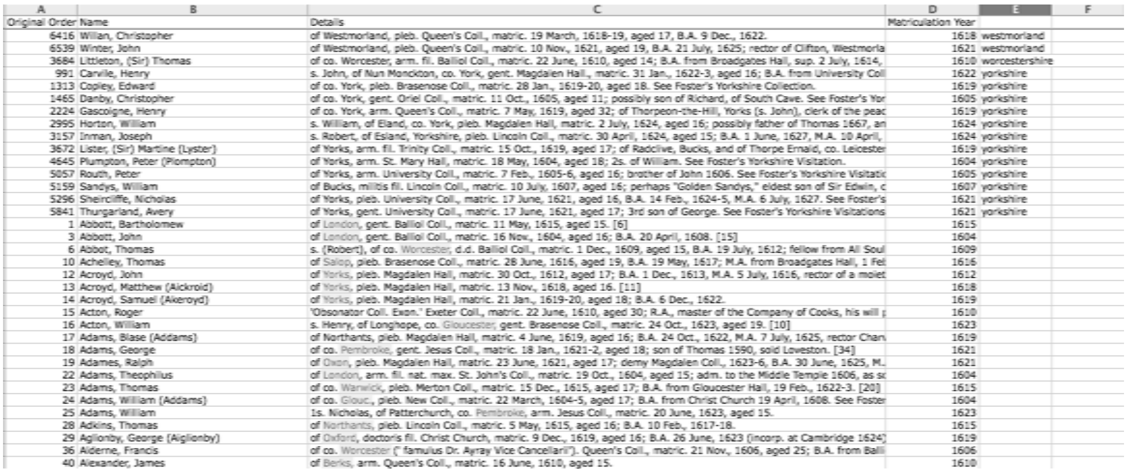
Then by: ● Ascending ○ Descending

Then by: ● Ascending ○ Descending

My list has: ● Header row ○ No header row

Adding an original order column and sorting the entries

Now you can sort the data and read some of the entries for which no match was found. If you find there is a place name in there, add it to your 'gazetteer.txt' file, one entry per line. You don't have to be exhaustive at this stage. You could add a handful more entries and then try the code again to see what impact they had on the result.



A	B	C	D	E	F
Original Order	Name	Details	Matriculation Year		
6416	Wilan, Christopher	of Westmorland, pleb. Queen's Coll., matric. 19 March, 1618-19, aged 17, B.A. 9 Dec., 1622.	1618	westmorland	
6539	Winter, John	of Westmorland, pleb. Queen's Coll., matric. 10 Nov., 1621, aged 19, B.A. 21 July, 1625; rector of Clifton, Westmorla	1621	westmorland	
3684	Littleton, (Sir) Thomas	of co. Worcester, arm. fil. Balliol Coll., matric. 22 June, 1610, aged 14; B.A. from Broadgates Hall, sup. 2 July, 1614,	1610	worcestershire	
991	Carvile, Henry	s. John, of Nun Monckton, co. York, gent. Magdalen Hall., matric. 31 Jan., 1622-3, aged 16; B.A. from University Coll	1622	yorkshire	
1313	Copley, Edward	of co. York, pleb. Brasenose Coll., matric. 28 Jan., 1619-20, aged 18. See Foster's Yorkshire Collection.	1619	yorkshire	
1465	Danby, Christopher	of co. York, gent. Oriel Coll., matric. 11 Oct., 1605, aged 11; possibly son of Richard, of South Cave. See Foster's Yor	1605	yorkshire	
2224	Gascoigne, Henry	of co. York, arm. Queen's Coll., matric. 7 May, 1619, aged 32; of Thorpeon-the-Hill, Yorks (s. John), clerk of the peac	1619	yorkshire	
2995	Horton, William	s. William, of Eland, co. York, pleb. Magdalen Hall, matric. 2 July, 1624, aged 16; possibly father of Thomas 1667, an	1624	yorkshire	
3157	Inman, Joseph	s. Robert, of Esland, Yorkshire, pleb. Lincoln Coll., matric. 30 April, 1624, aged 15; B.A. 1 June, 1627, M.A. 10 April,	1624	yorkshire	
3672	Lister, (Sir) Martine (Lyster)	of Yorks, arm. fil. Trinity Coll., matric. 15 Oct., 1619, aged 17; of Radclive, Bucks, and of Thorpe Ernald, co. Leicester	1619	yorkshire	
4645	Plumpton, Peter (Plompton)	of Yorks, arm. St. Mary Hall, matric. 18 May, 1604, aged 18; 2s. of William. See Foster's Yorkshire Visitation.	1604	yorkshire	
5057	Routh, Peter	of Yorks, arm. University Coll., matric. 7 Feb., 1605-6, aged 16; brother of John 1606. See Foster's Yorkshire Visitatic	1605	yorkshire	
5159	Sandys, William	of Bucks, militis fil. Lincoln Coll., matric. 10 July, 1607, aged 16; perhaps "Golden Sandys," eldest son of Sir Edwin, c	1607	yorkshire	
5296	Sheircliffe, Nicholas	of Yorks, pleb. University Coll., matric. 17 June, 1621, aged 16, B.A. 14 Feb., 1624-5, M.A. 6 July, 1627. See Foster's	1621	yorkshire	
5841	Thurgarland, Avery	of Yorks, gent. University Coll., matric. 17 June, 1621, aged 17; 3rd son of George. See Foster's Yorkshire Visitations	1621	yorkshire	
1	Abbott, Bartholomew	of London, gent. Balliol Coll., matric. 11 May, 1615, aged 15. [6]	1615		
3	Abbott, John	of London, gent. Balliol Coll., matric. 16 Nov., 1604, aged 16; B.A. 20 April, 1608. [15]	1604		
6	Abbot, Thomas	s. (Robert), of co. Worcester, d.d. Balliol Coll., matric. 1 Dec., 1609, aged 15, B.A. 19 July, 1612; fellow from All Soul	1609		
10	Achelley, Thomas	of Salop, pleb. Brasenose Coll., matric. 28 June, 1616, aged 19, B.A. 19 May, 1617; M.A. from Broadgates Hall, 1 Fel	1616		
12	Acroyd, John	of Yorks, pleb. Magdalen Hall, matric. 30 Oct., 1612, aged 17; B.A. 1 Dec., 1613, M.A. 5 July, 1616, rector of a moiet	1612		
13	Acroyd, Matthew (Aickroid)	of Yorks, pleb. Magdalen Hall, matric. 13 Nov., 1618, aged 16. [11]	1618		
14	Acroyd, Samuel (Akeroyd)	of Yorks, pleb. Magdalen Hall, matric. 21 Jan., 1619-20, aged 18; B.A. 6 Dec., 1622.	1619		
15	Acton, Roger	'Obsonator Coll. Exon.' Exeter Coll., matric. 22 June, 1610, aged 30; R.A., master of the Company of Cooks, his will p	1610		
16	Acton, William	s. Henry, of Longhope, co. Gloucester, gent. Brasenose Coll., matric. 24 Oct., 1623, aged 19. [10]	1623		
17	Adams, Blase (Addams)	of Northants, pleb. Magdalen Hall, matric. 4 June, 1619, aged 16; B.A. 24 Oct., 1622, M.A. 7 July, 1625, rector Charv	1619		
18	Adams, George	of co. Pembroke, gent. Jesus Coll., matric. 18 Jan., 1621-2, aged 18; son of Thomas 1590, sold Loveston. [34]	1621		
19	Adames, Ralph	of Oxon, pleb. Magdalen Hall, matric. 23 June, 1621, aged 17; demy Magdalen Coll., 1623-6, B.A. 30 June, 1625, M.	1621		
22	Adams, Theophilus	of London, arm. fil. nat. max. St. John's Coll., matric. 19 Oct., 1604, aged 15; adm. to the Middle Temple 1606, as sc	1604		
23	Adams, Thomas	of co. Warwick, pleb. Merton Coll., matric. 15 Dec., 1615, aged 17; B.A. from Gloucester Hall, 19 Feb., 1622-3. [20]	1615		
24	Adams, William (Addams)	of co. Glouc., pleb. New Coll., matric. 22 March, 1604-5, aged 17; B.A. from Christ Church 19 April, 1608. See Foster	1604		
25	Adams, William	1s. Nicholas, of Patterchurch, co. Pembroke, arm. Jesus Coll., matric. 20 June, 1623, aged 15.	1623		
28	Adkins, Thomas	of Northants, pleb. Lincoln Coll., matric. 5 May, 1615, aged 16; B.A. 10 Feb., 1617-18.	1615		
29	Aglionby, George (Aiglionby)	of Oxford, doctoris fil. Christ Church, matric. 9 Dec., 1619, aged 16; B.A. 26 June, 1623 (incorp. at Cambridge 1624)	1619		
36	Alderne, Francis	of co. Worcester (" famulus Dr. Ayray Vice Cancellarii"). Queen's Coll., matric. 21 Nov., 1606, aged 25; B.A. from Balli	1606		
40	Alexander, James	of Berks, arm. Queen's Coll., matric. 16 June, 1610, aged 15.	1610		

Missed place name words highlighted

Before you re-run your Python code, you'll have to update your `texts.txt` file so that the program runs on the texts in the correct order. Since the code will output the matches in the same order that it receives the files in `texts.txt`, it's important not to get this jumbled up if you've been sorting your spreadsheet where you intend to store your outputs. You can either re-sort the spreadsheet back to the original order before you run the program, or you can copy all of the cells in the 'details' column again and paste and save them into the `texts.txt` file.

I'd challenge you to make a few refinements to your gazetteer before moving ahead, just to make sure you have the hang of it.

Once you are happy with that, you can snag my completed list of English and Welsh counties, shortforms, and various other cities (London, Bristol etc) and places (Jersey, Ireland, etc).[261] My completed list contains 157 entries, and should get you all of the entries that can be extracted from the texts in this collection.

At this point you could stop, as you've achieved what you set out to do. This lesson taught you how to use a short Python program to search a fairly large number of texts for a set of keywords defined by you.

With the outputs from this lesson, you could fairly quickly map these entries by geolocating the place names. This might reveal new insights into spatial patterns of Oxford alumni.

Having the ability to search for large numbers of keywords at the same time opens up flexibility for your research process, and makes it feasible to do work that might otherwise just have seemed like it would take too long. You could try a completely different set of words, or use this technique on another set of texts entirely. The research questions are of course, endless.

If you would like to refine the program further, we can use Python to read directly from the CSV file and to print the results to a new CSV file so that everything happens automatically from the Terminal window without the need for cutting and pasting.

Printing the Results Back to the CSV File Using Python

Python has a built in code library that can handle working with CSV files, called `csv`

To use it and its features, you first have to import it. At the top of your `extractKeywords.py` program, add the following line:

```
    import csv
```

Now we are going to make some changes to our original program. Instead of cutting all of the texts into a `texts.txt` file, we'll use Python to read the data directly into our 'allTexts' variable. Replace:

```
#Import the texts you want to search
f = open('texts.txt', 'r')
allTexts = f.read().lower().split("\n")
f.close()
```

With this:

[261] http://programminghistorian.org/assets/extracting-keywords-final-gazetteer.txt

```
#Import the 'Details' column from the CSV file
allTexts = []
fullRow = []
with open('The_Dataset_-_Alumni_Oxonienses-Jas1.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        #the full row for each entry, which will be used to recreate the imp
roved CSV file in a moment
        fullRow.append((row['Name'], row['Details'], row['Matriculation Year
']))

        #the column we want to parse for our keywords
        row = row['Details'].lower()
        allTexts.append(row)
```

As this is an advanced option, I won't explain what every line does in detail, but you can take a look at the comments in the code to get an idea. Effectively this uses Python to read the CSV file and stores all of the information in the 'Details' column in the same 'allTexts' variable that we had it in previously, in exactly the same format as before. This code also stores each row of the CSV file into another list called 'fullRow', which will be used for writing a new CSV file containing our program's outputs.

There are a few extra lines of code here, but you didn't need to cut and paste anything into the `texts.txt` file, and there's no risk here of your sorting of your spreadsheet causing any issues about the order of inputs and outputs. This is therefore a more robust option. You can print out either of these variables using the 'print' feature, to make sure they contain what you'd expect of them.

TROUBLESHOOTING: If you get the following error when you attempt to read your CSV file using Python, the CSV file may have been saved on a Mac, and the Python CSV library is only able to read Windows-compatible CSV files

```
(Error: new-line character seen in unquoted field - do you need to open the
file in universal-newline mode?).
```

To solve this problem, open your CSV file in a spreadsheet program (eg., Excel) and 'Save As' and under format chose 'Windows Comma Separated (csv)'. This should solve the problem. To read more on this issue, see Stack Overflow.[262]

---

[262] 'CSV new-line character seen in unquoted field error', *Stack Overflow*: http://stackoverflow.com/questions/17315635/csv-new-line-character-seen-in-unquoted-field-error

Creating a new CSV file

Next we need to create a new CSV file where the results of the analysis can be stored. It's always a good idea to make a new file rather than try to append it to your only copy of the original data. It's also a good idea to append the current date and time to the filename for your new file. That way you can run the code lots of times as you refine everything and it will always be clear which file contains your most recent ouputs.

To do this, import the 'time' library just below where you imported the 'csv' library.

```
import time
```

And then add the following two lines of code right below where you were just working with the new CSV code:

```

#use the current date and time to create a unique output filename
timestr = time.strftime("%Y-%m-%d-(%H:%M:%S)")
filename = 'output-' + str(timestr) + '.csv'
```

This will create a variable called 'filename', which we'll use when we make the new output file.

The rest of the process involves creating that new output file, putting in the correct headers, pasting in the original data, and then pasting in our new outputs from our gazetteer matching. That involves quite a few tweaks to the original code, so to keep everything as clear as possible, I've included the finished code below. I have appended 'NEW!', 'OLD!' and 'CHANGED!' in the comments for each section so that you can see at a glance which bits have changed:

```
#NEW!
import csv
import time

#OLD! Import the keywords
f = open('gazetteer.txt', 'r')
allKeywords = f.read().lower().split("\n")
f.close()


#CHANGED! Import the 'Details' column from the CSV file
allTexts = []
fullRow = []
with open('The_Dataset_-_Alumni_Oxonienses-Jas1.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        #the full row for each entry,
        #which will be used to recreate the improved CSV file in a moment
        fullRow.append((row['Name'], row['Details'], row['Matriculation Year']))

        #the column we want to parse for our keywords
        row = row['Details'].lower()
        allTexts.append(row)
```

```
#NEW! a flag used to keep track of which row is being printed to the CSV file
counter = 0

#NEW! use the current date and time to create a unique output filename
timestr = time.strftime("%Y-%m-%d-(%H:%M:%S)")
filename = 'output-' + str(timestr) + '.csv'

#NEW! Open the new output CSV file to append ('a') rows one at a time.
with open(filename, 'a') as csvfile:

    #NEW! define the column headers and write them to the new file
    fieldnames = ['Name', 'Details', 'Matriculation Year', 'Placename']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()

    #NEW! define the output for each row and then print to the output csv file
    writer = csv.writer(csvfile)

    #OLD! this is the same as before, for currentRow in fullRow:
    for entry in allTexts:

        matches = 0
        storedMatches = []

        #for each entry:
        allWords = entry.split(' ')
        for words in allWords:

            #remove punctuation that will interfere with matching
            words = words.replace(',', '')
            words = words.replace('.', '')
            words = words.replace(';', '')

            #if a keyword match is found, store the result.
            if words in allKeywords:
                if words in storedMatches:
                    continue
                else:
                    storedMatches.append(words)
                matches += 1

        #CHANGED! send any matches to a new row of the csv file.
        if matches == 0:
            newRow = fullRow[counter]
        else:
            matchTuple = tuple(storedMatches)
            newRow = fullRow[counter] + matchTuple

        #NEW! write the result of each row to the csv file
        writer.writerows([newRow])
        counter += 1
```

The code is heavily commented so if you spend some time, you should be able to figure it out. Save this code and rerun it using Python and you should get a file called `output.csv` appearing in your working directory, which if you open it should contain all of the same information as you had before, but without the need to do any cutting or pasting.

To give a brief outline of what has been changed from the original version:

The texts were extracted automatically from the original datafile instead of having to paste them into a `texts.txt` file.

Using the 'time' library, we used the current date and time to create a unique and easily decypherable filename for our output file.

Using the 'csv' library we created a new .csv file using that filename, and put in the column headers we wanted to use.

We then ran the same matching code as before, checking 'allTexts' against 'allWords' and storing the results.

Instead of printing the results to the screen, we stored each row's original data (Name, Details, Matriculation Year) + the matches to a tuple[263] called 'newRow'.

Using the 'csv' library we wrote the 'newRow' data to the new CSV file, one row at a time.

This approach created longer and more complex code, but the result is a powerful program that reads from a CSV file, matches the texts against the contents of a gazetteer, and then automatically writes the output to a clean new CSV file with no intermediary steps for you the user. You didn't have to go that extra mile, but hopefully you can see the advantages if you made it all the way through.

Suggested Further Reading

Readers who have completed this lesson might be interested in then geo-referencing the output using the Google API and mapping the results. You can learn more about this process from Fred Gibbs's tutorial, Extract and Geocode Placenames from a Text File.[264] This will let you visualise the practical outputs of this tutorial. Alternatively, readers may be interested in Jim Clifford et. al's tutorial on georeferencing in QGIS 2.0,[265] an open source GIS program.[266]

About the Author

Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[263] 'Tuples and Sequences', *Python*: https://docs.python.org/2/tutorial/datastructures.html#tuples-and-sequences

[264] Fred Gibbs, 'Extract and Geocode Placenames from a Text File' (12 October 2012): http://fredgibbs.net/tutorials/extract-geocode-placenames-from-text-file.html

[265] Jim Clifford, Josh MacFadyen and Daniel Macfarlane, 'Georeferencing in QGIS 2.0', *The Programming Historian*, 2013.

[266] 'Geographic information system', *Wikipedia*: https://en.wikipedia.org/wiki/Geographic_information_system

26. Understanding Regular Expressions

Doug Knox – 2013

Lesson Goals

In this exercise we will use advanced find-and-replace capabilities in a word processing application in order to make use of structure in a brief historical document that is essentially a table in the form of prose. Without using a general programming language, we will gain exposure to some aspects of computational thinking, especially pattern matching, that can be immediately helpful to working historians (and others) using word processors, and can form the basis for subsequent learning with more general programming environments.

We will start with something like this:

```
Arizona. — Quarter ended June 30, 1907. Estimated population,
 122,931. Total number of deaths 292, including diphtheria 1, enteric
 fever 4, scarlet fever 11, smallpox 2, and 49 from tuberculosis.
```

And use pattern matching to transform it to something like this:

Arizona.	Quarter ended June 30, 1907.	Deaths	diphtheria	1
Arizona.	Quarter ended June 30, 1907.	Deaths	enteric fever	4
Arizona.	Quarter ended June 30, 1907.	Deaths	scarlet fever	11
Arizona.	Quarter ended June 30, 1907.	Deaths	smallpox	2
Arizona.	Quarter ended June 30, 1907.	Deaths	tuberculosis	49

What Are Regular Expressions and for Whom Is this Useful?

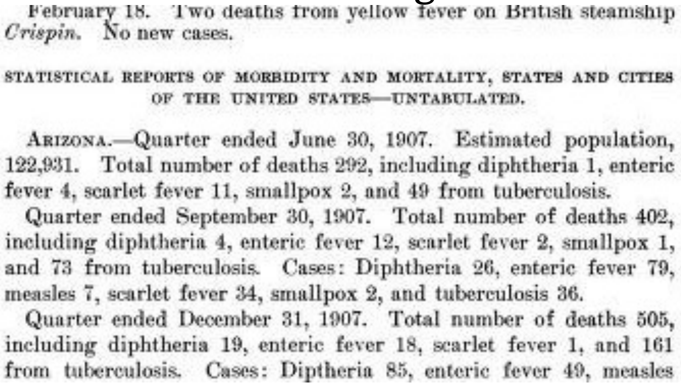
Perhaps you are not sure yet you want to be a *programming* historian, you just want to work more effectively with your sources. Historians, librarians, and others in the humanities and social sciences often work with textual sources that have implicit structure. It is also not unheard of in the humanities to have to do tedious textual work with semi-structured notes and bibliographic references, where it can help to have some knowledge of pattern-matching options.

As a simple example, if we want to find a reference to a particular year, say 1877, in a document, it's easy enough to search for that single date. But if we want to find any references to years in latter half of the 19th century, it is impractical to search several dozen times for 1850, 1851, 1852, etc., in turn. By using regular expressions we can use a concise pattern like "18[5-9][0-9]" to effectively match any year from 1850 to 1899.

In this exercise we will use LibreOffice Writer and LibreOffice Calc, which are free software desktop applications for word processing and spreadsheets, respectively. Installation packages for Linux, Mac, or Windows can be downloaded from http://www.libreoffice.org/download. Other word processing software and programming languages have similar pattern-matching capabilities. This exercise uses LibreOffice because it is freely available, and its regular expression syntax is closer to what you will find in programming environments than Microsoft Office's syntax. If you complete this exercise and find regular expressions useful, however, it should be relatively easy to adapt what you learn and apply it in other contexts.

While we will start with simple patterns, we will get to more complicated or intimidating-looking ones fairly quickly. The aim here is to share what is involved in doing useful work with a plausible example, and not to linger too long on first principles with simplified toy examples. If you are impatient, it should be possible to go through the examples fairly quickly by copying and pasting the patterns offered, without necessarily following every detail, in order to get a general sense of what is possible. If the result is promising, you could go through a second time to decide what details could be useful to pick up for your own work. But typing everything yourself is the best way to make it your own.

Getting the Text



February 18. Two deaths from yellow fever on British steamship *Crispin*. No new cases.

STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES AND CITIES OF THE UNITED STATES—UNTABULATED.

ARIZONA.—Quarter ended June 30, 1907. Estimated population, 122,931. Total number of deaths 292, including diphtheria 1, enteric fever 4, scarlet fever 11, smallpox 2, and 49 from tuberculosis.

Quarter ended September 30, 1907. Total number of deaths 402, including diphtheria 4, enteric fever 12, scarlet fever 2, smallpox 1, and 73 from tuberculosis. Cases: Diphtheria 26, enteric fever 79, measles 7, scarlet fever 34, smallpox 2, and tuberculosis 36.

Quarter ended December 31, 1907. Total number of deaths 505, including diphtheria 19, enteric fever 18, scarlet fever 1, and 161 from tuberculosis. Cases: Diptheria 85, enteric fever 49, measles

Screenshot of the unstructured text

The Internet Archive has copies of hundreds of early 20th-century public domain U.S. public health reports digitized through JSTOR and organized under the title 'Early Journal Content.' These are of a convenient length for an exercise and can plausibly represent broad classes of textual resources that are useful in many kinds of historical research. For our exercise, we will use a five-page report of monthly morbidity and mortality statistics for states and cities in the United States, published in February 1908, available at http://archive.org/details/jstor-4560629/.

Take a moment to scan the pages through the Read Online link[267] to become familiar with it. This document is organized as paragraphs rather than tables, but there are clearly latent structures that can help us tabulate this ourselves. Nearly every paragraph of the report starts with geographic information, specifies a time span for the statistics, optionally includes a population estimate, and then reports deaths and nonlethal cases of illness.

The page-flipping interface shows us what the original document looked like. But if we want to tabulate figures and enable ourselves to make comparisons and calculations over geography, we will need to represent the document as text and numbers, and not just images. In addition to offering several image formats for download, the Internet Archive makes available plain-text versions that have been created by means of Optical Character Recognition (OCR) software. OCR of old texts is often imperfect, but what it produces is useful in ways images can't be; it can be searched, copied, and edited as text.

Switch to the Full Text view[268]. We will start from this base, ignoring the last part of the previous report. Copy the text from "STATISTICAL REPORTS…" to the end into a new LibreOffice document. When working with material you care about, be sure to save a copy somewhere separately from your working copy, so that you can get back to your original if something goes wrong.

Ordinary search and replace

We can see some Optical Character Recognition (OCR) errors, where the Internet Archive's automated transcription software has made mistakes, although for the most part this looks like a good transcription. There are two places where the OCR has inserted double quotation marks into this file mistakenly, in both cases by putting them between a comma following a month and a four-digit year, as in

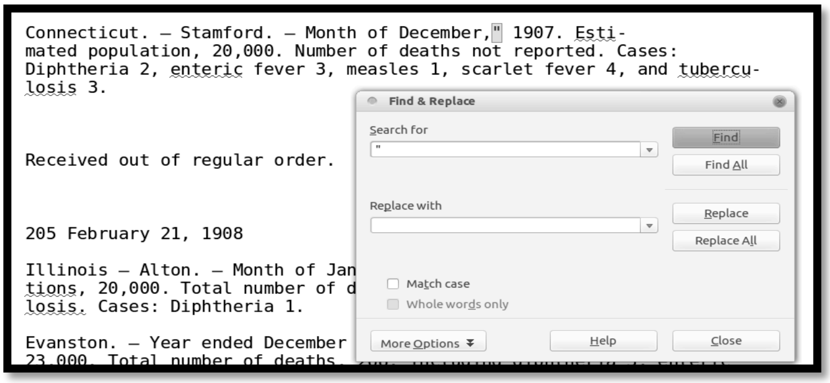
```
December," 1907.
```

We can find these by doing a search (`Edit → Find` with shortcut Ctrl-F or Cmd-F on a Mac) for double quotation marks, and confirm that these are the only two instances of quotation marks in the file. In this case we can simply delete them. Rather than do so by hand, just for practice try using LibreOffice's find-and-replace function (`Ctrl-H` or `Cmd-Alt-F` on Mac).

Replace " with nothing.

[267] 'Statistical Reports of Morbidity and Mortality, States and Cities of the United States: Untablulated', *Internet Archive*: http://archive.org/stream/jstor-4560629/4560629#page/n0/mode/2up

[268] 'Full text of "Statistical Report on Morbidity and Mortality, States and Cities of the United States: Untabulated"', *Internet Archive*: http://archive.org/stream/jstor-4560629/4560629_djvu.txt



```
Connecticut. — Stamford. — Month of December," 1907. Esti-
mated population, 20,000. Number of deaths not reported. Cases:
Diphtheria 2, enteric fever 3, measles 1, scarlet fever 4, and tubercu-
losis 3.


Received out of regular order.


205 February 21, 1908

Illinois — Alton. — Month of Jan
tions, 20,000. Total number of d
losis. Cases: Diphtheria 1.

Evanston. — Year ended December
```

Search for: "

Replace with:

□ Match case
□ Whole words only

Screenshot of Find and Replace feature

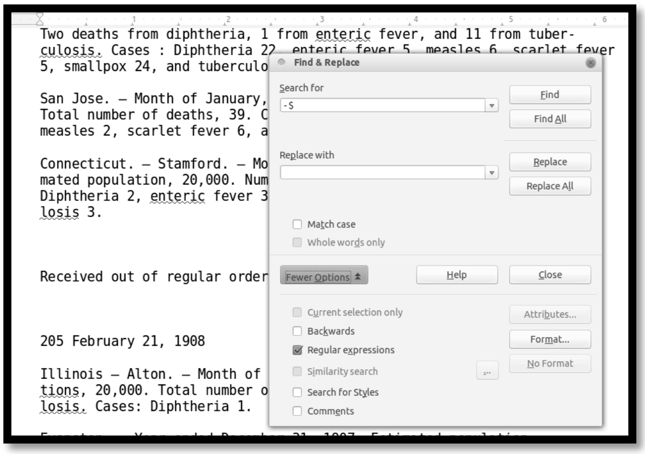
Finding structure for rows

We are just getting started, but to estimate how far we have to go, select the full text from LibreOffice Writer (`Ctrl-A`) and paste it into LibreOffice Calc (`File->New->Spreadsheet`). Each line of text becomes a single-celled row of the spreadsheet. What we would like is for each row of the spreadsheet to represent one kind of record in a consistent form. It would take a lot of tedious work to tabulate this by hand with this as our starting point. In what follows we will be doing all our work with regular expressions in Writer, but keep Calc open in the background. We can return to it to paste future iterations and gauge our progress.

Returning to Writer, we will want to get rid of the line breaks that we don't need — but there are some end-of-line hyphenations we should clean up first. This time we will start using regular expressions. On the Find & Replace box show `More Options` (Other Options on Mac) and make sure the `Regular expressions` checkbox is selected. This will enable us to use special symbols to define general patterns to match.

Using find-and-replace,

replace `-  $` (hyphen-space-dollar-sign) with nothing.



```
Two deaths from diphtheria, 1 from enteric fever, and 11 from tuber-
culosis. Cases : Diphtheria 22, enteric fever 5, measles 6, scarlet fever
5, smallpox 24, and tuberculo

San Jose. — Month of January,
Total number of deaths, 39. C
measles 2, scarlet fever 6, a

Connecticut. — Stamford. — Mo
mated population, 20,000. Num
Diphtheria 2, enteric fever 3
losis 3.


Received out of regular order


205 February 21, 1908

Illinois — Alton. — Month of
tions, 20,000. Total number o
losis. Cases: Diphtheria 1.
```

Search for: - $

Replace with:

□ Match case
□ Whole words only
□ Current selection only
□ Backwards
☑ Regular expressions
□ Similarity search
□ Search for Styles
□ Comments

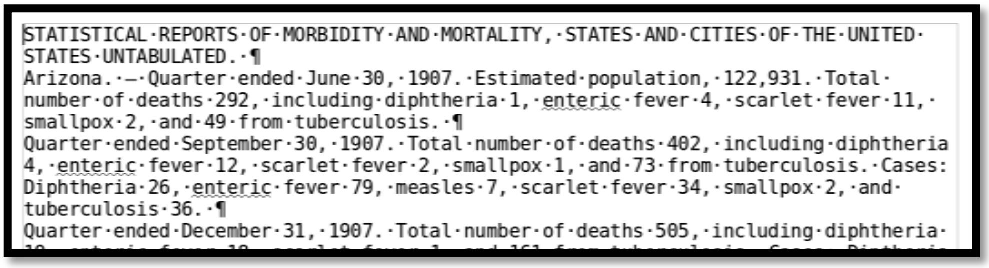
The 'More Options' tab in Open Office Find & Replace

The dollar sign symbol is a special symbol in this case that matches the end of each line. You might start by clicking `Find` and then `Replace` when you see that the highlighted selection matches your expectations. After repeating this a few times you can click `Replace All` to replace all the rest at once. If you make a mistake or are uncertain, you can undo recent steps with `Edit → Undo` from the menu bar, or keyboard shortcut `Ctrl+Z` (Cmd+Z on Mac). In this document there are 27 total matches for this particular pattern.

Next, again using find-and-replace,

replace all `$` (just a dollar sign) with nothing.

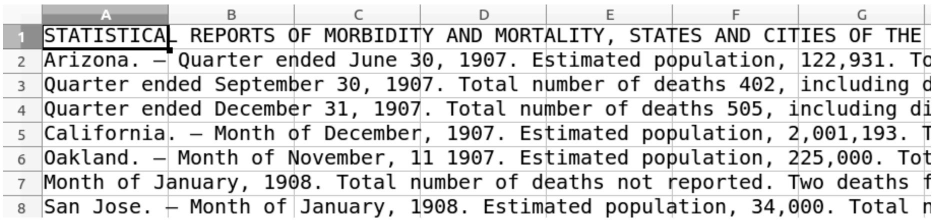
There are 225 replacements with this pattern. At first it may not be clear what happened here, but this has in fact made each paragraph a single paragraph or logical line. In LibreOffice (and similar word processing programs) you can turn on nonprinting characters (View→Nonprinting Characters with shortcut `Ctrl-F10 on Windows or Linux`) to see line and paragraph breaks.



```
STATISTICAL·REPORTS·OF·MORBIDITY·AND·MORTALITY,·STATES·AND·CITIES·OF·THE·UNITED·
STATES·UNTABULATED.·¶
Arizona.·—·Quarter·ended·June·30,·1907.·Estimated·population,·122,931.·Total·
number·of·deaths·292,·including·diphtheria·1,·enteric·fever·4,·scarlet·fever·11,·
smallpox·2,·and·49·from·tuberculosis.·¶
Quarter·ended·September·30,·1907.·Total·number·of·deaths·402,·including·diphtheria
4,·enteric·fever·12,·scarlet·fever·2,·smallpox·1,·and·73·from·tuberculosis.·Cases:
Diphtheria·26,·enteric·fever·79,·measles·7,·scarlet·fever·34,·smallpox·2,·and·
tuberculosis·36.·¶
Quarter·ended·December·31,·1907.·Total·number·of·deaths·505,·including·diphtheria·
```

Non-Printing Characters in LibreOffice

As a last way of confirming that we are starting to get a more useful structure from this, let's copy the full text from Writer again and paste it into a blank spreadsheet. This should confirm that each health record is now a separate row in the spreadsheet (although we also have page headings and footnotes mixed in — we will clean those up shortly).



```
    A         B         C         D         E         F         G
1 STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES AND CITIES OF THE
2 Arizona. — Quarter ended June 30, 1907. Estimated population, 122,931. To
3 Quarter ended September 30, 1907. Total number of deaths 402, including d
4 Quarter ended December 31, 1907. Total number of deaths 505, including di
5 California. — Month of December, 1907. Estimated population, 2,001,193. T
6 Oakland. — Month of November, 11 1907. Estimated population, 225,000. Tot
7 Month of January, 1908. Total number of deaths not reported. Two deaths f
8 San Jose. — Month of January, 1908. Estimated population, 34,000. Total n
```

The improved structure, shown in LibreOffice Calc

Finding structure for columns

Spreadsheets organize information in two dimensions, rows and columns. We have seen that lines in Writer correspond to rows in Calc. How do we make columns?

Spreadsheet software can read and write plain-text files using any of several conventions for representing breaks between columns. One common format uses commas to separate columns, and such files are often stored with the extension ".csv" for "comma-separated values." Another common variant is to use a tab character, a special kind of space, to separate columns. Because our text contains commas, to avoid confusion we will use a tab character to separate columns. Though one could save a intermediate plain-text file, in this exercise we will assume we are copying and pasting directly from Writer to Calc.

Back in Writer, let's start making columns by splitting the place-and-time information from the reported numbers. Almost all reports include the words

```
Total number of deaths
```

Search for this and replace it with exactly the same phrase, but with "\t" at the front of the string representing a tab character:

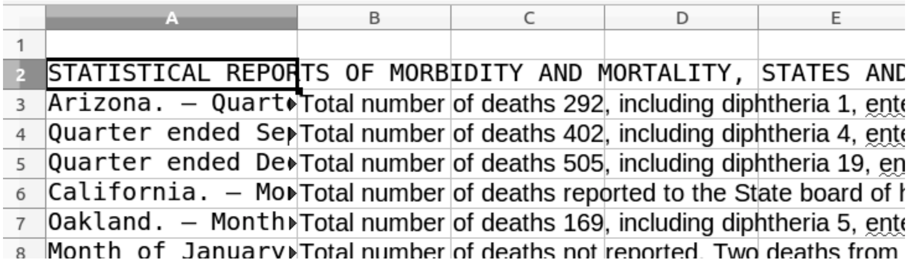
```
\tTotal number of deaths
```

After making this replacement (which makes 53 changes), select all the text and copy and paste it into an empty spreadsheet again.

Does it look like nothing changed? LibreOffice Calc is putting the full text of each paragraph in a single cell, tabs and all. We need to insist on a plain-text interpretation to get Calc to ask us what to do with tabs. Let's try again. You can empty the spreadsheet conveniently by selecting all (`Ctrl-A`) and deleting the selection.

In an empty spreadsheet, select `Edit → Paste Special,` (or right-click to reach the same) and then select "unformatted text" from the options in the window appears. That should result in a popup "Text Import" window. Make sure the Tab checkbox is selected under Separator options and then click "OK". (Before clicking OK you may want to try checking and unchecking Comma and Space as separators to preview what they would do here, but we do not want to treat them as separators in this context.)

Now we see the promising start of a table structure, with geography and time span still in column A, but with "Total number of deaths" and subsequent text clearly aligned in a separate column.



	A	B	C	D	E
1					
2	STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES AND				
3	Arizona. – Quart	Total number of deaths 292, including diphtheria 1, ente			
4	Quarter ended Se	Total number of deaths 402, including diphtheria 4, ente			
5	Quarter ended De	Total number of deaths 505, including diphtheria 19, en			
6	California. – Mo	Total number of deaths reported to the State board of			
7	Oakland. – Month	Total number of deaths 169, including diphtheria 5, ente			
8	Month of January	Total number of deaths not reported. Two deaths from			

The newly tab-delimited version of the data shown in LibreOffice Calc

Do you have any instances that moved over into a third column or beyond? In that case you may inadvertently have put in too many tabs. In the structure we have right now we don't expect to ever see two tab characters in a row. Back in LibreOffice Writer we can check for this and fix the problem by searching for

`\t\t` and replacing with `\t`

`repeating as needed` until no more double-tabs are found.

Sometimes multiple applications of a replacement pattern introduce additional changes after the first, which may or may not be what we intend, and sometimes multiple applications will have no effect beyond the first application. It is worth keeping this distinction in mind while working with regular expressions.

Regular Expression	Meaning
`[Ab1]`	a character class, matching one instance of any of `A`, `b`, or `1` in this case
`[a-z]`	all lowercase letters within a range
`[0-9]`	all digits
`.`	any character
`*`	zero or more
`+`	one or more
`( )`	if contents within parentheses match, define a group for future reference
`$1`	refer to a matched group (this is the notation in LibreOffice; other notations such as are sometimes used elsewhere)
`\t`	Tab
`^`	beginning of line
`$`	end of line

The general idea of regular expressions

Before doing any more practical work with the file, this is a good time for a brief introduction to regular expressions. Regular expressions (or "regexes" for short) are a way of defining patterns that can apply to sequences of things. They have the funny name that they do because of their origins in computer science and formal language theory, and they are incorporated into most general programming languages.

Regexes are also often available in some form in advanced word processors, providing a more powerful means of find-and-replace than matching exact sequences letter by letter. There are different syntaxes and implementations of regular expressions, and what we have available in word processing programs often isn't as extensive, robust, or in conformance with wider practice as what one finds in programming language contexts, but there are essential common principles. LibreOffice for the most part follows notational conventions that you will see in other contexts. If you use a proprietary word processor you will likely find similar functionality even if the notation differs.

For a more complete list of regular expressions in LibreOffice, see their List of Regular Expressions.[269]

Applying regular expressions

Let's start to use some of these to remove the page headings with date and page number. Switch back to your LibreOffice Writer window.

Replace:

```
^.*February 21.*1908.*$
```

with nothing (4 matches).

Replace

```
^.*Received out of regular order.*$
```

with nothing (2 matches).

Here `^` (caret) matches the beginning of the line, `.` (period) matches any character, `.*` (period-asterisk) matches any sequence of zero or more characters, and `$` (dollar-sign) matches the end of the line. By spelling out the date, we will match only the lines where that sequence appears, letter by letter, and by using `.*` at both ends we match all lines with that sequence regardless of what else is before or after it on the line. After making this replacement, we will be left with some blank lines.

To remove the blank lines in LibreOffice,

[269] 'List of Regular Expressions': https://help.libreoffice.org/Common/List_of_Regular_Expressions

Replace

```
^$
```

with nothing (5 matches).

(In other regular expression environments, other techniques for working with line endings will be necessary; some may be more convenient than what LibreOffice offers, but this will work now for our purposes.)

Some records list a state, some a city with the state implicit, some a state and city together. The text does not have enough structure to give us a reliable way of distinguishing the California and Oakland records so that we will be able automatically to put California in a state column and Oakland in a city column. We will eventually need to do some editing by hand, drawing on our own knowledge. But there is a lot of consistency in the references to spans of time. We can use those references to develop structures that will help keep similar segments aligned across rows.

For convenience, let's put some markers in the text that won't be confused with anything already present. We can easily distinguish these markers from existing text, and easily remove them later when we don't need them. Let's match time span references and put "<t>" at the beginning of them and "</t>" at the end, with the mnemonic "t" for time. We could put a more verbose marker in, like "<time>" or a more meaningless and untidy-looking one, like "asdfJKL;" as long as that sequence wasn't for some reason already in our text. But in this exercise we will use markers like "<t>" If you have seen HTML or XML, these look a lot like the tags that mark elements. We are not creating acceptable HTML or well-formed XML by doing this, and we will remove these markers quickly, but there is a resemblance.

Obligatory warning: Regular expressions are powerful, but they do have their limits and (when used to modify material that someone cares about) they can be dangerous, in that a mistake can inadvertently remove or scramble a lot of information quickly. Also, as XML aficionados may passionately tell you, regular expressions are not up to the job of general-purpose parsing of XML. After one sees how useful regular expressions are at dealing with certain kinds of patterns, there is a temptation to think, whenever we see a pattern that a computer ought to be able to help with, that regular expressions are all we need. In many cases that will turn out not to be true. Regular expressions are not adequate to deal with hierarchically nested patterns that XML is good at describing.

But that's OK. In the context of this tutorial, we don't claim to know anything in particular about XML, or to care about formal language grammars. We just want to put some convenient markers into a text in order to get some leverage in making a relatively simple implicit structure a bit more explicit, and we will take those markers out before we are done. There is a reason why such markers are useful. If you find yourself intrigued by what can be done with patterns in this exercise, you may want to learn more about HTML and XML, and learn what can be done with appropriate methods that their more explicit structure makes possible.

Defining segments

The next few patterns will rapidly get more complicated. If you slow down to consult the reference to how the symbols define patterns, however, the patterns should start to make sense.

Geographic references in our text are followed by emdashes (dashes that are roughly the width of the letter 'm'; wider than endashes.) We can replace these with tab characters, which will effectively help us put states and cities in separate columns of the spreadsheet.

Replace

```
[ ]?—[ ]?
```

with

```
\t
```

You should have 42 matches. (One easy way to get the emdash into your pattern is to copy and paste from an existing emdash in the text itself. The square brackets aren't entirely necessary here, but help make visible the fact that we are matching a blank space — optionally matching it, thanks to the question mark. That means our pattern will accept an emdash with or without a space on either or both sides of it.)

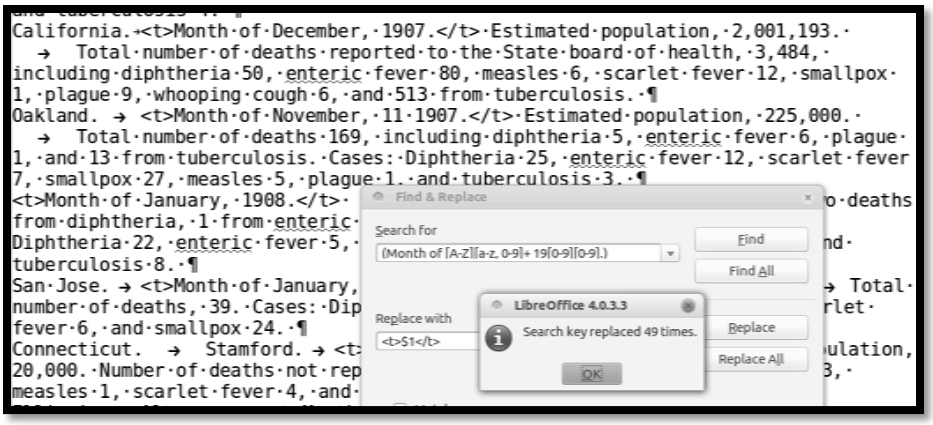
Now we will look for explicit references to time and wrap them in "<t>" and "</t>" markers before and after. Once we have those markers they will provide some scaffolding on which we can build further patterns. Note that in the next pattern we want to be sure to apply the replacement just once, otherwise some time references may be repeatedly wrapped. It will be most efficient to use `Replace All` just once for each wrapping pattern.

Replace

```
(Month of [A-Z][a-z, 0-9]+ 19[0-9][0-9].)
```

with

```
<t>$1</t>
```



```
California.→<t>Month·of·December,·1907.</t>·Estimated·population,·2,001,193.·
  →  Total·number·of·deaths·reported·to·the·State·board·of·health,·3,484,·
including·diphtheria·50,·enteric·fever·80,·measles·6,·scarlet·fever·12,·smallpox·
1,·plague·9,·whooping·cough·6,·and·513·from·tuberculosis.·¶
Oakland. → <t>Month·of·November,·11·1907.</t>·Estimated·population,·225,000.·
  →  Total·number·of·deaths·169,·including·diphtheria·5,·enteric·fever·6,·plague·
1,·and·13·from·tuberculosis.·Cases:·Diphtheria·25,·enteric·fever·12,·scarlet·fever
7,·smallpox·27,·measles·5,·plague·1.·and·tuberculosis·3.·¶
<t>Month·of·January,·1908.</t>·                                            o·deaths
from·diphtheria,·1·from·enteric·
Diphtheria·22,·enteric·fever·5,·                                            nd·
tuberculosis·8.·¶
San·Jose. → <t>Month·of·January,                                            → Total·
number·of·deaths,·39.·Cases:·Dip                                            rlet·
fever·6,·and·smallpox·24.·¶
Connecticut.  →  Stamford. → <t                                             ulation,
20,000.·Number·of·deaths·not·rep                                            3,·
measles·1,·scarlet·fever·4,·and·
```

Find & Replace

Search for

(Month of [A-Z][a-z, 0-9]+ 19[0-9][0-9].)

Replace with

<t>$1</t>

LibreOffice 4.0.3.3

Search key replaced 49 times.

Finding time using Regular Expressions

Here we are using parentheses to define everything that we match in the search pattern as a single group, and in the replacement pattern we use $1 to simply repeat that match, with a few additional characters before and after it.

In addition to months, we need to match quarterly reports with a similar approach:

Replace

```
([-A-Za-z ]+ ended [A-Z][a-z, 0-9]+ 19[0-9][0-9].)
```

with

```
<t>$1</t>
```

You should have 7 more matches. It looks like we have references to time accounted for. Extending this strategy to other kinds of information here, let's use "<p>" for population estimates, "<N>" for total number of deaths, and "<c>" for the word "Cases," which separates mortality from morbidity. (If you are familiar with HTML or XML, you may recognize "<p>" as a paragraph marker. We're not using it in the same way here.)

Here are some patterns to wrap each of those kinds of information, all using the same strategy we just used:

Replace

```
(Estimated population, [0-9,]+.)
```

with

`<p>$1</p>` (34 matches).

Replace

```
(Total number of deaths[A-Za-z ,]* [0-9,]+)
```

with

`<N>$1</N>` (48 matches).

Replace

```
(Cases ?:)
```

with

`<c>$1</c>` (49 matches).

This next part is a little trickier. It would be great if we could get hold of the disease (let's use "<d>") and count ("<n>") segments. Because the prose in this document is so formulaic, especially following the indication of total number of deaths, in this case we will be able to get pretty far without having to match each disease name explicitly, one by one. First match the disease-count pair after the word "including":

Replace

```
</N> including ([A-Za-z ]+) ([0-9]+),
```

with

`</N> including <d>$1</d> <n>$2</n>` (29 matches).

And then iteratively match disease-count pairs that appear after existing markers:

Replace

```
> ([A-Za-z ]+) ([0-9]+)([.,])
```

with

```
> <d>$1</d> <n>$2</n>
```

Note that we are getting rid of commas after the disease counts by ignoring the third match in our replacement.

Repeat this replacement as many times as necessary until there are no further matches. It should take you seven iterations.

Our patterns have not done anything with phrases like 'and 3 from tuberculosis.' We can match those phrases and reverse the order so that the disease name appears before the count:

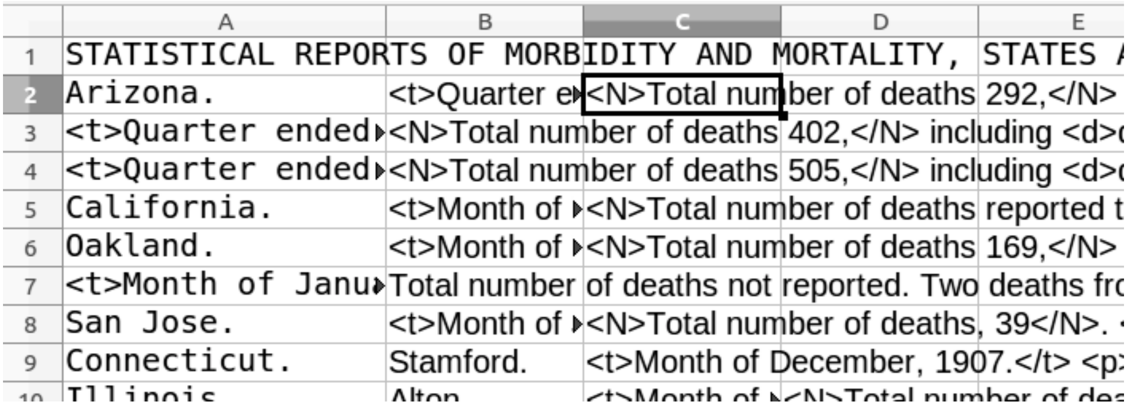
Replace

```
and ([0-9])+ from ([a-z ]+)
```

with

`<d>$2</d> <n>$1</n>` (32 matches).

It looks like our markers are now capturing a lot of the semantic structure that we are interested in. Now let's copy and paste ("paste special … unformatted") into LibreOffice Calc to see how close we are to getting a table. We are successfully separating location data into cells, but the cells are not aligned vertically yet. We want to get all of the time references into the third column.



	A	B	C	D	E
1	STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES A				
2	Arizona.	<t>Quarter e	<N>Total number of deaths		292,</N>
3	<t>Quarter ended	<N>Total number of deaths		402,</N> including <d>(	
4	<t>Quarter ended	<N>Total number of deaths		505,</N> including <d>(	
5	California.	<t>Month of	<N>Total number of deaths		reported t
6	Oakland.	<t>Month of	<N>Total number of deaths		169,</N>
7	<t>Month of Janu	Total number of deaths not reported. Two deaths fro			
8	San Jose.	<t>Month of	<N>Total number of deaths, 39</N>.		
9	Connecticut.	Stamford.	<t>Month of December, 1907.</t> <p		
10	Illinois	Alton	<t>Month of	<N>Total number of dea	

Measuring progress using LibreOffice Calc

The instances with two columns of location information should already be OK. The rows with one location need an extra column. Most are cities, so we will put the locations into the second column, and in a few instances we will need to move state names back to the first column by hand. Go back to your LibreOffice Writer window and:

Replace

```
^([A-Za-z .]+\t<t>)
```

with

`\t$1` (30 matches).

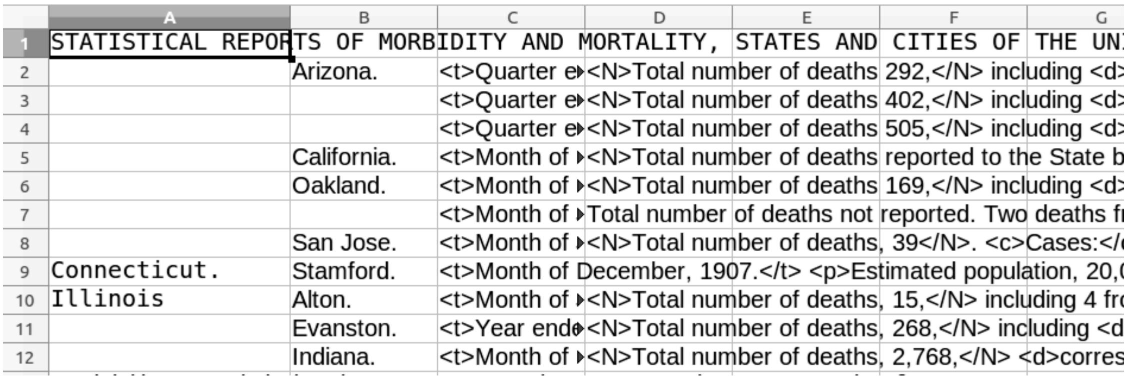
Now fix the cases with no location information, where the location is implicitly the same as the row above, and the time span is different.

Replace

```
^<t>
```

with

`\t\t<t>` (19 matches)



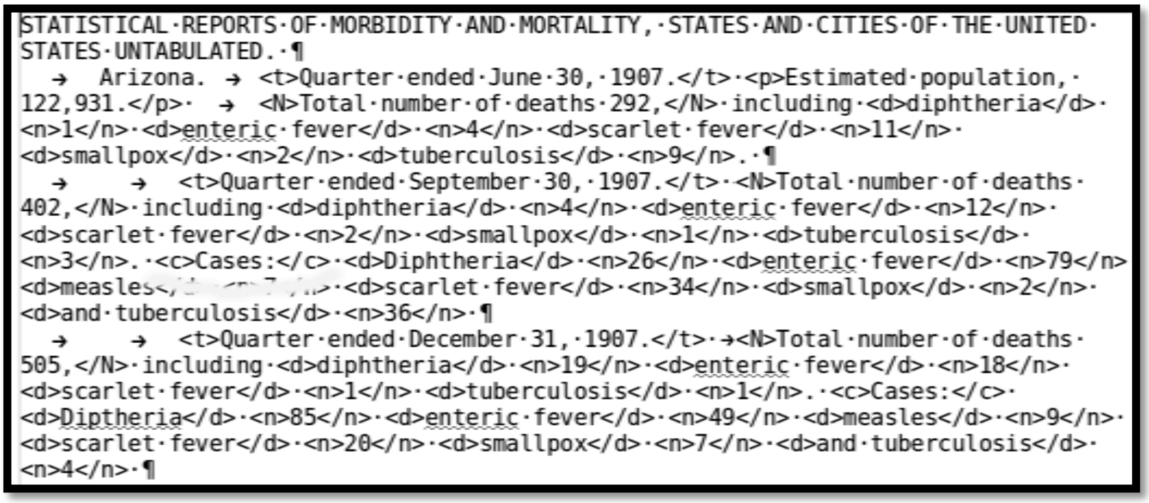
	A	B	C	D	E	F	G
1	STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES AND CITIES OF THE UN						
2		Arizona.	<t>Quarter e	<N>Total number of deaths		292,</N> including <d	
3			<t>Quarter e	<N>Total number of deaths		402,</N> including <d	
4			<t>Quarter e	<N>Total number of deaths		505,</N> including <d	
5		California.	<t>Month of	<N>Total number of deaths		reported to the State b	
6		Oakland.	<t>Month of	<N>Total number of deaths		169,</N> including <d	
7			<t>Month of	Total number of deaths not		reported. Two deaths fr	
8		San Jose.	<t>Month of	<N>Total number of deaths		, 39</N>. <c>Cases:</	
9	Connecticut.	Stamford.	<t>Month of December, 1907.</t> <p>Estimated population, 20,(				
10	Illinois	Alton.	<t>Month of	<N>Total number of deaths		, 15,</N> including 4 fro	
11		Evanston.	<t>Year ende	<N>Total number of deaths		, 268,</N> including <d	
12		Indiana.	<t>Month of	<N>Total number of deaths		, 2,768,</N> <d>corres	

Further refining the results

The first few columns should look better after pasting this again into Calc. The Writer text is still our working copy, so if you want to fix up the state names, you could do so now in Writer by deleting the tab character before a state name and introducing a new tab character after it. Or you could wait until we are done with our work in Writer, and fix them in Calc after we are ready for that to be our live working copy. But we are not there yet.

We need to decide how to handle the lists of diseases. The rows have different lists of varying lengths. While it would be easy enough now to insert tab characters to put each disease and mortality or morbidity count into a separate column, the columns would not be that helpful. Diseases and tallies would not be vertically aligned. What we can do instead is make a new row for each disease. The reports distinguish between mortality counts and morbidity counts, which are already conveniently separated by "Cases:". (There is one case, Indiana, where the text marks this section with the word "Morbidity". Our searching patterns missed this. You can fix the markup there by hand now, if you like, or ignore it since this is an exercise. It's a good example of how automated tools aren't a full substitute for editing or looking at your sources, and it won't be the last such example.)

We can start by making a new row for "cases" lists, so that we can handle them separately. Head back to LibreOffice Writer.



```
STATISTICAL·REPORTS·OF·MORBIDITY·AND·MORTALITY,·STATES·AND·CITIES·OF·THE·UNITED·
STATES·UNTABULATED.·¶
  →   Arizona.  →  <t>Quarter·ended·June·30,·1907.</t>·<p>Estimated·population,·
122,931.</p>·  →  <N>Total·number·of·deaths·292,</N>·including·<d>diphtheria</d>·
<n>1</n>·<d>enteric·fever</d>·<n>4</n>·<d>scarlet·fever</d>·<n>11</n>·
<d>smallpox</d>·<n>2</n>·<d>tuberculosis</d>·<n>9</n>.·¶
  →     →   <t>Quarter·ended·September·30,·1907.</t>·<N>Total·number·of·deaths·
402,</N>·including·<d>diphtheria</d>·<n>4</n>·<d>enteric·fever</d>·<n>12</n>·
<d>scarlet·fever</d>·<n>2</n>·<d>smallpox</d>·<n>1</n>·<d>tuberculosis</d>·
<n>3</n>.·<c>Cases:</c>·<d>Diphtheria</d>·<n>26</n>·<d>enteric·fever</d>·<n>79</n>
<d>measles</d>·<n>7</n>·<d>scarlet·fever</d>·<n>34</n>·<d>smallpox</d>·<n>2</n>·
<d>and·tuberculosis</d>·<n>36</n>·¶
  →     →   <t>Quarter·ended·December·31,·1907.</t>·→<N>Total·number·of·deaths·
505,</N>·including·<d>diphtheria</d>·<n>19</n>·<d>enteric·fever</d>·<n>18</n>·
<d>scarlet·fever</d>·<n>1</n>·<d>tuberculosis</d>·<n>1</n>.·<c>Cases:</c>·
<d>Diptheria</d>·<n>85</n>·<d>enteric·fever</d>·<n>49</n>·<d>measles</d>·<n>9</n>·
<d>scarlet·fever</d>·<n>20</n>·<d>smallpox</d>·<n>7</n>·<d>and·tuberculosis</d>·
<n>4</n>·¶
```

Making a new row for 'cases'

Replace

```
^(.*\t)(.*\t)(<t>.*</t>)(.*)(<c>.*)
```

with

`$1$2$3$4\n$1$2$3\t$5` (47 matches).

One thing to notice here is that we are using some of the replacement patterns twice. We are matching the three fields up to the time reference, then matching everything before "<c>" in a fourth group, and everything from "<c>" on in a fifth. In the replacement pattern, we put groups 1-4 back in order, then introduce a newline and print groups 1-3 again, followed by a tab and group 5. We've effectively moved the case listings to their own lines, and copied the place and time fields verbatim.

Let's go further, and split all the case lists into separate rows:

Replace

```
^(.*\t)(.*\t)(<t>.*</t>)(.*<c>.*)(<d>.*</d>) (<n>.*</n>)
```

with

```
$1$2$3$4\n$1$2$3\tCases\t$5$6
```

and **repeat** as many times as necessary until there are no more replacements (seven iterations).

Now similarly split all the mortality lists into separate rows:

Replace

```
^(.*\t)(.*\t)(<t>.*</t>)(.*<N>.*)(<d>.*</d>) (<n>.*</n>)
```

with

```
$1$2$3$4\n$1$2$3\tDeaths\t$5$6
```

and **`repeat`** as many times as necessary until there are no more replacements (eight iterations).

This is getting very close now to a tabular structure, as you can see if you paste again into Calc, though if you want to wait just a bit, some cleanup work with short and simple patterns will get us most of the rest of the way:

Replace

```
.*</c> $
```

with nothing

Replace

```
^$
```

with nothing

Replace

```
<n>
```

with

```
\t
```

Replace

```
</n>
```

with nothing

Replace

```
<d>and
```

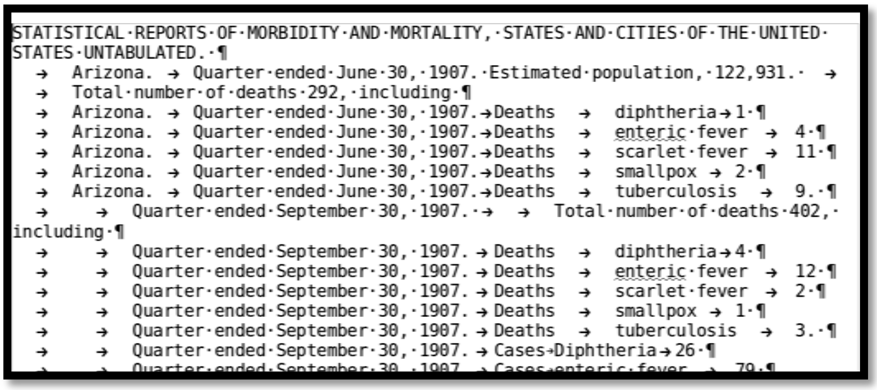
with

```
<d>
```

Replace

```
</?[tdp]>
```

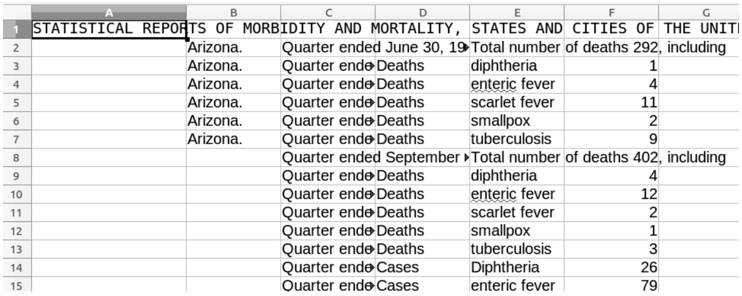
with nothing



```
STATISTICAL·REPORTS·OF·MORBIDITY·AND·MORTALITY,·STATES·AND·CITIES·OF·THE·UNITED·
STATES·UNTABULATED.·¶
  →  Arizona. → Quarter·ended·June·30,·1907.·Estimated·population,·122,931.·  →
  →  Total·number·of·deaths·292,·including·¶
  →  Arizona. → Quarter·ended·June·30,·1907.→Deaths  →  diphtheria→1·¶
  →  Arizona. → Quarter·ended·June·30,·1907.→Deaths  →  enteric·fever → 4·¶
  →  Arizona. → Quarter·ended·June·30,·1907.→Deaths  →  scarlet·fever → 11·¶
  →  Arizona. → Quarter·ended·June·30,·1907.→Deaths  →  smallpox → 2·¶
  →  Arizona. → Quarter·ended·June·30,·1907.→Deaths  →  tuberculosis  →  9.·¶
  →     →  Quarter·ended·September·30,·1907.·→  →  Total·number·of·deaths·402,·
including·¶
  →     →  Quarter·ended·September·30,·1907.→Deaths  →  diphtheria→4·¶
  →     →  Quarter·ended·September·30,·1907.→Deaths  →  enteric·fever → 12·¶
  →     →  Quarter·ended·September·30,·1907.→Deaths  →  scarlet·fever → 2·¶
  →     →  Quarter·ended·September·30,·1907.→Deaths  →  smallpox → 1·¶
  →     →  Quarter·ended·September·30,·1907.→Deaths  →  tuberculosis  →  3.·¶
  →     →  Quarter·ended·September·30,·1907.→Cases→Diphtheria→26·¶
  →     →  Quarter·ended·September·30,·1907.→Cases→enteric·fever  →  79·¶
```

The final view in LibreOffice Writer

Now copy and paste this into Calc, and you should see a (mostly) well-structured table.



	A	B	C	D	E	F	G
1	STATISTICAL REPORTS OF MORBIDITY AND MORTALITY, STATES AND CITIES OF THE UNITI						
2		Arizona.	Quarter ended June 30, 19		Total number of deaths 292, including		
3		Arizona.	Quarter ende	Deaths	diphtheria	1	
4		Arizona.	Quarter ende	Deaths	enteric fever	4	
5		Arizona.	Quarter ende	Deaths	scarlet fever	11	
6		Arizona.	Quarter ende	Deaths	smallpox	2	
7		Arizona.	Quarter ende	Deaths	tuberculosis	9	
8			Quarter ended September		Total number of deaths 402, including		
9			Quarter ende	Deaths	diphtheria	4	
10			Quarter ende	Deaths	enteric fever	12	
11			Quarter ende	Deaths	scarlet fever	2	
12			Quarter ende	Deaths	smallpox	1	
13			Quarter ende	Deaths	tuberculosis	3	
14			Quarter ende	Cases	Diphtheria	26	
15			Quarter ende	Cases	enteric fever	79	

The final view in LibreOffice Calc

If this were not an exercise but a source we were editing for research or publication, there are still things that we would need to fix. We didn't do anything with estimated population figures. Our pattern-matching wasn't sophisticated enough to manage everything. In lines that didn't have patterns like "Total number of deaths 292, including," we missed all subsequent patterns that assumed we had already put in an "</N>" marker.

Next possibilities

Some of these problems could be fixed by additional pattern-matching steps, some by hand-editing of the source document at particular points along the way, and some by later editing of the data in spreadsheet or similar tabular form.

We might want to consider other structures for the table, too — perhaps mortality and morbidity would be more convenient to tally if they were in different columns. Word processors are not the best tools for making use of these kinds of structures. Spreadsheets, XML, and programmatic tools for working with data are much more likely to be helpful. But word processors do have advanced find-and-replace functions that are good to get to know. Regular expressions and advanced pattern matching can be helpful in editing, and can provide a bridge between sequences with implicit structure and more explicit structures that we may want to match or create.

There are more than 400 public health reports like this one available from the Internet Archive. If we wanted to tabulate all of them, LibreOffice would not be the best primary tool. It would be better to learn a little Python, Ruby, or shell scripting. Programmer-oriented plain text editors, including classic ones such as Emacs and Vi or Vim, have great regular expression support as well as other features useful for dealing with plain text in a programmatic way. If you are comfortable opening up a Unix-like shell command line (in Mac or Linux, or on Windows through a virtual machine or the Cygwin environment), you can learn and use regular expressions very well with tools like "grep" for searching and "sed" for line-oriented replacing.

Regular expressions can be immensely useful in dealing with patterns across hundreds of files at once. The patterns we have used in this example would need to be refined and extended to deal with assumptions that are certain to be mistaken when applied to longer texts or larger sets of texts, but with a programming language we could record what we are doing in a short script, and refine and rerun it repeatedly to get closer to what we want.

To learn more

The Wikipedia page on regular expressions[270] is a useful place to find a brief history of regular expressions and their relation to formal language theory, as well as an overview of syntactic variants and formal standardization efforts.

The documentation for whatever tools you use will be invaluable for practical use, especially for work in word processing environments where regular expression implementations may be especially idiosyncratic. There are many resources available to learn how to use regular expressions in programming contexts; which is best for you may depend on what programming language is most familiar or convenient to start with.

There are a number of freely available web-based regular expression editors. Rubular,[271] built on the Ruby programming language, has a helpful interface that lets you test regular expressions against a sample text and dynamically shows matches and matched groups. David Birnbaum, Chair of the Department of Slavic Languages and Literatures at the University of Pittsburg, has some good materials on how to work with regular expressions and XML tools[272] to help mark up plain-text files in TEI XML. Zed Shaw has begun developing a book, freely available online, Learn Regex the Hard Way.[273] The book's exercises are built around a Python-based program developed by the author.

[270] 'Regular Expressions', *Wikipedia:* https://en.wikipedia.org/wiki/Regular_expression

[271] 'Rubular: a Ruby regular expression editor': http://rubular.com/

[272] 'Regular expressions (regex)': http://dh.obdurodon.org/regex.html

[273] Zed Shaw, 'Learn Regex the Hard Way': http://regex.learncodethehardway.org/book/

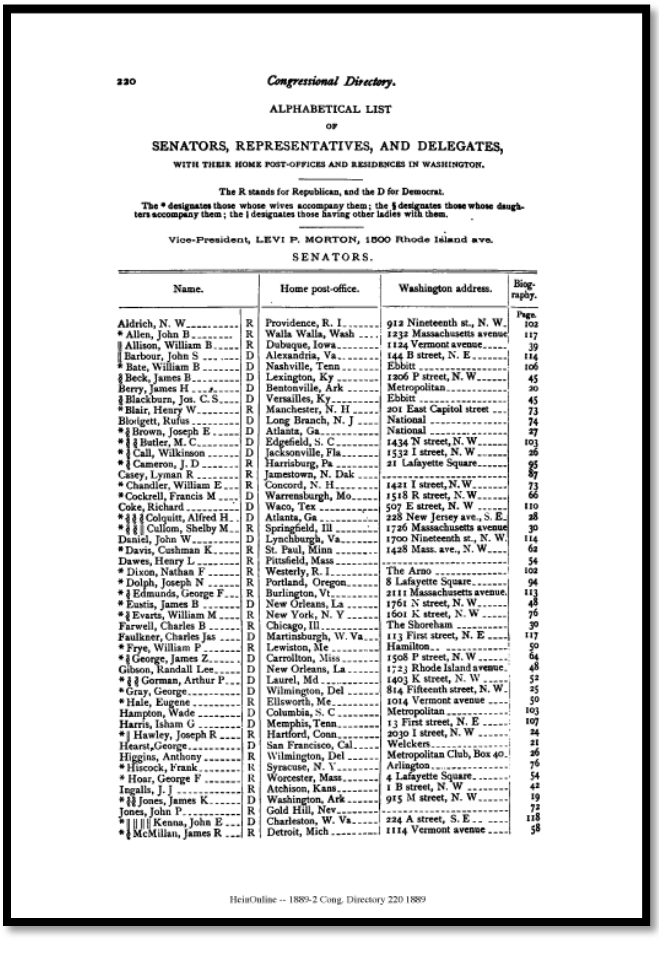
About the Author

Doug Knox is the Assistant Director of the Humanities Digital Workshop at Washington University in St. Louis.

27. Cleaning OCR'd text with Regular Expressions

Laura Turner O'Hara – 2013

Optical Character Recognition (OCR)—the conversion of scanned images to machine-encoded text—has proven a godsend for historical research. This process allows texts to be searchable on one hand and more easily parsed and mined on the other. But we've all noticed that the OCR for historic texts is far from perfect. Old type faces and formats make for unique OCR. Take for example, this page from the *Congressional Directory* from the 50th Congress (1887). The PDF scan downloaded from HeinOnline[274] looks organized:



220 *Congressional Directory.*

ALPHABETICAL LIST
OF
SENATORS, REPRESENTATIVES, AND DELEGATES,
WITH THEIR HOME POST-OFFICES AND RESIDENCES IN WASHINGTON.

The R stands for Republican, and the D for Democrat.

The * designates those whose wives accompany them; the § designates those whose daughters accompany them; the ‖ designates those having other ladies with them.

Vice-President, LEVI P. MORTON, 1800 Rhode Island ave.

SENATORS.

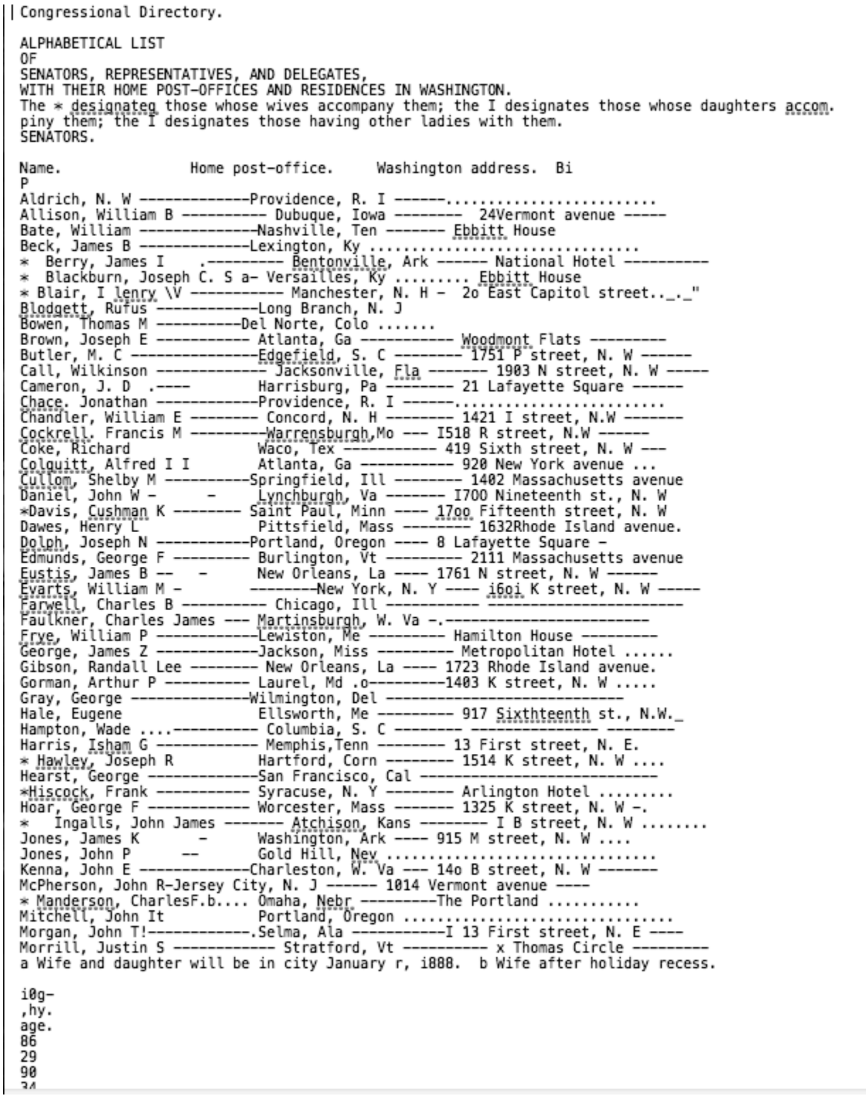
Name.		Home post-office.	Washington address.	Biography.
				Page.
Aldrich, N. W	R	Providence, R. I	912 Nineteenth st., N. W.	102
* Allen, John B	R	Walla Walla, Wash	1232 Massachusetts avenue	117
‖ Allison, William B	R	Dubuque, Iowa	1124 Vermont avenue	39
‖ Barbour, John S	D	Alexandria, Va	144 B street, N. E	114
* Bate, William B	D	Nashville, Tenn	Ebbitt	106
§ Beck, James B	D	Lexington, Ky	1206 P street, N. W	45
Berry, James H	D	Bentonville, Ark	Metropolitan	20
§ Blackburn, Jos. C. S	D	Versailles, Ky	Ebbitt	45
* Blair, Henry W	R	Manchester, N. H	201 East Capitol street	73
Blodgett, Rufus	D	Long Branch, N. J	National	74
* § Brown, Joseph E	D	Atlanta, Ga	National	27
* § § Butler, M. C	D	Edgefield, S. C	1434 N street, N. W	103
* § Call, Wilkinson	D	Jacksonville, Fla	1532 I street, N. W	26
* § Cameron, J. D	R	Harrisburg, Pa	21 Lafayette Square	95
Casey, Lyman R	R	Jamestown, N. Dak		87
* Chandler, William E	R	Concord, N. H	1421 I street, N. W	73
* Cockrell, Francis M	D	Warrensburgh, Mo	1518 R street, N. W	66
Coke, Richard	D	Waco, Tex	507 E street, N. W	110
* § § § Colquitt, Alfred H	D	Atlanta, Ga	228 New Jersey ave., S. E.	28
* § § ‖ Cullom, Shelby M	R	Springfield, Ill	1726 Massachusetts avenue	30
Daniel, John W	D	Lynchburgh, Va	1700 Nineteenth st., N. W.	114
* Davis, Cushman K	R	St. Paul, Minn	1428 Mass. ave., N. W	62
Dawes, Henry L	R	Pittsfield, Mass		54
* Dixon, Nathan F	R	Westerly, R. I	The Arno	102
* Dolph, Joseph N	R	Portland, Oregon	8 Lafayette Square	94
* § Edmunds, George F	R	Burlington, Vt	2111 Massachusetts avenue.	113
* Eustis, James B	D	New Orleans, La	1761 N street, N. W	48
* § Evarts, William M	R	New York, N. Y	1601 K street, N. W	76
Farwell, Charles B	R	Chicago, Ill	The Shoreham	30
Faulkner, Charles Jas	D	Martinsburgh, W. Va	113 First street, N. E	117
* Frye, William P	R	Lewiston, Me	Hamilton	50
* § George, James Z	D	Carrollton, Miss	1508 P street, N. W	64
Gibson, Randall Lee	D	New Orleans, La	1723 Rhode Island avenue.	48
* § § Gorman, Arthur P	D	Laurel, Md	1403 K street, N. W	52
* Gray, George	D	Wilmington, Del	814 Fifteenth street, N. W.	25
* Hale, Eugene	R	Ellsworth, Me	1014 Vermont avenue	50
Hampton, Wade	D	Columbia, S. C	Metropolitan	103
Harris, Isham G	D	Memphis, Tenn	13 First street, N. E	107
* ‖ Hawley, Joseph R	R	Hartford, Conn	2030 I street, N. W	24
Hearst, George	D	San Francisco, Cal	Welckers	21
Higgins, Anthony	R	Wilmington, Del	Metropolitan Club, Box 40.	26
* Hiscock, Frank	R	Syracuse, N. Y	Arlington	76
* Hoar, George F	R	Worcester, Mass	4 Lafayette Square	54
Ingalls, J. J	R	Atchison, Kans	1 B street, N. W	42
* §§ Jones, James K	D	Washington, Ark	915 M street, N. W	19
Jones, John P	R	Gold Hill, Nev		72
* ‖ ‖ ‖ ‖ Kenna, John E	D	Charleston, W. Va	224 A street, S. E	118
* § McMillan, James R	R	Detroit, Mich	1114 Vermont avenue	58

HeinOnline -- 1889-2 Cong. Directory 220 1889

This is a screenshot of the PDF page

[274] 'HeinOnline': http://home.heinonline.org/

However, the OCR layer (downloaded as a text file*) shows that the machine-encoded text is not nearly as neat:



```
| Congressional Directory.

ALPHABETICAL LIST
OF
SENATORS, REPRESENTATIVES, AND DELEGATES,
WITH THEIR HOME POST-OFFICES AND RESIDENCES IN WASHINGTON.
The * designateg those whose wives accompany them; the I designates those whose daughters accom.
piny them; the I designates those having other ladies with them.
SENATORS.

Name.               Home post-office.     Washington address.  Bi
P
Aldrich, N. W -------------Providence, R. I ------........................
Allison, William B ---------- Dubuque, Iowa --------  24Vermont avenue -----
Bate, William --------------Nashville, Ten ------- Ebbitt House
Beck, James B -------------Lexington, Ky ................................
*  Berry, James I    .---------- Bentonville, Ark ------ National Hotel ----------
*  Blackburn, Joseph C. S a- Versailles, Ky ......... Ebbitt House
* Blair, I lenry \V ----------- Manchester, N. H -  2o East Capitol street.._._"
Blodgett, Rufus ------------Long Branch, N. J
Bowen, Thomas M ----------Del Norte, Colo .......
Brown, Joseph E ----------- Atlanta, Ga ----------- Woodmont Flats ---------
Butler, M. C ---------------Edgefield, S. C -------- 1751 P street, N. W ------
Call, Wilkinson ------------- Jacksonville, Fla ------- 1903 N street, N. W -----
Cameron, J. D  .----        Harrisburg, Pa -------- 21 Lafayette Square ------
Chace. Jonathan ------------Providence, R. I ------........................
Chandler, William E -------- Concord, N. H -------- 1421 I street, N.W -------
Cockrell. Francis M ---------Warrensburgh,Mo --- I518 R street, N.W ------
Coke, Richard               Waco, Tex ----------- 419 Sixth street, N. W ---
Colquitt, Alfred I I        Atlanta, Ga ----------- 920 New York avenue ...
Cullom, Shelby M ----------Springfield, Ill -------- 1402 Massachusetts avenue
Daniel, John W -      -     Lynchburgh, Va ------- I7OO Nineteenth st., N. W
*Davis, Cushman K -------- Saint Paul, Minn ---- 17oo Fifteenth street, N. W
Dawes, Henry L              Pittsfield, Mass -------- 1632Rhode Island avenue.
Dolph, Joseph N -----------Portland, Oregon ---- 8 Lafayette Square -
Edmunds, George F --------- Burlington, Vt --------- 2111 Massachusetts avenue
Eustis, James B --   -      New Orleans, La ---- 1761 N street, N. W ------
Evarts, William M -        --------New York, N. Y ---- i6oi K street, N. W -----
Farwell, Charles B ---------- Chicago, Ill ----------- ----------------------
Faulkner, Charles James --- Martinsburgh, W. Va -.------------------------
Frye, William P ------------Lewiston, Me --------- Hamilton House ---------
George, James Z ------------Jackson, Miss --------- Metropolitan Hotel ......
Gibson, Randall Lee -------- New Orleans, La ---- 1723 Rhode Island avenue.
Gorman, Arthur P ---------- Laurel, Md .o---------1403 K street, N. W .....
Gray, George --------------Wilmington, Del ----------------------------
Hale, Eugene                Ellsworth, Me --------- 917 Sixthteenth st., N.W._
Hampton, Wade ....---------- Columbia, S. C -------- --------------- --------
Harris, Isham G ------------ Memphis,Tenn -------- 13 First street, N. E.
* Hawley, Joseph R          Hartford, Corn -------- 1514 K street, N. W ....
Hearst, George -------------San Francisco, Cal ----------------------------
*Hiscock, Frank ----------- Syracuse, N. Y -------- Arlington Hotel .........
Hoar, George F ------------ Worcester, Mass ------- 1325 K street, N. W -.
*   Ingalls, John James ------- Atchison, Kans -------- I B street, N. W ........
Jones, James K       -      Washington, Ark ---- 915 M street, N. W ....
Jones, John P      --       Gold Hill, Nev ...............................
Kenna, John E -------------Charleston, W. Va --- 14o B street, N. W -------
McPherson, John R-Jersey City, N. J ------ 1014 Vermont avenue ----
* Manderson, CharlesF.b.... Omaha, Nebr ---------The Portland ...........
Mitchell, John It           Portland, Oregon ................................
Morgan, John T!------------.Selma, Ala -----------I 13 First street, N. E ----
Morrill, Justin S ------------ Stratford, Vt ---------- x Thomas Circle ---------
a Wife and daughter will be in city January r, i888.  b Wife after holiday recess.

i0g-
,hy.
age.
86
29
90
34
```

This is a screenshot of the OCR

Note: If you do not have the option to download a text file, you can use the pdfminer module[275] to extract text from the pdf.

Since I want to use this to map the Washington residences for Members of these late 19th-century Congresses, how might I make this data more useable?

The answer is Regular Expressions or "regex." Here's what regex did for me. Though this is not a "real" CSV file (the commas are not quite right), it can be easily viewed in Excel and prepped for geocoding. Much better than the text file from above, right?

[275] 'PDFMiner': http://www.unixuser.org/~euske/python/pdfminer/index.html

```
Aldrich, N. W,Providence, R. I
Allison, William B, Dubuque, Iowa,24Vermont avenue,
Bate, William,Nashville, Ten, Ebbitt House
Beck, James B,Lexington, Ky
Berry, James I, Bentonville, Ark, National Hotel,
Blair, I lenry \V, Manchester, N. H,2o East Capitol stree_._'
Blodgett, Rufus,Long Branch, N. J
Bowen, Thomas M,Del Norte, Colo
Brown, Joseph E, Atlanta, Ga, Woodmont Flats,
Butler, M. C,Edgefield, S. C, 1751 P street NW
Call, Wilkinson, Jacksonville, Fla, 1903 N street NW
Cameron, J. D,Harrisburg, Pa, 21 Lafayette Square,
Chace, Jonathan,Providence, R, I
Chandler, William E, Concord, N. H, 1421 I street NW
Cockrell, Francis M,Warrensburgh,Mo, I518 R street NW
Coke, Richard,Waco, Tex, 419 Sixth street NW
Colquitt, Alfred I I,Atlanta, Ga, 920 New York avenue
Cullom, Shelby M,Springfield, Ill, 1402 Massachusetts avenue
Daniel, John W,,Lynchburgh, Va, I7OO Nineteenth st. NW
Davis, Cushman K, Saint Paul, Minn, 17oo Fifteenth street NW
Dawes, Henry L,Pittsfield, Mass, 1632Rhode Island avenue.
Dolph, Joseph N,Portland, Oregon, 8 Lafayette Square,
Edmunds, George F, Burlington, Vt, 2111 Massachusetts avenue
Eustis, James B,,New Orleans, La, 1761 N street NW
Evarts, William M,New York, N. Y, i6oi K street NW
Farwell, Charles B, Chicago, Ill,
Faulkner, Charles James, Martinsburgh, W. Va,
Frye, William P,Lewiston, Me, Hamilton House,
George, James Z,Jackson, Miss, Metropolitan Hotel
Gibson, Randall Lee, New Orleans, La, 1723 Rhode Island avenue.
Gorman, Arthur P, Laurel, Md .,1403 K street NW
Gray, George,Wilmington, Del,
Hale, Eugene,Ellsworth, Me, 917 Sixthteenth st. NW
Hampton, Wade, Columbia, S. C,
Harris, Isham G, Memphis,Tenn, 13 First street NE
Hawley, Joseph R,Hartford, Corn, 1514 K street NW
Hearst, George,San Francisco, Cal,
Hiscock, Frank, Syracuse, N. Y, Arlington Hotel
Hoar, George F, Worcester, Mass, 1325 K street NW
Ingalls, John James, Atchison, Kans, I B street NW
Jones, James K,Washington, Ark, 915 M street NW
Jones, John P,Gold Hill, Nev
Kenna, John E,Charleston, W. Va, 14o B street NW
McPherson, John ,Jersey City, N. J, 1014 Vermont avenue,
Manderson, CharlesF. Omaha, Nebr,The Portland
Morgan, John T,.Selma, Ala,I 13 First street NE
Morrill, Justin S, Stratford, Vt, x Thomas Circle
```

Regular Expressions (Regex)

Regex is not a programming language. Rather it follows a syntax used in many different languages, employing a series of characters to find and/or replace precise patterns in texts. For example, using this sample text:

```
Let's get all this bad OCR and $tuff. Gr8!
```

1. You could isolate all the capital letters (L, O, C, R, G) with this regex:

```
[A-Z]
```

2. You could isolate the first capital letter (L) with this regex:

```
^[A-Z]
```

3. You could isolate all characters BUT the capital letters with this regex:

```
[^A-Z]
```

4. You could isolate the acronym "OCR" with this regex:

```
[A-Z]{3}
```

5. You could isolate the punctuation using this regex:

```
[[:punct:]]
```

6. You could isolate all the punctuation, spaces, and numbers this way:

```
[[:punct:], ,0-9]
```

The character set is not that large, but the patterns can get complicated. Moreover, different characters can mean different things depending on their placement. Take for example, the difference between example 2 and example 3 above. In example 2, the caret (^) means isolate the pattern at the beginning of the line or document. However, when you put the caret inside the character class (demarcated by `[]`) it means "except" these sets of characters.

The best way to understand Regular Expressions is to learn what the characters do in different positions and practice, practice, practice. And since experimentation is best way to learn, I suggest using a regex tester tool and experiment with the syntax. For Mac users, I had a lot of luck with the Patterns App[276] (Mac Store $2.99), which allowed me to see what the regular expressions were doing in real time. It also comes with a built-in cheat sheet for the symbols, but I actually found this generic (meaning it works across languages) cheat sheet[277] more comprehensive.

Python and Regex

In this tutorial, I use the Regular Expressions Python module to extract a "cleaner" version of the *Congressional Directory* text file. Though the documentation[278] for this module is fairly comprehensive, beginners will have more luck with the simpler Regular Expression HOWTO documentation.[279]

Two things to note before you get started

[276] 'Patterns', *Krillapps*: http://krillapps.com/patterns/

[277] Dave Child, 'Regular Expressions Cheat Sheet', *Cheatography*: http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/

[278] 'Regular expression operations', *Python*: https://docs.python.org/2/library/re.html

[279] 'Regular Expression HOWTO', *Python*: https://docs.python.org/2/howto/regex.html#regex-howto

From what I've observed, Python is *not* the most efficient way to use Regular Expressions if you have to clean a single document. Command Line programs like sed or grep[280] appear to be more efficient for this process. (I will leave it to the better grep/sed users to create tutorials on those tools.) I use Python for several reasons: 1) I understand the syntax best; 2) I appreciate seeing each step written out in a single file so I can easily backtrack mistakes; and 3) I want a program I could use over and over again, since I am cleaning multiple pages from the *Congressional Directory*.

The OCR in this document is far from consistent (within a single page or across multiple pages). Thus, the results of this cleaning tutorial are not perfect. **My goal is to let regex do the heavy lifting and export a document in my chosen format that is *more* organized than the document with which I started.** This significantly reduces, but does not eliminate, any hand-cleaning I might need to do before geocoding the address data.

My example Python File

Here's the Python file that I used to created to clean my document:

```
#cdocr.py
#strip the punctuation and extra information from HeinOnline text document

#import re module
import re

#Open the text file with the ocr
ocr = open('../../data/txt/50-1-p1.txt')
#read the text file into a list
Text = ocr.readlines()

#Create an empty list to fill with lines of corrected text
CleanText = []

# checks each line in the imported text file for all the following patterns
for line in Text:
    #lines with multi-dashes contain data - searches for those lines
    # -- does not isolate intro text lines with one dash.
    dashes = re.search('(--+)', line)

    #isolates lines with dashes and cleans
    if dashes:
        #replaces dashes with my chosen delimiter
        nodash = re.sub('.(-+)', ',', line)
        #strikes multiple periods
        nodots = re.sub('.(\.\.+)', '', nodash)
        #strikes extra spaces
        nospaces = re.sub('(  +)', ',', nodots)
        #strikes *
        nostar = re.sub('.[*]', '', nospaces)
        #strikes new line and comma at the beginning of the line
```

[280] 'GNU sed': http://www.gnu.org/software/sed/; 'GNU Grep': http://www.gnu.org/software/grep/

```
        flushleft = re.sub('^\W', '', nostar)
        #getting rid of double commas (i.e. - Evarts)
        comma = re.sub(',{2,3}', ',', flushleft)
        #cleaning up some words that are stuck together (i.e. -  Dawes, Mand
erson)
        #skips double OO that was put in place of 00 in address
        caps = re.sub('[A-N|P-Z]{2,}', ',', comma)
        #Clean up NE and NW quadrant indicators by removing periods
        ne = re.sub('(\,*? N\. ?E.)', ' NE', caps)
        nw = re.sub('(\,*? N\. ?W[\.\,]*?_?)$', ' NW', ne) #MAKE VERBOSE
        #Replace periods with commas between last and first names (i.e. - Ch
ace, Cockrell)
        match = re.search('^([A-Z][a-z]+\. )', nw) #MAKE VERBOSE
        if match:
            names = re.sub('\.', ',', nw)
        else:
            names = nw
           #Append each line to CleanText list while it loops through
        CleanText.append(names)

#Saving into a 'fake' csv file
fcsv = open('cdocr2/50-1p1.csv', 'w')
#Write each line in CleanText to a file
for line in CleanText:
    fcsv.write(line)
```

I've commented it pretty extensively, so I will explain why I structured the code the way I did. I will also demonstrate a different way to format long regular expressions for better legibility.

Lines 16-22 – Notice in my original text file that my data is all on lines with multiple dashes. This code effectively isolates those lines. I use the `re.search()` function[281] to find all lines with multiple dashes. The "if" statement on line 20 only works with the lines with dashes in the rest of the code. (This eliminates all introductory text and the rows of page numbers that follow the data I want.)

Lines 23-40 – This is the long process by which I eliminate all of the extraneous punctuation and put the pieces of my data (last name, first name, home post office, washington address) into different fields for a csv document. I use the `re.sub()` function,[282] which substitutes pattern with another character. I comment extensively here, so you can see what each piece does. This may not be the most efficient way of doing this, but by doing this piece by piece, I could check my work as I went. As I built loop, I checked each step by printing the variable in the command line. So, for example, after line 24 (when I eliminate the dashes), I would add "print nodash" (inside the if loop) before I ran the file in the command line. I checked each step to make sure my patterns were only changing the things I wanted and not changing things I did *not* want changed.

Lines 41-46 - I used a slightly different method here. The OCR in the text file separated some names with a period (for example, Chace.Jonathan vs.

[281] 're.search', *Python Documentation*: https://docs.python.org/2/library/re.html#re.search

[282] 're.sub', *Python Documentation*: https://docs.python.org/2/library/re.html#re.sub

Chase,Jonathan). I wanted to isolate the periods that came up in this pattern and change those periods to commas. So I searched for the pattern `^([A-Z][a-z]+\.)`, which looks at the beginning of a line (^) and finds a pattern with one capital letter, multiple lowercase letters and a period. After I had isolated that pattern, I substitute the period those lines that fit the pattern with a comma.

Using Verbose Mode

Most regular expressions are difficult to read. But lines 39 and 40 look *especially* bad. How might you clarify these patterns for people who might look at your code (or for yourself when you are staring at them at 2:00 AM someday)? You can use the module's verbose mode.[283] By putting your patterns in verbose mode, python ignores white space and the # character, so you can split the patterns across multiple lines and comment each piece. ***Keep in mind that, because it ignores spaces, if spaces are part of your pattern, you need to escape them with a backslash (\). Also note that re.VERBOSE and re.X are the same thing.***

Here are lines 39 and 40 in verbose mode:

```
#This is the same as (\,*? N\. ?E.)
#All spaces need to be escaped in verbose mode.
ne_pattern = re.compile(r'''
    (           #start group
        \,*?    #look for comma (escaped); *? = 0 or more commas with fewest results
        \ N\.?  #look for (escaped) space + N that might have an (escaped) period af
ter it
        \ ?E    #look for an E that may or may not have an space in front of it
        .       #the E might be followed by another character.
    )           #close group
    $           #ONLY look at the end of a line
''', re.VERBOSE)

#This is the same as (\,*? N\. ?W[\.\,]*?_?)$
nw_pattern = re.compile(r'''
    (           #start group
        \,*?    #look for comma (escaped); *? = 0 or more commas with fewest results
        \ N\.?  #look for escaped space+N that might have an escaped period after it
        \ ?W    #look for an W that may or may not have an space in front of it
        [\.\,]*? #look for commas or periods (both escaped) that might come after W
        _?      #look for underscore that comes after NW quadrant indicators
    )           #close group
    $           #ONLY look at the end of a line
''', re.X)
```

In above example, I use the `re.compile()` function[284] to save the pattern for future use. So, adjusting my full python code to use verbose mode would look like the following. Note that I define my verbose patterns on lines 17-39 and store them in variables (ne_pattern and nw_pattern). I use them in my loop on lines 65 and 66.

```
#cdocrverbose.py
#strip the punctuation and extra information from HeinOnline text document
```

[283] 're.verbose', *Python Documentation*: https://docs.python.org/2/library/re.html#re.VERBOSE

[284] 're.compile', *Python Documentation:* https://docs.python.org/2/library/re.html#re.compile

```
#import re module
import re

#Open the text file with the ocr
ocr = open('../../data/txt/50-1-p1.txt')
#read the text file into a list
Text = ocr.readlines()

#Create an empty list to fill with lines of corrected text
CleanText = []

##Creating verbose patterns for the more complicated pieces that I use later on.##

#This is the same as (\,*? N\. ?E.)
#All spaces need to be escaped in verbose mode.
ne_pattern = re.compile(r'''
    (               #start group
        \,*?        #look for comma (escaped); *? = 0 or more commas with fewest results
        \ N\.?      #look for (escaped) space + N that might have an escaped period after it
        \ ?E        #look for an E that may or may not have an space in front of it
        .           #the E might be followed by another character.
    )               #close group
    $               #ONLY look at the end of a line
''', re.VERBOSE)

#This is the same as (\,*? N\. ?W[\.\,]*?_?)$
nw_pattern = re.compile(r'''
    (               #start group
        \,*?        #look for comma (escaped); *? = 0 or more commas with fewest results
        \ N\.?      #look for escaped space + N that might have an escaped period after it
        \ ?W        #look for an W that may or may not have an space in front of it
        [\.\,]*?    #look for commas or periods (both escaped) that might come after W
        _?          #look for underscore that comes after one of these NW quadrant indicators
    )               #close group
    $               #ONLY look at the end of a line
''', re.VERBOSE)

# checks each line in the imported text file for all the following patterns
for line in Text:
    #lines with multi-dashes contain data - searches for those lines
    # -- does not isolate intro text lines with one dash.
    dashes = re.search('(--+)', line)

    #isolates lines with dashes and cleans
    if dashes:
        #replaces dashes with my chosen delimiter
        nodash = re.sub('.(-+)', ',', line)
        #strikes multiple periods
        nodots = re.sub('.(\.\.+)', '', nodash)
        #strikes extra spaces
        nospaces = re.sub('(  +)', ',', nodots)
        #strikes *
        nostar = re.sub('.[*]', '', nospaces)
        #strikes new line and comma at the beginning of the line
        flushleft = re.sub('^\W', '', nostar)
        #getting rid of double commas (i.e. - Evarts)
        comma = re.sub(',{2,3}', ',', flushleft)
        #cleaning up some words that are stuck together (i.e. -  Dawes, Manderson)
        #skips double OO that was put in place of 00 in address
        caps = re.sub('[A-N|P-Z]{2,}', ',', comma)
        #Clean up NE and NW quadrant indicators by removing
        #periods (using Verbose regex defined above)
        ne = re.sub(ne_pattern, ' NE', caps)
        nw = re.sub(nw_pattern, ' NW', ne)
        #Replace periods with commas between last and first names
        #(i.e. - Chace, Cockrell)
        match = re.search('^([A-Z][a-z]+\.)', nw)
        if match:
            names = re.sub('\.', ',', nw)
        else:
            names = nw
         #Append each line to CleanText list while it loops through
        CleanText.append(names)

#Saving into a 'fake' csv file
fcsv = open('cdocr2/50-1p1.csv', 'w')
#Write each line in CleanText to a file
for line in CleanText:
    fcsv.write(line)
```

In conclusion, I will note that this is not for the faint of heart. Regular Expressions are powerful. Yes, they are powerful enough to completely destroy your data. So practice on copies and take it one itty bitty step at a time.

About the Author

Laura Turner O'Hara works in the Office of the Historian at the U.S. House of Representatives.

28. Generating an Ordered Data Set from an OCR Text File

Jon Crump – 2014

Lesson goals

This tutorial illustrates strategies for taking raw OCR output from a scanned text, parsing it to isolate and correct essential elements of metadata, and generating an ordered data set (a python dictionary) from it. These illustrations are specific to a particular text, but the overall strategy, and some of the individual procedures, can be adapted to organize any scanned text, even if it doesn't look like this one.

Table of Contents

Introduction

It is often the case that historians involved in digital projects wish to work with digitized texts, so they think "OK, I'll just scan this fabulously rich and useful collection of original source material and do wonderful things with the digital text that results". (Those of us who have done this, now smile ruefully). Such historians quickly discover that even the best OCR results in unacceptably high error rates. So the historian now thinks "OK I'll get some grant money, and I'll enlist the help of an army of RAs/Grad students/Undergrads/Barely literate street urchins, to correct errors in my OCR output. (We smile again, even more sadly now).

1. There is little funding for this kind of thing. Increasingly, projects in the humanities have focused upon NLP/Data Mining/Machine Learning/Graph Analysis, and the like, frequently overlooking the fundamental problem of generating useable digital texts. The presumption has often been, well, Google scanned all that stuff didn't they? What's the matter with their scans?

2. Even if you had such an army of helpers, proof-reading the OCR output of, say, a collection of twelfth century Italian charters transcribed and published in 1935, will quickly drive them all mad, make their eyes bleed, and the result will still be a great wad of text containing a great many errors, and you will **still** have to do **something** to it before it becomes useful in any context.

Going through a text file line by line and correcting OCR errors one at a time is hugely error-prone, as any proof reader will tell you. There are ways to automate some of this tedious work. A scripting language like Perl or Python can allow you to search your OCR output text for common errors and correct them using "Regular Expressions", a language for describing patterns in text. (So called because they express a "regular language".[285] See L.T. O'Hara's tutorial on Regular Expressions)[286] Regular Expressions, however, are only useful if the expressions you are searching for are ... well ... regular. Unfortunately, much of what you have in OCR output is highly *irregular*. If you could impose some order on it: create an ordered data set out of it, your Regular Expression tools would become much more powerful.

Consider, for example, what happens if your OCR interpreted a lot of strings like this "21 July, 1921" as "2l July, 192l", turning the integer '1' into an 'l'. You would love to be able to write a search and replace script that would turn all instances of 2l into 21, but then what would happen if you had lots of occurrences of strings like this in your text: "2lb. hammer". You'd get a bunch of 21b. hammers; not what you want. If only you could tell your script: only change 2l into 21 in sections where there are dates, not weights. If you had an ordered data set, you could do things like that.

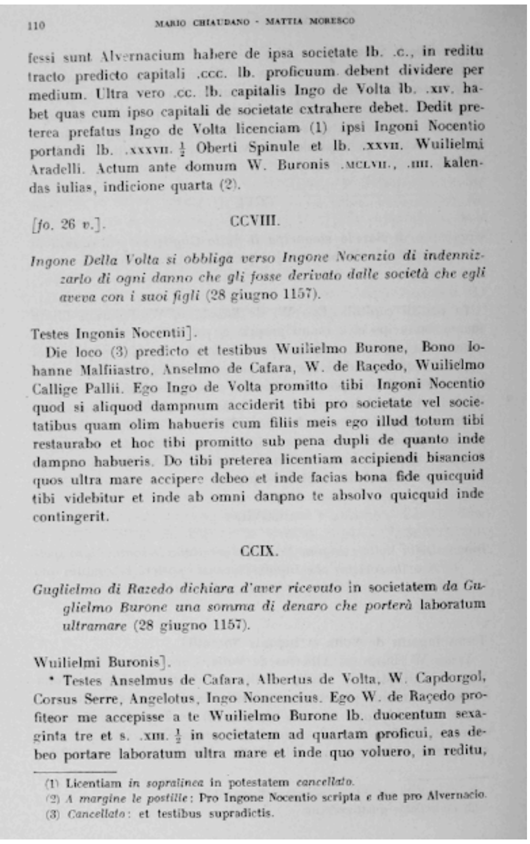
Very often the texts that historians wish to digitize are, in fact, ordered data sets: ordered collections of primary source documents, or a legal code say, or a cartulary. But the editorial structure imposed upon such resources is usually designed for a particular kind of data retrieval technology i.e., a codex, a book. For a digitized text you need a different kind of structure. If

[285] 'Regular language', *Wikipedia*: https://en.wikipedia.org/wiki/Regular_language

[286] 'Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).

you can get rid of the book related infrastructure and reorganize the text according to the sections and divisions that you're interested in, you will wind up with data that is much easier to do search and replace operations on, and as a bonus, your text will become immediately useful in a variety of other contexts as well.

This is where a scripting language like Python comes very much in handy. For our project we wanted to prepare some of the documents from a 12th century collection of *imbreviatura* from the Italian scribe known as Giovanni Scriba[287] so that they could be marked up by historians for subsequent NLP analysis or potentially for other purposes as well. The pages of the 1935 published edition look like this.



110 MARIO CHIAUDANO - MATTIA MORESCO

fessi sunt Alvernacium habere de ipsa societate lb. .c., in reditu tracto predicto capitali .ccc. lb. proficuum debent dividere per medium. Ultra vero .cc. !b. capitalis Ingo de Volta lb. .xiv. habet quas cum ipso capitali de societate extrahere debet. Dedit preterea prefatus Ingo de Volta licenciam (1) ipsi Ingoni Nocentio portandi lb. .xxxvii. ½ Oberti Spinule et lb. .xxvii. Wuilielmi Aradelli. Actum ante domum W. Buronis .mclvii., .iii. kalendas iulias, indicione quarta (2).

[fo. 26 v.]. CCVIII.

Ingone Della Volta si obbliga verso Ingone Nocenzio di indennizzarlo di ogni danno che gli fosse derivato dalle società che egli aveva con i suoi figli (28 giugno 1157).

Testes Ingonis Nocentii].

Die loco (3) predicto et testibus Wuilielmo Burone, Bono Iohanne Malfiiastro, Anselmo de Cafara, W. de Raçedo, Wuilielmo Callige Pallii. Ego Ingo de Volta promitto tibi Ingoni Nocentio quod si aliquod dampnum acciderit tibi pro societate vel societatibus quam olim habueris cum filiis meis ego illud totum tibi restaurabo et hoc tibi promitto sub pena dupli de quanto inde dampno habueris. Do tibi preterea licentiam accipiendi bisancios quos ultra mare accipere debeo et inde facias bona fide quicquid tibi videbitur et inde ab omni danpno te absolvo quicquid inde contingerit.

CCIX.

Guglielmo di Razedo dichiara d'aver ricevuto in societatem *da Guglielmo Burone una somma di denaro che porterà* laboratum *ultramare* (28 giugno 1157).

Wuilielmi Buronis].

• Testes Anselmus de Cafara, Albertus de Volta, W. Capdorgol, Corsus Serre, Angelotus, Ingo Noncencius. Ego W. de Raçedo profiteor me accepisse a te Wuilielmo Burone lb. duocentum sexaginta tre et s. .xiii. ½ in societatem ad quartam proficui, eas debeo portare laboratum ultra mare et inde quo voluero, in reditu,

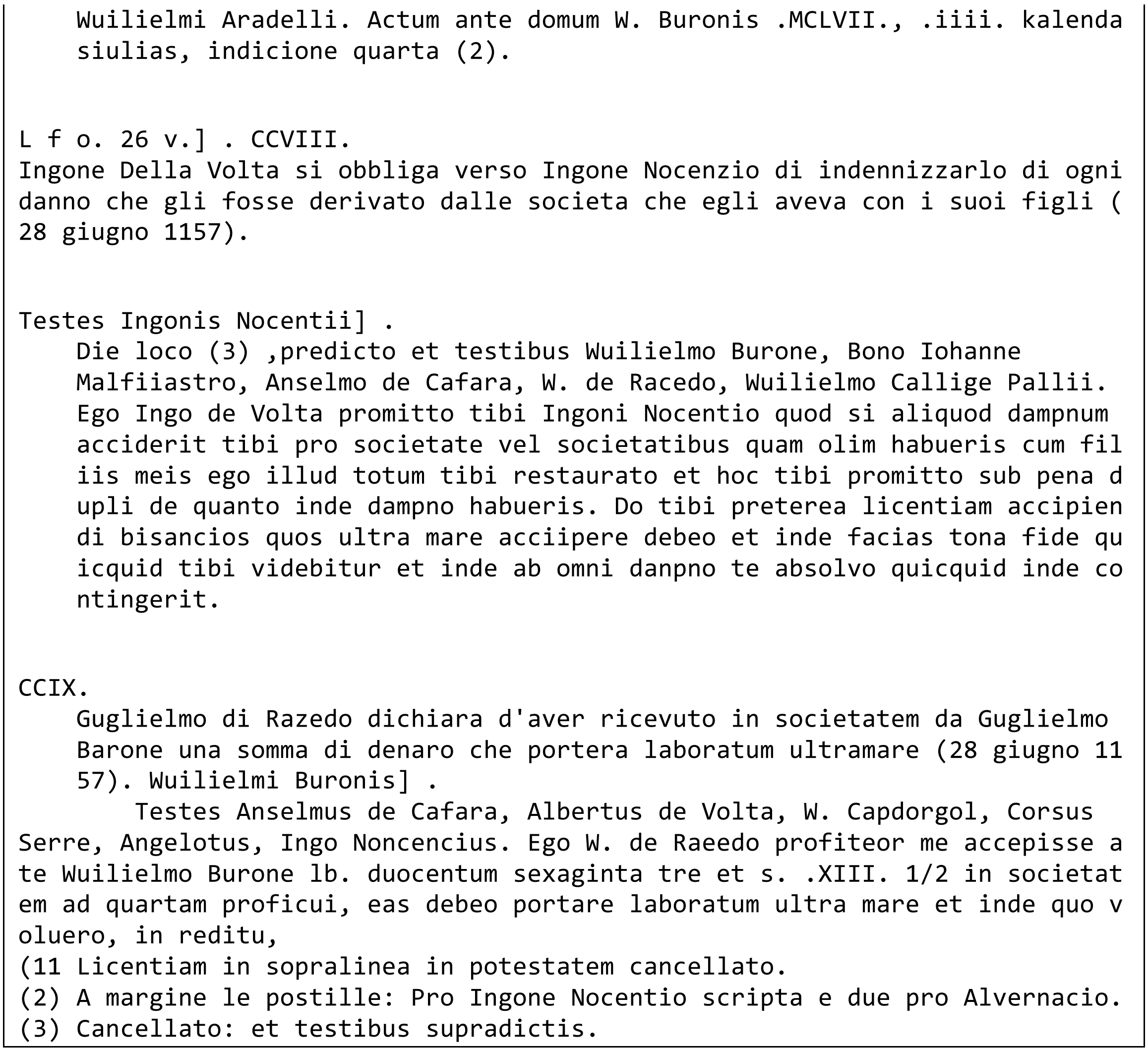
(1) Licentiam *in sopralinea* in potestatem *cancellato.*
(2) *A margine le postille*: Pro Ingone Nocentio scripta *e* due pro Alvernacio.
(3) *Cancellato*: et testibus supradictis.

GS page 110

The OCR output from such scans look like this even after some substantial clean-up (I've wrapped the longest lines so that they fit here):

```
110 MARIO CHIAUDANO MATTIA MORESCO
    professi sunt Alvernacium habere de i;psa societate lb. .c., in reditu
    tracto predicto capitali .ccc. lb. proficuum. debent dividere per medium
    . Ultra vero .cc. lb. capitalis Ingo de Volta lb. .xiv. habet quas cum I
    pso capitali de scicietate extrahere debet. Dedit preterea prefatus Ingo
    de Volta licenciam (1)
    ipsi Ingoni Nocentio portandi lb. .xxxvII. 2 Oberti Spinule et Ib. .xxvI
    I.
```

[287] Giovanni Scriba, ‘Il cartolare di Giovanni Scriba…’, *WorldCat*: http://www.worldcat.org/title/cartolare-di-giovanni-scriba/oclc/17591390



```
    Wuilielmi Aradelli. Actum ante domum W. Buronis .MCLVII., .iiii. kalenda
    siulias, indicione quarta (2).


L f o. 26 v.] . CCVIII.
Ingone Della Volta si obbliga verso Ingone Nocenzio di indennizzarlo di ogni
danno che gli fosse derivato dalle societa che egli aveva con i suoi figli (
28 giugno 1157).


Testes Ingonis Nocentii] .
    Die loco (3) ,predicto et testibus Wuilielmo Burone, Bono Iohanne
    Malfiiastro, Anselmo de Cafara, W. de Racedo, Wuilielmo Callige Pallii.
    Ego Ingo de Volta promitto tibi Ingoni Nocentio quod si aliquod dampnum
    acciderit tibi pro societate vel societatibus quam olim habueris cum fil
    iis meis ego illud totum tibi restaurato et hoc tibi promitto sub pena d
    upli de quanto inde dampno habueris. Do tibi preterea licentiam accipien
    di bisancios quos ultra mare acciipere debeo et inde facias tona fide qu
    icquid tibi videbitur et inde ab omni danpno te absolvo quicquid inde co
    ntingerit.


CCIX.
    Guglielmo di Razedo dichiara d'aver ricevuto in societatem da Guglielmo
    Barone una somma di denaro che portera laboratum ultramare (28 giugno 11
    57). Wuilielmi Buronis] .
        Testes Anselmus de Cafara, Albertus de Volta, W. Capdorgol, Corsus
Serre, Angelotus, Ingo Noncencius. Ego W. de Raeedo profiteor me accepisse a
te Wuilielmo Burone lb. duocentum sexaginta tre et s. .XIII. 1/2 in societat
em ad quartam proficui, eas debeo portare laboratum ultra mare et inde quo v
oluero, in reditu,
(11 Licentiam in sopralinea in potestatem cancellato.
(2) A margine le postille: Pro Ingone Nocentio scripta e due pro Alvernacio.
(3) Cancellato: et testibus supradictis.
```

In the scan of the original, the reader's eye readily parses the page: the layout has meaning. But as you can see, reduced to plain text like this, none of the metadata implied by the page layout and typography can be differentiated by automated processes.

You can see from the scan that each charter has the following metadata associated with it.

Charter number
Page number
Folio number
An Italian summary, ending in a date of some kind
A line, usually ending with a ']' that marks a marginal notation in the original
Frequently a collection of in-text numbered footnote markers, whose text appears at the bottom of each page, sequentially numbered, and restarting from 1 on each new page.
The Latin text of the charter itself

This is typical of such resources, though editorial conventions will vary widely. The point is: this is an **ordered** data set, not just a great big string of characters. With some fairly straightforward Python scripts, we can turn our OCR output into an ordered data set, in this case, a python dictionary,[288] **before** we start trying to proofread the Latin charter texts. With such an ordered data set in hand, we can do proofreading, and potentially many other kinds of tasks, much more effectively.

So, the aim of this tutorial is to take a plain text file, like the OCR output above and turn it into a python dictionary with fields for the Latin text of the charter and for each of the metadata elements mentioned above:

```
{
.
.
.
 52: {'chid': 'GScriba_LII',
      'chno': 52,
      'date': datetime.date(1156, 3, 27),
      'folio': '[fo. 6 r.]',
      'footnotes': [(1, 'Cancellato: m.')],
      'marginal': 'no marginal]',
      'pgno': 29,
      'summary': 'I consoli di Genova riconoscono con sentenza il diritto di
 Romano di Casella di pagarsi sui beni di Gerardo Confector per un credito c
he aveva verso il medesimo (27 marzo 1156).',
      'text': ['    In pontili capituli consules E. Aurie, W. Buronus, Ogeri
us Ventus laudaverunt quod Romanus de Casella haberet in bonis Gerardi Confe
ctoris s. .xxvi. denariorum et possit eos accipere sine contradicione eius e
t omnium pro eo. Hoc ideo quia, cum; Romanus ante ipsos inde conquereretur,
ipso Gerardo debitum non negante, sed quod de usura esset obiiciendo, iuravi
t nominatus Romanus quod capitalis erat (1) et non de usura, unde ut supra l
audaverunt , .MCLVI., sexto kalendas aprilis, indicione tercia.\n']},
 53: {'chid': 'GScriba_LIII',
      'chno': 53,
      'date': datetime.date(1156, 3, 27),
      'folio': '[fo. 6 r.]',
      'footnotes': [],
      'marginal': 'Belmusti]',
      'pgno': 29,
      'summary': "Maestro Arnaldo e Giordan nipote del fu Giovanni di Piacen
za si obbligano di pagare una somma nell'ottava della prossima Pasqua, per m
erce ricevuta (27 marzo 1156).",
      'text': ['  Testes Conradus Porcellus, Albericus, Vassallus Gambalixa,
 Petrus Artodi. Nos Arnaldus magister et Iordan nepos quondam Iohannis Place
ntie accepimus a te Belmusto tantum bracile unde promittimus dare tibi vel t
uo certo misso lb. .xLIII. denariorum usque octavam proximi pasce, quod si n
on fecerimus penam dupli tibi stipulanti promittimus, bona pignori, possis u
numquemque convenire de toto. Actum prope campanile Sancti Laurentii, milles
imo centesimo .Lv., sexto kalendas aprilis, indictione tercia.\n']},
.
.
. etc.
}
```

[288] 'Dictionaries', *Python*: https://docs.python.org/2/tutorial/datastructures.html#dictionaries

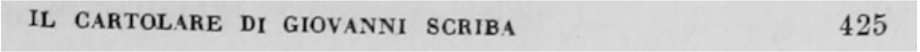
Remember, this is just a text representation of a data structure that lives in computer memory. Python calls this sort of structure a 'dictionary', other programming languages may call it a 'hash', or an 'associative array'. The point is that it is infinitely easier to do any sort of programmatic analysis or manipulation of a digital text if it is in such a form, rather than in the form of a plain text file. The advantage is that such a data structure can be queried, or calculations can be performed on the data, without first having to parse the text.

A couple of useful functions before we start:

We're going to borrow a couple of functions written by others. They both represent some pretty sophisticated programming. Understanding what's going on in these functions is instructive, but not necessary. Reading and using other people's code is how you learn programming, and is the soul of the Open-Source movement. Even if you don't fully understand how somebody does it, you can nevertheless test functions like this to see that they reliably do what they say they can, and then just apply it to your immediate problem if they are relevant.

Levenshtein distance

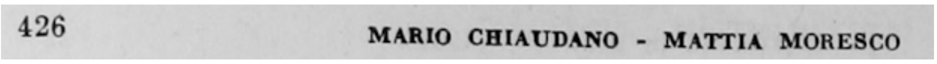
You will note that some of the metadata listed above is page-bound and some of it is charter-bound. Getting these untangled from each other is our aim. There is a class of page-bound data that is useless for our purposes, and only meaningful in the context of a physical book: page headers and footers. In our text, these look like this on *recto* leaves (in a codex, a book, *recto* is the right-side page, and *verso* its reverse, the left-side page)



IL CARTOLARE DI GIOVANNI SCRIBA 425

recto header

and this on *verso* leaves:



426 MARIO CHIAUDANO - MATTIA MORESCO

verso header

We'd like to preserve the page number information for each charter on the page, but the header text isn't useful to us and will just make any search and replace operation more difficult. So we'd like to find header text and replace it with a string that's easy to find with a Regular Expression, and store the page number.

Unfortunately, regular expressions won't help you much here. This text can appear on any line of our OCR output text, and the ways in which OCR software can foul it up are effectively limitless. Here are some examples of page headers, both *recto* and *verso* in our raw OCR output.

```
260 11141110 CH[AUDANO MATTIA MORESCO
IL CIRTOL4RE DI CIOVINN1 St'Itlltl  269
IL CJIRTOL.%RE DI G:OVeNNl FIM P%   297
IL CIP.TQLIRE DI G'OVeNNI SCI Dt    r.23
332 T1uu:0 CHIAUDANO M:11TIA MGRESCO
IL CIRTOL.'RE DI G:OV.I\N( sca:FR   339
342 NI .\ßlO CHIAUDANO 9LtTTIA MORESCO
```

These strings are not regular enough to reliably find with regular expressions; however, if you know what the strings are *supposed* to look like, you can compose some kind of string similarity algorithm to test each string against an exemplar and measure the likelihood that it is a page header. Fortunately, I didn't have to compose such an algorithm, Vladimir Levenshtein did it for us in 1965 (see: http://en.wikipedia.org/wiki/Levenshtein_distance). A computer language can encode this algorithm in any number of ways; here's an effective Python function that will work for us:

```
def lev(seq1, seq2):
    """ Return Levenshtein distance metric
    (ripped from http://pydoc.net/Python/Whoosh/2.3.2/whoosh.support.levensh
tein/)
     """
    oneago = None
    thisrow = range(1, len(seq2) + 1) + [0]
    for x in xrange(len(seq1)):
        twoago, oneago, thisrow = oneago, thisrow, [0] * len(seq2) + [x + 1]

        for y in xrange(len(seq2)):
            delcost = oneago[y] + 1
            addcost = thisrow[y - 1] + 1
            subcost = oneago[y - 1] + (seq1[x] != seq2[y])
            thisrow[y] = min(delcost, addcost, subcost)
            # This block deals with transpositions
            if (x > 0 and y > 0 and seq1[x] == seq2[y - 1]
                and seq1[x-1] == seq2[y] and seq1[x] != seq2[y]):
                thisrow[y] = min(thisrow[y], twoago[y - 2] + 1)
    return thisrow[len(seq2) - 1]
```

Again, this is some pretty sophisticated programming, but for our purposes all we need to know is that the `lev()` function takes two strings as parameters and returns a number that indicates the 'string distance' between them, or, how many changes had to be made to turn the first string into the second. So: `lev("fizz", "buzz")` returns '2'

Roman to Arabic numerals

You'll also note that in the published edition, the charters are numbered with roman numerals. Converting roman numerals into arabic is an instructive puzzle to work out in Python. Here's the cleanest and most elegant solution I know:

```
def rom2ar(rom):
    """ From the Python tutor mailing list:
    János Juhász janos.juhasz at VELUX.com
    returns arabic equivalent of a Roman numeral """
    roman_codec = {'M':1000, 'D':500, 'C':100, 'L':50, 'X':10, 'V':5, 'I':1}
    roman = rom.upper()
    roman = list(roman)
    roman.reverse()
    decimal = [roman_codec[ch] for ch in roman]
    result = 0

    while len(decimal):
        act = decimal.pop()
        if len(decimal) and act < max(decimal):
            act = -act
        result += act

    return result
```

(run <this little script>[289] to see in detail how `rome2ar` works. Elegant programming like this can offer insight; like poetry.)

Some other things we'll need:

At the top of your Python module, you're going to want to import some python modules that are a part of the standard library. (see Fred Gibbs's tutorial 'Installing Python Modules with pip').[290]

1. First among these is the "re" (regular expression) module `import re`. Regular expressions are your friends. However, bear in mind Jamie Zawinski's quip:

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

   (Again, have a look at L.T. O'Hara's introduction here at the Programming Historian 'Cleaning OCR'd text with Regular Expressions')[291]

2. Also: `from pprint import pprint`. `pprint` is just a pretty-printer for python objects like lists and dictionaries. You'll want it because python dictionaries are much easier to read if they are formatted.

3. And: `from collections import Counter`. We'll want this for the Find and normalize footnote markers and texts section below. This is not really necessary, but we'll do some counting that would require a lot of lines of fiddly code and this will save us the trouble. The collections module has lots of deep magic in it and is well worth getting familiar with. (Again, see Doug Hellmann's PyMOTW for the 'collections'

[289] Available at: http://programminghistorian.org/assets/Roman_to_Arabic.txt

[290] Fred Gibbs, 'Installing Python Modules with pip', *The Programming Historian*, (2013).

[291] Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).

module.[292] I should also point out that his book *The Python Standard Library By Example*[293] is one well worth having.)

A very brief review of regular expressions as they are implemented in python

L.T. O'Hara's introduction to using python flavored regular expressions is invaluable.[294] In this context we should review a couple of basic facts about Python's implementation of regular expressions, the `re` module, which is part of Python's standard library.

`re.compile()` creates a regular expression object that has a number of methods. You should be familiar with `.match()`, and `.search()`, but also `.findall()` and `.finditer()`
Bear in mind the difference between `.match()` and `.search()`: `.match()` will only match at the **`beginning`** of a line, whereas `.search()` will match anywhere in the line **`but then it stops`**, it'll **`only`** return the first match it finds.
`.match()` and `.search()` return match objects. To retrieve the matched string you need `mymatch.group(0)`. If your compiled regular expression has grouping parentheses in it (like our 'slug' regex below), you can retrieve those substrings of the matched string using `mymatch.group(1)` etc.
`.findall()` and `.finditer()` will return **`all`** occurrences of the matched string; `.findall()` returns them as a list of strings, but .finditer() returns an **`iterator of match objects`**. (read the docs on the method `.finditer()`).[295]

Iterative processing of text files

We'll start with a single file of OCR output. We will iteratively generate new, corrected versions of this file by using it as input for our python scripts. Sometimes our script will make corrections automatically, more often, our scripts will simply alert us to where problems lie in the input file, and we will make corrections manually. So, for the first several operations we're going to want to produce new and revised text files to use as input for our subsequent operations. Every time you produce a text file, you should version it and duplicate it so that you can always return to it. The next time you run your code (as you're developing it) you might alter the file in an unhelpful way and it's easiest just to restore the old version.

[292] 'Collections – Container data types': https://pymotw.com/2/collections/index.html#module-collections

[293] Doug Hellmann, *The Python Standard Library by Example*: https://doughellmann.com/blog/the-python-standard-library-by-example/

[294] Laura Turner O'Hara, 'Cleaning OCR'd text with Regular Expressions', *The Programming Historian* (2013).

[295] 're.finditer', *Python*: https://docs.python.org/2/library/re.html#re.finditer

The code in this tutorial is highly edited; it is **not** comprehensive. As you continue to refine your input files, you will write lots of little *ad hoc* scripts to check on the efficacy of what you've done so far. Versioning will ensure that such experimentation will not destroy any progress that you've made.

A note on how to deploy the code in this tutorial:

The code in this tutorial is for Python 2.7.x, Python 3 is quite a different animal.

When you write code in a text file and then execute it, either at the command line, or from within your text editor or IDE, the Python interpreter executes the code line by line, from top to bottom. So, often the code on the bottom of the page will depend on code above it.

One way to use the code snippets in section 2 might be to have all of them in a single file and comment out the bits that you don't want to run. Each time you execute the file, you will want to be sure that there is a logical control flow from the `#!` line at the top, through your various `import`s and assignment of global variables, and each loop, or block.

Or, each of the subsections in section 2 can also be treated as a separate script, each would then have to do its own `import`ing and assignment of global variables.

In section 3, "Creating the Dictionary", you will be operating on a data set in computer memory (the `charters` dictionary) that will be generated from the latest, most correct, input text you have. So you will want to maintain a single python module in which you define the dictionary at the top, along with your `import` statements and the assignment of global variables, followed by each of the four loops that will populate and then modify that dictionary.

```
#!/usr/bin/python

import re
from pprint import pprint
from collections import Counter

# followed by any global variables you will need, like:

n = 0
this_folio  = '[fo. 1 r.]'
this_page = 1

# compiled regular expressions like:
slug = re.compile("(\[~~~~\sGScriba_)(.*)\s::::\s(\d+)\s~~~~\]")
fol = re.compile("\[fo\.\s?\d+\s?[rv]\.\s?\]")
pgbrk = re.compile("~~~~~ PAGE (\d+) ~~~~~")

# the canonical file you will be reading from
fin = open("/path/to/your/current/canonical.txt", 'r')
```

```
GScriba = fin.readlines()


# then the empty dictionary:
charters = dict()

# followed by the 4 'for' loops in section 2 that will populate and then mod
ify this dictionary
```

Chunk up the text by pages

First of all, we want to find all the page headers, both *recto* and *verso* and replace them with consistent strings that we can easily find with a regular expression. The following code looks for lines that are similar to what we know are our page headers to within a certain threshold. It will take some experimentation to find what this threshold is for your text. Since my *recto* and *verso* headers are roughly the same length, both have the same similarity score of 26.

NOTA BENE: The *lev()* function described above returns a measure of the 'distance' between two strings, so, the shorter the page header string, the more likely it is that this trick will not work. If your page header is just "Header", then any line comprised of a six letter word might give you a string distance of 6, eg: *lev("Header", "Foobar")* returns '6', leaving you none the wiser. In our text, however, the header strings are long and complex enough to give you meaningful scores, eg:

```
lev("RANDOM STRING OF SIMILAR LENGTH:    38", 'IL CARTOLARE DI GIOVANNI
SCRIBA')
```

returns 33, but one of our header strings, even badly mangled by the OCR, returns 20:

```
lev("IL CIRTOL4RE DI CIOVINN1 St'Itlltl     269", 'IL CARTOLARE DI GIOVANNI
SCRIBA')
```

So we can use `lev()` to find and modify our header strings thus:

```
# At the top, do the importing you need and define the lev() function as des
cribed above, and then:

fin = open("our_base_OCR_result.txt", 'r') # read our OCR output text
fout = open("out1.txt", 'w') # create a new textfile to write to when we're
ready
GScriba = fin.readlines() # turn our input file into a list of lines

for line in GScriba:
    # get a Levenshtein distance score for each line in the text
    recto_lev_score = lev(line, 'IL CARTOLARE DI GIOVANNI SCRIBA')
    verso_lev_score = lev(line, 'MARIO CHIAUDANO - MATTIA MORESCO')

    # you want to use a score that's as high as possible,
    # but still finds only potential page header texts.
    if recto_lev_score < 26 :

        # If we increment a variable 'n' to
       #count the number of headers we've found,
        # then the value of that variable should be our page number.
        n += 1
        print "recto: %s %s" % (recto_lev_score, line)

        # Once we've figured out our optimal 'lev' score, we can 'uncomment'
        # all these `fout.write()` lines to write out our new text file,
        # replacing each header with an easy-to-find string that contains
        # the page number: our variable 'n'.

        #fout.write("~~~~~ PAGE %d ~~~~~\n\n" % n)
    elif verso_lev_score < 26 :
        n += 1
        print "verso: %s %s" % (verso_lev_score, line)
        #fout.write("~~~~~ PAGE %d ~~~~~\n\n" % n)
    else:
        #fout.write(line)
        pass

print n
```

There's a lot of calculation going on in the `lev()` function. It isn't very efficient to call it on every line in our text, so this might take some time, depending on how long our text is. We've only got 803 charters in vol. 1. That's a pretty small number. If it takes 30 seconds, or even a minute, to run our script, so be it.

If we run this script on our OCR output text, we get output that looks like this:

```
.
.
.
verso: 8 426    MARIO CHIAUDANO MAITIA MORESCO
recto: 5 IL CARTOLARE DI GIOVANNI SCRIBA    427
verso: 11 , ,   428 MARIO CHIAUDANO MATTIA MORESCO
recto: 5 IL CARTOLARE DI GIOVANNI SCRIBA    499
verso: 7 430    MARIO CHIAUDANO MATTIA MORESCO
recto: 5 IL CARTOLARE DI GIOVANNI SCRIBA    431
verso: 8 432    MARIO CHIAUDASO MATTIA MORESCO
430
```

For each line, the output tells us that it's page *verso* or *recto*, the Levenshtein "score", and then the text of the line (complete with all the errors in it. Note that the OCR misread the pg. number for pg. 429). The lower the Levenshtein "score", the closer the line is to the model you've given it.

This tells you that the script found 430 lines that are probably page headers. You know how many pages there should be, so if the script didn't find all the headers, you can go through the output looking at the page numbers, find the pages it missed, and fix the headers manually, then repeat until the script finds all the page headers.

Once you've found and fixed the headers that the script didn't find, you can then write out the corrected text to a new file that will serve as the basis for the other operations below. So, instead of

```
quicquid volueris sine omni mea et
(1) Spazio bianco nel ms.

12  MARIO CSIAUDANO MATTIA MORESCO
heredum meorum contradicione. Actum in capitulo .MCLV., mensis iulii, indici
one secunda.
```

we'll have a textfile like this:

```
quicquid volueris sine omni mea et
(1) Spazio bianco nel ms.

~~~~~ PAGE 12 ~~~~~

heredum meorum contradicione. Actum in capitulo .MCLV., mensis iulii, indici
one secunda.
```

Note that for many of the following operations, we will use `GScriba = fin.readlines()` so `GScriba` will be a **`python list`** of the lines in our input text. Keep this firmly in mind, as the `for` loops that we will use will depend on the fact that we will iterate through the lines of our text **In Document Order**.

Chunk up the text by charter (or sections, or letters, or what-have-you)

The most important functional divisions in our text are signaled by upper case roman numerals on a separate line for each of the charters. So we need a regex to find roman numerals like that. Here's one: `romstr = re.compile("\s*[IVXLCDM]{2,}")`. We'll put it at the top of our module file as a 'global' variable so it will be available to any of the bits of code that come later.

The script below will look for capital roman numerals that appear on a line by themselves. Many of our charter numbers will fail that test and the script will report `there's a charter roman numeral missing?`, often because there's something before or after it on the line; or, `KeyError`, often because the OCR has garbled the characters (e.g. CCG for 300, XOII for 492). Run this script repeatedly, correcting `out1.txt` as you do until all the charters are accounted for.

```
# At the top, do the importing you need, then define rom2ar()
#as described above, and then:

n = 0
romstr = re.compile("\s*[IVXLCDM]{2,}")
fin = open("out1.txt", 'r')
fout = open("out2.txt", 'w')
GScriba = fin.readlines()

for line in GScriba:
    if romstr.match(line):
        rnum = line.strip().strip('.')
        # each time we find a roman numeral by itself
```

```
         #on a line we increment n:
        # that's our charter number.
        n += 1
        try:
            # translate the roman to the arabic and it should be equal to n.
            if n != rom2ar(rnum):
                # if it's not, then alert us
                print "%d, there's a charter roman numeral missing?, because line nu
mber %d reads: %s" % (n, GScriba.index(line), line)


                # then set 'n' to the right number
                n = rom2ar(rnum)
        except KeyError:
            print n, "KeyError, line number ", GScriba.index(line), " reads: ", line
```

Since we know how many charters there should be. At the end of our loop, the value of n should be the same as the number of charters. And, in any iteration of the loop, if the value of n does not correspond to the next successive charter number, then we know we've got a problem somewhere, and the print statements should help us find it.

Here's a sample of the output our script will give us:

```
23 there's a charter roman numeral missing?, because line number  156  reads:  XXIV.
25 there's a charter roman numeral missing?, because line number  186  reads:  XXVII
I.
36 KeyError, line number  235  reads:  XXXV1.
37 KeyError, line number  239  reads:  XXXV II.
38 there's a charter roman numeral missing?, because line number  252  reads:  XL.
41 there's a charter roman numeral missing?, because line number  262  reads:  XLII.
43 KeyError, line number  265  reads:  XL:III.
```

NOTA BENE: Our regex will report an error for the single digit Roman numerals ('I','V','X' etc.). You could test for these in the code, but sometimes leaving a known and regular error is a help to check on the efficacy of what you're doing. Our aim is to satisfy ourselves that any inconsistencies on the charter number line are understood and accounted for.

Once we've found, and fixed, all the roman numeral charter headings, then we can write out a new file with an easy-to-find-by-regex string, a 'slug,' for each charter in place of the bare roman numeral. Comment out the `for` loop above, and replace it with this one:

```
for line in GScriba:
    if romstr.match(line):
        rnum = line.strip().strip('.')
        num = rom2ar(rnum)
        fout.write("[~~~~ GScriba_%s :::: %d ~~~~]\n" % (rnum, num))
    else:
        fout.write(line)
```

While it's important in itself for us to have our OCR output reliably divided up by page and by charter, the most important thing about these initial operations is that you know how many pages there are, and how many charters there are, and you can use that knowledge to check on subsequent operations. If you want to do something to every charter, you can reliably test whether or not it worked because you can count the number of charters that it worked on.

Find and normalize folio markers

Our OCR'd text is from the 1935 published edition of *Giovanni Scriba*. This is a transcription of a manuscript cartulary which was in the form of a bound book. The published edition preserves the pagination of that original by noting where the original pages change: [fo. 16 r.] the face side of the 16th leaf in the book, followed by its reverse [fo. 16 v.]. This is metadata that we want to preserve for each of the charters so that they can be referenced with respect to the original, as well as with respect to the published edition by page number.

Many of the folio markers (e.g. "[fo. 16 v.]") appear on the same line as the roman numeral for the charter heading. To normalize those charter headings for the operation above, we had to put a line break between the folio marker and the charter number, so many of the folio markers are on their own line already. However, sometimes the folio changes in the middle of the charter text somewhere. We want these markers to stay where they are; we will have to treat those two cases differently. For either case, we need to make sure all the folio markers are free of errors so that we can reliably find them by means of a regular expression. Again, since we know how many folios there are, we can know if we've found them all. Note that because we used `.readlines()`, `GScriba` is a list, so the script below will print the line number from the source file as well as the line itself. This will report all the correctly formated folio markers, so that you can find and fix the ones that are broken.

```
# note the optional quantifiers '\s?'. We want to find as many as we can, and
# the OCR is erratic about whitespace, so our regex is permissive. But as
# you find and correct these strings, you will want to make them consistent.
fol = re.compile("\[fo\.\s?\d+\s?[rv]\.\s?\]")

for line in GScriba:
    if fol.match(line):
        # since GScriba is a list, we can get the index of any of its members to fin
d the line number in our input file.
        print GScriba.index(line), line
```

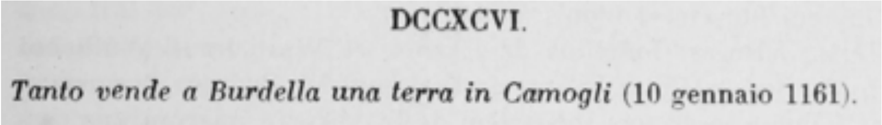
We would also like to ensure that no line has more than one folio marker. We can test that like this:

```
for line in GScriba:
    all = fol.findall(line)
    if len(all) > 1:
        print GScriba.index(line), line
```

Again, as before, once you've found and corrected all the folio markers in your input file, save it with a new name and use it as the input to the next section.

Find and normalize the Italian summary lines.

This important line is invariably the first one after the charter heading.



DCCXCVI.

Tanto vende a Burdella una terra in Camogli (10 gennaio 1161).

italian summary line

Since those roman numeral headings are now reliably findable with our 'slug' regex, we can now isolate the line that appears immediately after it. We also know that the summaries always end with some kind of parenthesized date expression. So, we can compose a regular expression to find the slug and the line following:

```
slug_and_firstline = re.compile("(\[~~~~\sGScriba_)(.*)\s::::\s(\d+)\s~~~~\]
\n(.*)(\(\d?.*\d+\))")
```

Let's break down that regex using the verbose mode (again, see O'Hara's tutorial in this book). Our 'slug' for each charter takes the form "[~~~~ GScriba_CCVII :::: 207 ~~]" for example. The compiled pattern above is exactly equivalent to the folowing (note the re.VERBOSE switch at the end):

```
slug_and_firstline = re.compile(r"""
    (\[~~~~\sGScriba_)  # matches the "[~~~~ GScriba_" bit
    (.*)                # matches the charter's roman numeral
    \s::::\s            # matches the " :::: " bit
    (\d+)               # matches the arabic charter number
    \s~~~~\]\n          # matches the last " ~~~~ " bit and the line ending
    (.*)                # matches all of the next line up to:
    (\(\d?.*\d+\))      # the paranthetical expression at the end
    """, re.VERBOSE)
```

the parentheses mark match groups, so each time our regex finds a match, we can refer in our code to specific bits of the match it found:

`match.group(0)` is the whole match, both our slug and the line that follows it.
`match.group(1)` = "[~~ GScriba_"
`match.group(2)` = the charter's roman numeral
`match.group(3)` = the arabic charter number
`match.group(4)` = the whole of the Italian summary line up to the parenthesized date expression
`match.group(5)` = the parenthesized date expression. Note the escaped parentheses.

Because our OCR has a lot of mysterious whitespace (OCR software is not good at parsing whitespace and you're likely to get newlines, tabs, spaces, all mixed up without rhyme or reason), we want to hunt for this regex as substrings of a great big string, so this time we're going to use `.read()`

instead of `.readlines()`. And we'll also need a counter to keep track of the lines we find. This script will report the charter numbers where the first line does not conform to our regex model. This will usually happen if there's no line break after our charter header, or if the Italian summary line has been broken up into multiple lines.

```
num_firstlines = 0
fin = open("your_current_source_file.txt", 'r')


# NB: GScriba is not a list of lines this time, but a single big string.
GScriba = fin.read()

# finditer() creates an iterator 'i' that we can do a 'for' loop over.
i = slug_and_firstline.finditer(GScriba)

# each element 'x' in that iterator is a regex match object.
for x in i:
    # count the summary lines we find. Remember, we know how many
    # there should be, because we know how many charters there are.
    num_firstlines += 1

    chno = int(x.group(3)) # our charter number is a string, we need an integer

    # chno should equal n + 1, if it doesn't, report to us
    if chno != n + 1:
        print "problem in charter: %d" % (n + 1)
       #NB: this will miss consecutive problems.

    # then set n to the right charter number
    n = chno

# print out the number of summary lines we found
print "number of italian summaries: ", num_firstlines
```

Again, run the script repeatedly until all the Italian Summary lines are present and correct, then save your input file with a new name and use it the input file for the next bit:

Find and normalize footnote markers and texts

One of the trickiest bits to untangle, is the infuriating editorial convention of restarting the footnote numbering with each new page. This makes it hard to associate a footnote text (page-bound data), with a footnote marker (charter-bound data). Before we can do that we have to ensure that each footnote text that appears at the bottom of the page, appears in our source file on its own separate line with no leading white-space. And that **none** of the footnote markers within the text appears at the beginning of a line. And we must ensure that every footnote string, "(1)" for example, appears **exactly** twice on a page: once as an in-text marker, and once at the bottom for the footnote text. The following script reports the page number of any page that fails that test, along with a list of the footnote strings it found on that page.

```
# Don't forget to import the Counter module:
from collections import Counter
fin = open("your_current_source_file.txt", 'r')
GScriba = fin.readlines() # GScriba is a list again
r = re.compile("\(\d{1,2}\)")
# there's lots of ways for OCR to screw this up, so be alert.


pg = re.compile("~~~~~ PAGE \d+ ~~~~~")
pgno = 0

pgfnlist = []
# remember, we're processing lines in document order. So for each page
# we'll populate a temporary container, 'pgfnlist', with values. Then
# when we come to a new page, we'll report what those values are and
# then reset our container to the empty list.

for line in GScriba:
    if pg.match(line):
        # if this test is True, then we're starting a new page,
        #so increment pgno

        pgno += 1

        # if we've started a new page,
       #then test our list of footnote markers
        if pgfnlist:
            c = Counter(pgfnlist)

            # if there are fn markers that do not appear exactly twice,
            # then report the page number to us
            if 1 in c.values(): print pgno, pgfnlist

            # then reset our list to empty
            pgfnlist = []

    # for each line, look for ALL occurences of our footnote marker regex
    i = r.finditer(line)


    for mark in [eval(x.group(0)) for x in i]:
        # and add them to our list for this page
        pgfnlist.append(mark)
```

Note: the elements in the iterator 'i' are string matches. We want the strings that were matched, group(0). e.g. "(1)". And if we do eval("(1)") we get an integer that we can add to our list.

Our `Counter` is a very handy special data structure. We know that we want each value in our `pgfnlist` to appear twice. Our `Counter` will give us a hash where the keys are the elements that appear, and the values are how many times each element appears. Like this:

```
>>> l = [1,2,3,1,3]
>>> c = Counter(l)
>>> print c
Counter({1: 2, 3: 2, 2: 1})
```

So if for a given page we get a list of footnote markers like this `[1,2,3,1,3]`, then the test `if 1 in c.values()` will indicate a problem because we know each element must appear **`exactly twice`**:

```
>>> l = [1,2,3,1,3]
>>> c = Counter(l)
>>> print c.values()
[2, 1, 2]
```

whereas, if our footnote marker list for the page is complete `[1,2,3,1,2,3]`, then:

```
>>> l = [1,2,3,1,2,3]
>>> c = Counter(l)
>>> print c.values()
[2, 2, 2] # i.e. 1 is not in c.values()
```

As before, run this script repeatedly, correcting your input file manually as you discover errors, until you are satisfied that all footnotes are present and correct for each page. Then save your corrected input file with a new name.

Our text file still has lots of OCR errors in it, but we have now gone through it and found and corrected all the specific metadata bits that we want in our ordered data set. Now we can use our corrected text file to build a Python dictionary.

Creating the Dictionary

Now that we've cleaned up enough of the OCR that we can successfully differentiate the component parts of the page from each other, we can now sort the various bits of the meta-data, and the charter text itself, into their own separate fields of a Python dictionary.

We have a number of things to do: correctly number each charter as to charter number, folio, and page; separate out the Italian summary and the marginal notation lines; and associate the footnote texts with their appropriate charter. To do all this, sometimes it is convenient to make more than one pass.

Create a skeleton dictionary.

We'll start by generating a python dictionary whose keys are the charter numbers, and whose values are a nested dictionary that has fields for some of the metadata we want to store for each charter. So it will have the form:

```
charters = {
    .
    .
    .
    300: {
            'chid': "our charter ID",
            'chno': 300,
            'footnotes': [], # an empty list for now
            'folio': "the folio marker for this charter",
            'pgno': "the page number in the printed edition for this charter
",
            'text': [] # an empty list for now
          },
    301: {
            'chid': "our charter ID",
            'chno': 301,
            'footnotes': [], # an empty list for now
            'folio': "the folio marker for this charter",
            'pgno': "the page number in the printed edition for this charter
",
            'text': [] # an empty list for now
          },
    .
    .
    . etc.
}
```

For this first pass, we'll just create this basic structure and then in subsequent loops we will add to and modify this dictionary until we get a dictionary for each charter, and fields for all the metadata for each charter. Once this loop disposes of the easily searched lines (folio, page, and charter headers) and creates an empty container for footnotes, the fall-through default will be to append the remaining lines to the text field, which is a python list.

```
slug = re.compile("(\[~~~~sGScriba_)(.*)\s::::\s(\d+)\s~~~~\]")
fol = re.compile("\[fo\.\s?\d+\s?[rv]\.\s?\]")
pgbrk = re.compile("~~~~~ PAGE (\d+) ~~~~~")

fin = open('your_current_source_file.txt', 'r')
GScriba = fin.readlines()

# we will also need these global variables with starting values
n = 0
this_folio = '[fo. 1 r.]'
this_page = 1

# ‘charters’ is also defined as a global variable. The ‘for’ loop below
# and in the following sections, will build on and modify this dictionary
charters = dict()

for line in GScriba:
       if fol.match(line):
              # use this global variable to keep track of the folio number.
              # we’ll create the ‘folio’ field using the value of this variable
              this_folio = fol.match(line).group(0)
              continue #update the variable but otherwise do nothing
       if slug.match(line):
              # if our ‘slug’ regex matches, we know we have a new charter
              # so get the data from the match groups
```

```
                m = slug.match(line)
                child = 'GScroba_' + m.group(2)
                chnm = int(m.group(3))

                # then create an empty nested dictionary
                charters[chno] = {}

                # and an empty container for the lines we won’t use this pass
                templist = []
        # this works because we are proceeding in a document order
        # templist continues to exist as we iterate through each line
        # in the charter, then is reset to the empty list when we
        # start a new charter(slug.match(line)).
        continue # we generate the entry, but do nothing with the text

if chno:
        # if a charter dictionary has been created
        # then we can now populate it with data from our slug.match above
                d = charters[chno]
                d['footnotes'] = []
                d['child'] = child
                d['chno'] = chno
                d['folio'] = this_folio
                d['pgno'] = this_page

                if re.match('^\(\d+\)’, line:
                        # this line is footnote text, because it has a footnote marker
                        # line '(1)' at the beginning. So we’ll deal with it later
                        continue
                elif fol.search(line):
                        # if folio changes with the charter text, update the variable
                        this_folio = fol.search(line).group(0)
                        templist.append(line)
                else:
                        # any line not otherwise accounted for,
                        #add to temporary container
                        templist.append(line)
                # add the temporary container to the dictionary after using
                # a list comprehension to strip  out any empty lines.
                d['text'] = [x for x in templist if not x == '\n']
```

Add the 'marginal notation' and Italian summary lines to the dictionary

When we generated the dictionary of dictionaries above, we assigned fields for footnotes (just an empty list for now), charterID, charter number, the folio, and the page number. All remaining lines were appended to a list and assigned to the field 'text'. In all cases, the first line of each charter's text field should be the Italian summary as we have insured above. The second line in MOST cases, represents a kind of marginal notation usually ended by the ']' character (which OCR misreads a lot). We have to find the cases that do not meet this criterion, supply or correct the missing ']', and in the cases where there is no marginal notation I've supplied "no marginal]" in my working text. The following diagnostic script will print the charter number and first two lines of the text field for those charters that do not meet these criteria. Run this script separately against the `charters` dictionary, and correct and update your canonical text accordingly.

```
n = 0
for ch in charters:
    txt = charters[ch]['text'] # remember: the text field is a python list o
f strings
    try:
        line1 = txt[0]
        line2 = txt[1]
        if line2 and ']' not in line2:
            n += 1
            print "charter: %d\ntext, line 1: %s\ntext, line 2: %s" % (ch,
line1, line2)
    except:
        print ch, "oops" # to pass the charters from the missing page 214
```

Note: The try: except: blocks are made necessary by the fact that in my OCR output, the data for pg 214 somehow got missed out. This often happens. Scanning or photographing each page of a 600 page book is tedious in the extreme. It's very easy to skip a page. You will inevitably have anomalies like this in your text that you will have to isolate and work around. The Python try: except: pattern makes this easy. Python is also very helpful here in that you can do a lot more in the except: clause beyond just printing "oops". You could call a function that performs a whole separate operation on those anomalous bits.

Once we're satisfied that line 1 and line 2 in the 'text' field for each charter in the `charters` dictionary are the Italian summary and the marginal notation respectively, we can make another iteration of the `charters` dictionary, removing those lines from the text field and creating new fields in the charter entry for them.

NOTA BENE: we are now modifying a data structure in memory rather than editing successive text files. So this script should be **`added`** *to the one above that created your skeleton dictionary. That script creates the charters dictionary in memory, and this one modifies it*

```
for ch in charters:
    d = charters[ch]
    try:
        d['summary'] = d['text'].pop(0).strip()
        d['marginal'] = d['text'].pop(0).strip()
    except IndexError: # this will report that the charters on p 214 are mi
ssing
        print "missing charter ", ch
```

Assign footnotes to their respective charters and add to dictionary

The trickiest part is to get the footnote texts appearing at the bottom of the page associated with their appropriate charters. Since we are, perforce, analyzing our text line by line, we're faced with the problem of associating a given footnote reference with its appropriate footnote text when there are perhaps many lines intervening.

For this we go back to the same list of lines that we built the dictionary from. We're depending on all the footnote markers appearing within the charter text, i.e. none of them are at the beginning of a line. And, each of the footnote texts is on a separate line beginning with '(1)' etc. We design regexes that can distinguish between the two and construct a container to hold them as we iterate over the lines. As we iterate over the lines of the text file, we find and assign markers and texts to our temporary container, and then, each time we reach a page break, we assign them to their appropriate fields in our existing Python dictionary `charters` and reset our temporary container to the empty `dict`.

Note how we construct that temporary container. `fndict` starts out as an empty dictionary. As we iterate through the lines of our input text, if we find footnote markers within the line, we create an entry in `fndict` whose key is the footnote number, and whose value is another dictionary. In that dictionary we record the id of the charter that the footnote belongs to, and we create an empty field for the footnote text. When we find the footnote texts (`ntexts`) at the bottom of the page, we look up the footnote number in our container `fndict` and write the text of the line to the empty field we made. So when we come to the end of the page, we have a dictionary of footnotes that looks like this:

```
{1: {'chid': 158, 'fntext': 'Nel ms. de due volte e ripa cancellato.'},
 2: {'chid': 158, 'fntext': 'Sic nel ms.'},
 3: {'chid': 159, 'fntext': 'genero cancellato nel ms.'}}
```

Now we have all the necessary information to assign the footnotes to the empty 'footnotes' list in the `charters` dictionary: the number of the footnote (the key), the charter it belongs to (chid), and the text of the footnote (fntext).

This is a common pattern in programming, and very useful: in an iterative process of some kind, you use an accumulator (our `fndict`) to gather bits of data, then when your sentinel encounters a specified condition (the pagebreak) it does something with the data.

```
fin = open("your_current_source_file.txt", 'r')
GScriba = fin.readlines()

# in notemark, note the 'lookbehind' expression '?<!' to insure that
# the marker '(1)' does not begin the string
notemark = re.compile(r"\(\d+\)(?<!^\(\d+\))")
notetext = re.compile(r"^\(\d+\)")
this_charter = 1
pg = re.compile("~~~~~ PAGE \d+ ~~~~~")
pgno = 1
fndict = {}

for line in GScriba:
    nmarkers = notemark.findall(line)
    ntexts = notetext.findall(line)
    if pg.match(line):
        # This is our 'sentinel'. We've come to the end of a page,
        # so we record our accumulated footnote data in the 'charters' dict.
        for fn in fndict:
```

```
            chid = fndict[fn]['chid']
            fntext = fndict[fn]['fntext']
            charters[int(chid)]['footnotes'].append((fn, fntext))
        pgno += 1
        fndict = {}  # and then re-initialize our temporary container
    if slug.match(line): # here's the beginning of a charter, so update the
variable.
        this_charter = int(slug.match(line).group(3))
    if nmarkers:
        for marker in [eval(x) for x in nmarkers]:
            # create an entry with the charter's id and an empty text field
            fndict[marker] = {'chid':this_charter, 'fntext': ''}
    if ntexts:
        for text in [eval(x) for x in ntexts]:
            try:
                # fill in the appropriate empty field.
                fndict[text]['fntext'] = re.sub('\(\d+\)', '', line).strip()
            except KeyError:
                print "printer's error? ", "pgno:", pgno, line
```

Note that the `try: except:` blocks come to the rescue again here. The loop above kept breaking because in 3 instances it emerged that there existed footnotes at the bottom of a page for which there were no markers within the text. This was an editorial oversight in the published edition, not an OCR error. The result was that when I tried to address the non-existent entry in `fndict`, I got a `KeyError`. My `except:` clause allowed me to find and look at the error, and determine that the error was in the original and nothing I could do anything about, so when generating the final version of `charters` I replaced the `print` statement with `pass`. Texts made by humans are messy; no getting around it. `try: except:` exists to deal with that reality.

NOTA BENE: Again, bear in mind that we are modifying a data structure in memory rather than editing successive text files. So this loop should be ***added*** *to your script* ***below*** *the summary and marginal loop, which is* ***below*** *the loop that created your skeleton dictionary.*

Parse Dates and add to the dictionary

Dates are hard. Students of British history cling to Cheyney[296] as to a spar on a troubled ocean. And, given the way the Gregorian calendar was adopted so gradually, and innumerable other local variations, correct date reckoning for medieval sources will always require care and local knowledge. Nevertheless, here too Python can be of some help.

Our Italian summary line invariably contains a date drawn from the text, and it's conveniently set off from the rest of the line by parentheses. So we can parse them and create Python `date` objects. Then, if we want, we can do some simple calendar arithmetic.

[296] C.R. Cheney, Michael Jones, *A handbook of dates for students of British History* (2000).

First we have to find and correct all the dates in the same way as we have done for the other metadata elements. Devise a diagnostic script that will iterate over your `charters` dictionary, report the location of errors in your canonical text, and then fix them in your canonical text manually. Something like this:

```
summary_date = re.compile('\((\d{1,2})?(.*?)(\d{1,4})?\)') # we want to catc
h them all, and some have no day or month, hence the optional quantifiers: `
?`.

# And we want to make Python speak Italian:
ital2int = {'gennaio': 1, 'febbraio': 2, 'marzo': 3, 'aprile': 4, 'maggio':
5, 'giugno': 6, 'luglio': 7, 'agosto': 8, 'settembre': 9, 'ottobre': 10, 'no
vembre': 11, 'dicembre': 12}

import sys
for ch in charters:
    try:
        d = charters[ch]
        i = summary_date.finditer(d['summary'])
        dt = list(i)[-1]
        # Always the last parenthetical expression, in case there is more than one.

        if dt.group(2).strip() not in ital2int.keys():
            print "chno. %d fix the month %s" % (d['chno'], dt.group(2))
    except:
        print d['chno'], "The usual suspects ", sys.exc_info()[:2]
```

Note: When using try/except blocks, you should usually trap **`specific`** *errors in the except clause, like ValueError and the like; however, in ad hoc scripts like this, using sys.exc_info is a quick and dirty way to get information about any exception that may be raised. (The sys module*[297] *is full of such stuff, useful for debugging)*

Once you're satisfied that all the parenthetical date expressions are present and correct, and conform to your regular expression, you can parse them and add them to your data structure as dates rather than just strings. For this you can use the `datetime` module.

This module is part of the standard library, is a deep subject, and ought to be the subject of its own tutorial, given the importance of dates for historians. As with a lot of other python modules, a good introduction is Doug Hellmann's PyMOTW(module of the week).[298] An even more able extension library is mxDateTime.[299] Suffice it here to say that the `datetime.date` module expects parameters like this:

```
>>> from datetime import date
>>> dt = date(1160, 12, 25)
>>> dt.isoformat()
'1160-12-25'
```

[297] 'sys – System-specific Configuration': https://pymotw.com/2/sys/index.html#module-sys

[298] 'datetime – Date/time value manipulation': https://pymotw.com/2/datetime/

[299] 'mxDateTime – Date/Time Library for Python', *EGenix.com*: http://www.egenix.com/products/python/mxBase/mxDateTime/

So here's our loop to parse the dates at the end of the Italian summary lines and store them in our `charters` dictionary (remembering again that we want to modify our in-memory data structure `charters` created above):

```
summary_date = re.compile('\((\d{1,2})?(.*?)(\d{1,4})?\)')
from datetime import date
for ch in charters:
    c = charters[ch]
    i = summary_date.finditer(c['summary'])
    for m in i:
        # remember 'i' is an iterator so even if there is more than one
        # parenthetical expression in c['summary'], the try clause will
        # succeed on the last one, or fail on all of them.
        try:
            yr = int(m.group(3))
            mo = ital2int[m.group(2).strip()]
            day = int(m.group(1))
            c['date'] = date(yr, mo, day)
        except:
            c['date'] = "date won't parse, see summary line"
```

Out of 803 charters, 29 wouldn't parse, mostly because the date included only month and year. You can store these as strings, but then you have two data types claiming to be dates. Or you could supply a 01 as the default day and thus store a Python date object, but Jan. 1, 1160 isn't the same thing as Jan. 1160 and thus distorts your metadata. Or you could just do as I have done and refer to the relevant source text: the Italian summary line in the printed edition.

Once you've got date objects, you can do date arithmetic. Supposing we wanted to find all the charters dated to within 3 weeks of Christmas, 1160.

```
# Let's import the whole thing and use dot notation: datetime.date() etc.
import datetime

# a timedelta is a span of time
week = datetime.timedelta(weeks=1)

for ch in charters:
    try:
        dt = charters[ch]['date']
        christmas = datetime.date(1160,12,25)
        if abs(dt - christmas) < week * 3:
            print "chno: %s, date: %s" % (charters[ch]['chno'], dt)
    except:
        pass # avoid this idiom in production code
```

Which will give us this result:

```
chno: 790, date: 1160-12-14
chno: 791, date: 1160-12-15
chno: 792, date: 1161-01-01
chno: 793, date: 1161-01-04
chno: 794, date: 1161-01-05
chno: 795, date: 1161-01-05
chno: 796, date: 1161-01-10
```

```
chno: 797, date: 1161-01-10
chno: 798, date: 1161-01-06
```

Cool, huh?

Our completed data structure

Now we've corrected our canonical text as much as we need to to differentiate between the various bits of meta-data that we want to capture, and we've created a data structure in memory, our `charters` dictionary, by making 4 passes, each one extending and modifying the dictionary in memory.

create the skeleton

separate out the `summary` and `marginal` lines and create dictionary fields for them.

collect and assign footnotes to their respective charters

parse the dates in the `summary` field, and add them to their respective charters

Print out our resulting dictionary using `pprint(charters)` and you'll see something like this:

```
{
.
.
.
 52: {'chid': 'GScriba_LII',
      'chno': 52,
      'date': datetime.date(1156, 3, 27),
      'folio': '[fo. 6 r.]',
      'footnotes': [(1, 'Cancellato: m.')],
      'marginal': 'no marginal]',
      'pgno': 29,
      'summary': 'I consoli di Genova riconoscono con sentenza il diritto di
 Romano di Casella di pagarsi sui beni di Gerardo Confector per un credito c
he aveva verso il medesimo (27 marzo 1156).',
      'text': ['    In pontili capituli consules E. Aurie, W. Buronus, Ogeri
us Ventus laudaverunt quod Romanus de Casella haberet in bonis Gerardi Confe
ctoris s. .xxvi. denariorum et possit eos accipere sine contradicione eius e
t omnium pro eo. Hoc ideo quia, cum; Romanus ante ipsos inde conquereretur,
ipso Gerardo debitum non negante, sed quod de usura esset obiiciendo, iuravi
t nominatus Romanus quod capitalis erat (1) et non de usura, unde ut supra l
audaverunt , .MCLVI., sexto kalendas aprilis, indicione tercia.\n']},
 53: {'chid': 'GScriba_LIII',
      'chno': 53,
      'date': datetime.date(1156, 3, 27),
      'folio': '[fo. 6 r.]',
      'footnotes': [],
      'marginal': 'Belmusti]',
      'pgno': 29,
      'summary': "Maestro Arnaldo e Giordan nipote del fu Giovanni di Piacen
za si obbligano di pagare una somma nell'ottava della prossima Pasqua, per m
erce ricevuta (27 marzo 1156).",
     'text': ['  Testes Conradus Porcellus, Albericus, Vassallus Gambalixa,
 Petrus Artodi. Nos Arnaldus magister et Iordan nepos quondam Iohannis Place
ntie accepimus a te Belmusto tantum bracile unde promittimus dare tibi vel t
uo certo misso lb. .xLIII. denariorum usque octavam proximi pasce, quod si n
on fecerimus penam dupli tibi stipulanti promittimus, bona pignori, possis u
numquemque convenire de toto. Actum prope campanile Sancti Laurentii, milles
imo centesimo .Lv., sexto kalendas aprilis, indictione tercia.\n']},
.
.
. etc.
}
```

Printing out your Python dictionary as a literal string is not a bad thing to do. For a text this size, the resulting file is perfectly manageable, can be mailed around usefully and read into a python repl session very simply using `eval()`, or pasted directly into a Python module file. On the other hand, if you want an even more reliable way to serialize it in an exclusively Python context, look into Pickle.[300] If you need to move it to some other context, JavaScript for example, or some RDF triple stores, Python's json module[301] will translate effectively. If you have to get some kind of XML output, I will be very sorry for you, but the lxml python module[302] may ease the pain a little.

Order from disorder, huzzah.

Now that we have an ordered data structure, we can do many things with it. As a very simple example, let's append some code that just prints `charters` out as html for display on a web-site:

```
fout = open("your_page.html", 'w') # create a text file to write the html to

# write to the file your html header with some CSS formatting declarations
fout.write("""
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html>
<head>
  <title>Giovanni Scriba Vol. I</title>
  <style>
    h1 {text-align: center; color: #800; font-size: 16pt; margin-bottom: 0px; margi
n-top: 16px;}
    ul {list-style-type: none;}
    .sep {color: #800; text-align: center}
    .charter {width: 650px; margin-left: auto; margin-right: auto; margin-top: 60px;
 border-top: double #800;}
    .folio {color: #777;}
    .summary {color: #777; margin: 12px 0px 12px 12px;}
    .marginal {color: red}
    .charter-text {margin-left: 16px}
    .footnotes
    .page-number {font-size: 60%}
  </style></head>

<body>
```

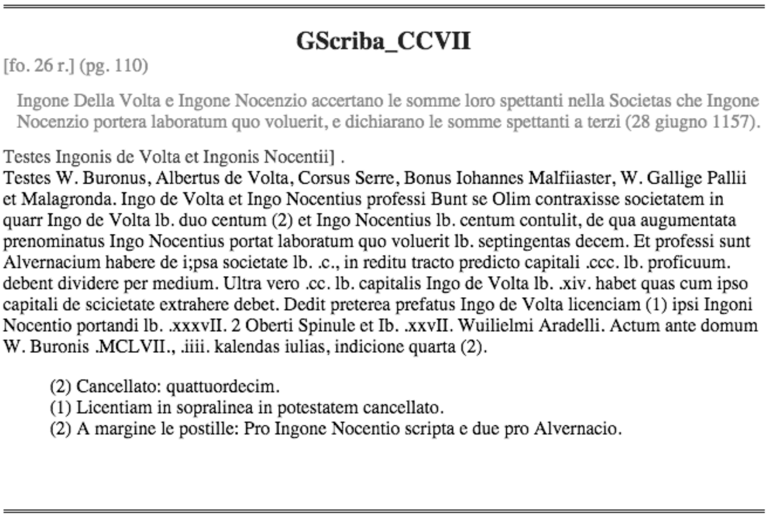
[300] 'Pickle - Python object serialization': https://docs.python.org/2/library/pickle.html

[301] 'json – JSON encoder and decoder': https://docs.python.org/2/library/json.html#module-json

[302] 'lxml – XML and HTML with Python', http://lxml.de/

```
""")

# a loop that will write out a blob of html code for each charter in our dictionary:
for x in charters:

    # use a shallow copy so charters[x] is not modified for this specialized purpose
    d = charters[x].copy()

    try:
        if d['footnotes']:
            # remember, this is a list of tuples. So you can feed them directly
            # to the string interpolation operator in the list comprehension.
            fnlist = ["<li>(%s) %s</li>" % t for t in d['footnotes']]
            d['footnotes'] = "<ul>" + ''.join(fnlist) + "</ul>"
        else:
            d['footnotes'] = ""

        d['text'] = ' '.join(d['text']) # d['text'] is a list of strings

        blob = """
            <div>
                <div class="charter">
                    <h1>%(chid)s</h1>
                    <div class="folio">%(folio)s (pg. %(pgno)d)</div>
                    <div class="summary">%(summary)s</div>
                    <div class="marginal">%(marginal)s</div>
                    <div class="text">%(text)s</div>
                    <div class="footnotes">%(footnotes)s</div>
                </div>
            </div>
            """

        fout.write(blob % d)

        # `string % dictionary` is a neat trick for html templating
        # that makes use of python's string interpolation syntax
        # see: http://www.diveintopython.net/html_processing/dictionary_based_string
_formatting.html

        fout.write("\n\n")
    except:
        # insert entries noting the absence of charters on the missing pg. 214
        erratum = """
            <div>
                <div class="charter">
                    <h1>Charter no. %d is missing because the scan for Pg. 214 was o
mmited</h1>
                </div>
            </div>
            """  % d['chno']

        fout.write(erratum)

fout.write("""</body></html>""")
```

Drop the resulting file on a web browser, and you've got a nicely formated electronic edition.



GScriba_CCVII

[fo. 26 r.] (pg. 110)

Ingone Della Volta e Ingone Nocenzio accertano le somme loro spettanti nella Societas che Ingone Nocenzio portera laboratum quo voluerit, e dichiarano le somme spettanti a terzi (28 giugno 1157).

Testes Ingonis de Volta et Ingonis Nocentii] .
Testes W. Buronus, Albertus de Volta, Corsus Serre, Bonus Iohannes Malfiiaster, W. Gallige Pallii et Malagronda. Ingo de Volta et Ingo Nocentius professi Bunt se Olim contraxisse societatem in quarr Ingo de Volta lb. duo centum (2) et Ingo Nocentius lb. centum contulit, de qua augumentata prenominatus Ingo Nocentius portat laboratum quo voluerit lb. septingentas decem. Et professi sunt Alvernacium habere de i;psa societate lb. .c., in reditu tracto predicto capitali .ccc. lb. proficuum. debent dividere per medium. Ultra vero .cc. lb. capitalis Ingo de Volta lb. .xiv. habet quas cum ipso capitali de scicietate extrahere debet. Dedit preterea prefatus Ingo de Volta licenciam (1) ipsi Ingoni Nocentio portandi lb. .xxxvII. 2 Oberti Spinule et Ib. .xxvII. Wuilielmi Aradelli. Actum ante domum W. Buronis .MCLVII., .iiii. kalendas iulias, indicione quarta (2).

(2) Cancellato: quattuordecim.
(1) Licentiam in sopralinea in potestatem cancellato.
(2) A margine le postille: Pro Ingone Nocentio scripta e due pro Alvernacio.

html formatted charter example

Being able to do this with your, still mostly uncorrected, OCR output is not a trivial advantage. If you're serious about creating a clean, error free, electronic edition of anything, you've got to do some serious proofreading. Having a source text formatted for reading is crucial; moreover, if your proofreader can change the font, spacing, color, layout, and so forth at will, you can increase their accuracy and productivity substantially. With this example in a modern web browser, tweaking those parameters with some simple CSS declarations is easy. Also, with some ordered HTML to work with, you might crowd-source the OCR error correction, instead of hiring that army of illiterate street urchins.

And, our original problem, OCR cleanup, is now much more tractable because we can target regular expressions for the specific sorts of metadata we have: errors in the Italian summary or in the Latin text? Or we could design search-and-replace routines just for specific charters, or groups of charters.

Beyond this though, there's lots you can do with an ordered data set, including feeding it back through a markup tool like the brat[303] as we did for the ChartEx project. Domain experts can then start adding layers of semantic tagging even if you don't do any further OCR error correction. Moreover, with an ordered dataset we can get all sorts of output, some other flavor of XML (if you must) for example: TEI (Text Encoding Initiative), or EAD (Encoded Archival Description). Or you could read your dataset directly into a relational database, or some kind of key/value store. All of these things are essentially impossible if you're working simply with a plain text file.

The bits of code above are in no way a turn-key solution for cleaning arbitrary OCR output. There is no such magic wand. The Google approach to scanning the contents of research libraries threatens to drown us in an

[303] 'brat rapid annotation tool', http://brat.nlplab.org/

ocean of bad data. Worse, it elides a fundamental fact of digital scholarship: digital sources are hard to get. Reliable, flexible, and useful digital texts require careful redaction and persistent curation. Google, Amazon, Facebook, *et alia* do not have to concern themselves with the quality of their data, just its quantity. Historians, on the other hand, must care first for the integrity of their sources.

The vast 18th and 19th century publishing projects, the *Rolls Series*, the *Monumenta Germaniae Historica*, and many others, bequeathed a treasure trove of source material to us by dint of a huge amount of very painstaking and detailed work by armies of dedicated and knowledgeable scholars. Their task was the same as ours: to faithfully transmit history's legacy from its earlier forms into a more modern form, thereby making it more widely accessible. We can do no less. We have powerful tools at our disposal, but while that may change the scale of the task, it does not change its nature.

About the Author

Jon Crump is an independent scholar and freelance digital humanist based in Seattle, Washington.

29. Transliterating non-ASCII characters with Python

Seth Bernstein – 2013

Lesson Goals:

This lesson shows how to use Python to transliterate automatically a list of words from a language with a non-Latin alphabet to a standardized format using the American Standard Code for Information Interchange (ASCII)[304] characters. It builds on readers' understanding of Python from the lessons "Understanding Web Pages and HTML," "Downloading Web Pages with Python," "From HTML to List of Words (part 1)" and "Intro to Beautiful Soup."[305] At the end of the lesson, we will use the transliteration dictionary to convert the names from a database of the Russian organization Memorial[306] from Cyrillic[307] into Latin characters.[308] Although the example uses Cyrillic characters, the technique can be reproduced with other alphabets using Unicode.[309]

What Is Transliteration and for Whom Is It Useful?

Transliteration is something that most people do every day, knowingly or not. Many English speakers would have trouble recognizing the name Владимир Путин but know that Vladimir Putin is Russia's current president. Transliteration is especially useful with names, because a standardized transliterated name is often the same as a translated name. (Exceptions are when someone's name is translated in a non-uniform way. Leon Trotsky's Russian name would be transliterated in a standardized form as Lev Trotskii.)

But transliteration has other uses too, especially for scholars. In many fields, the publishing convention is to transliterate any evidence used in the original. Moreover, citations from scholarly works need to be transliterated carefully so that readers can find and verify evidence used in

[304] 'ASCII', *Wikipedia*: https://en.wikipedia.org/wiki/Ascii

[305] William J. Turkel and Adam Crymble, 'Understanding Web Pages and HTML' (2012); 'Downloading Web Pages with Python' (2012); 'From HTML to List of Words (part 1)' (2012); Jeri Wieringa, 'Intro to Beautiful Soup', *The Programming Historian* (2012).

[306] 'Жертвы политического террора в СССР': http://lists.memo.ru/

[307] 'Cyrillic script', *Wikipedia*: https://en.wikipedia.org/wiki/Cyrillic_script

[308] 'Latin script', *Wikipedia*: https://en.wikipedia.org/wiki/Latin_script

[309] 'Unicode', *Wikipedia*: https://en.wikipedia.org/wiki/Unicode

texts. Finally, transliteration can be more practical for authors who can type more fluently with Latin letters than in the native alphabet of a language that does not use Latin characters.

Programming languages like Python also benefit from transliteration. Python handles Cyrillic relatively well in certain environments, like Terminal for MacOS or Linux,[310] or in Windows, IDLE, the official Python integrated development environment.[311] However, even in these Python converts non-ASCII characters into code. Other environments, like the Python shell for Windows (command line) or Komodo Edit,[312] know Unicode but will not print the Cyrillic characters that Unicode represents without tricky additional configuration. In environments that do support Cyrillic, switching between a Latin character set to write code and a non-Latin character set to handle inputs can be tedious. Thus, creating a program to transliterate evidence automatically eliminates the step of transliteration for researchers and it converts the text into a format that Python can handle more readily. **This lesson was built and tested using IDLE for Windows and Terminal for MacOS. The author strongly recommends that you follow along using the program tested on your operating system rather Windows Command Prompt or Komodo Edit.**

This lesson will be particularly useful for research in fields that use a standardized transliteration format, such as Russian history field, where the convention is to use a simplified version of the American Library Association-Library of Congress (ALA-LC) transliteration table.[313] (All tables currently available can be accessed here.) Researchers dealing with large databases of names can benefit considerably. However, this lesson will also allow practice with Unicode, character translation and using the parser Beautiful Soup in Python.[314]

Converting a Webpage to Unicode

The goal of this lesson is to take a list of names from a Russian database and convert them from Cyrillic into ASCII characters. The page we will use is from the site of the Russian human rights organization Memorial. During Glasnost[315] professional and amateur historians in the Soviet Union gained the ability to conduct research on previously taboo subjects, such as repression under Stalin. Banding together, they founded Memorial[316] to collect and publicize their findings. Today, the NGO conducts research on a

[310] 'Terminal (OS X)', *Wikipedia*: https://en.wikipedia.org/wiki/Terminal_%28OS_X%29

[311] 'IDLE (Python)', *Wikipedia*: https://en.wikipedia.org/wiki/IDLE_%28Python%29

[312] 'Komodo Edit': http://komodoide.com/komodo-edit/

[313] 'ALA-LC Romanization for Russia', *Wikipedia*: https://en.wikipedia.org/wiki/ALA-LC_romanization_for_Russian

[314] 'Beautiful Soup': http://www.crummy.com/software/BeautifulSoup/

[315] 'Glasnost', *Wikipedia*: https://en.wikipedia.org/wiki/Glasnost

[316] 'Жертвы политического террора в СССР': http://lists.memo.ru/

range of civil rights abuses in Russia, but collecting data about the victims of Stalinism remains one of its main functions. On the Memorial website researchers can find a database with some three million entries of people who were arrested or executed by Stalin's regime. It is an important resource on a dark topic. However, because the database has many, many names, it lends itself nicely to automated transliteration. This lesson will use just the first page of the database[317], but using the lesson on "Automated Downloading with Wget,"[318] it would be possible to go through the entire database as fast as your computer would allow.

We need to start by modifying the process found in the lesson "Downloading Web Pages with Python".[319] There we learned how to open and copy the HTML from a web page in Python. But what if we want to open a page in a language that does not use Latin characters? Python can do this but we need to tell it how to read these letters using a codec, a library of codes that allows Python to represent non-ASCII characters. Working with web pages makes this easy because almost all web pages specify what kind of encoding they use, in the page's *headers*. In Python, opening a web page does not just give you the HTML, but it creates an object with several useful characteristics. One is that we can access the headers by calling the `header()` method. This method returns something a lot like a Python dictionary with information that is important to web programmers. For our purposes, what is important is that the encoding is stored under the 'content-type' key.

```
#transliterator.py
import urllib2

page = urllib2.urlopen('http://lists.memo.ru/d1/f1.htm')

#what is the encoding?
print page.headers['content-type']
```

Under the 'content-type' key we find this information:

```
text/html; charset=windows-1251
```

The 'content-type' is telling us that the file stored at the url we accessed is in HTML and that its encoding (after 'charset=', meaning character set) is 'windows-1251', a common encoding for Cyrillic characters. You can visit the webpage and view the Page Source and see for yourself that the first line does in fact contain a 'content-type' variable with the value text/html; charset=windows-1251. It would not be so hard to work with the 'windows-1251' encoding. However, 'windows-1251' is specifically for Cyrillic and will not handle all languages. For the sake of learning a standard method, what we want is Unicode, a coding set that handles not just Cyrillic but

[317] Available at: http://lists.memo.ru/d1/f1.htm

[318] Ian Milligan, 'Automated Downloading with Wget', *The Programming Historian* (2012).

[319] William J. Turkel and Adam Crymble, 'Downloading Web Pages with Python', *The Programming Historian* (2012).

characters and symbols from virtually any language. (For more on Unicode, see the What is Unicode page.)[320] Converting into Unicode gives us the potential to create a transliteration table that could cover multiple languages and special characters in a way that region-specific character sets do not allow.

How do you convert the characters to Unicode? First, Python needs to know the original encoding of the source, 'windows-1251.' We could just assign 'windows-1251' to a variable by typing it manually but the encoding may not always be 'windows-1251.' There are other character sets for Cyrillic, not to mention other languages. Let's find a way to make the process more automatic for those cases. It helps that the encoding is the very last part of the string, so we can isolate it from everything that came before in the string. By using the `.split()` method, the string containing whatever encoding it is can be assigned to a variable. The `.split(separator)` method in Python returns a list of sections in the string that are split by some user-defined separator. Assigning no separator to `.split()` separates a string at the spaces. Another use of the `.split()` method is to separate by commas, which can help to work with comma separated value (csv) files.[321] In this case, though, by splitting the 'content-type' string at 'charset=', we get a *list* with two strings where the second will be the character set.

```
encoding = page.headers['content-type'].split('charset=')[1]
```

The encoding is assigned to the variable called '*encoding*'. You can check to see if this worked by printing the '*encoding*' variable. Now we can tell Python how to read the page as Unicode. Using the `unicode(object [, encoding])` method turns a string of characters into a Unicode object. A Unicode object is similar to a string but it can contain special characters. If they are in a non-ASCII character set, like here with 'windows-1251', we have to use the optional encoding parameter.

```
#read the HTML as a string into a variable
content = page.read()

# the unicode method tries to use ASCII so we need to tell it the encoding
content = unicode(content, encoding)
content[200:300]

u'"list-right">\r\n
<ul>
    <li>
<p class="name"><a name="n1"></a>\u0410-\u0410\u043a\u0443 \u0422\u0443\u043
b\u0438\u043a\u043e\u0432\u0438\u0447</p>
<p class="cont">\r\n\u0420\u043e\u0434\u0438\u043b\u0441\u044f\xa0\u0432 '</
p>
```

[320] 'What is Unicode?': http://lists.memo.ru/d1/f1.htm

[321] 'Comma-separated values', *Wikipedia*: https://en.wikipedia.org/wiki/Comma-separated_values

In some editors like Komodo, printing even Unicode will raise an error. Indeed, the inability of some Python environments to print Unicode out of the box is one big advantage of transliterating it into ASCII. In IDLE, though, we can print this content to see it in Cyrillic rather than Unicode:

```
# see what happens when Python prints Unicode
print content[200:300]

"list-right">
<ul>
    <li>
<p class="name"><a name="n1"></a>А-Аку Туликович</p>
Родился в
```

Excellent - the web page is now converted to Unicode. All the '\u0420'-type marks are Unicode and Python knows that they code to Cyrillic characters. The forward slash is called an '*escape character*' and allows Python to do things like use special characters in Unicode or signify a line break ('\n') in a document. Each counts as just one character. Now we can create a Python *dictionary* that will act as the transliteration table.

Unicode Transliteration Dictionary

A dictionary is an unordered collection of *key-object pairs*. What this means is that under each key, the dictionary stores some number or string or other object – even another dictionary. (See also the lesson "Counting Word Frequencies with Python.")[322] A dictionary has the following syntax:

```
my_dictionary = {'Vladimir': 'Putin', 'Boris': 'Yeltsin'}
print my_dictionary['Vladimir']

> Putin
```

How can we turn this into a transliteration table? Just make each Unicode character a key in the dictionary. Its value will be whatever character(s) it transliterates to. The table for Romanization of Russian is available from the Library of Congress.[323] This table needs to be simplified slightly. The ALA-LC suggests using characters with umlauts or ligatures to represent Cyrillic letters but those characters are no more ASCII than Cyrillic characters. So instead no umlauts or ligatures will be used.

Each Cyrillic letter has a different Unicode value. It would take time to find each one of them but fortunately Wikipedia has a table.[324] If the script were very rare, we could find it at the Unicode website.[325]

[322] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

[323] 'Russian', *Library of Congress:* http://www.lcweb.loc.gov/catdir/cpso/romanization/russian.pdf

[324] 'Cyrillic script in Unicode', *Wikipedia*: https://en.wikipedia.org/wiki/Cyrillic_script_in_Unicode

[325] 'Unicode 8.0 Character Code Charts', http://www.unicode.org/charts/

We just need to combine the transliteration table with the Unicode table. The Unicode value for the Russian letter "Ж" is 0416 and it transliterates to the Latin characters "Zh." Python needs more than just the Unicode identifier. It also needs to know to look out for a Unicode character. Therefore all the Unicode characters used in the dictionary should be in the format u'\uXXXX'. In this case, the letter Ж is u'\u0416'. We can create a transliteration dictionary and assign 'Zh' as the value for the key u'\u0416' in it.

```
cyrillic_translit = { u'\u0416': 'Zh'}
```

As it turns out, lowercase Cyrillic letters in Unicode have the same value as their uppercase counterparts except the value of the second number is two greater. Thus, 'ж' codes to 0436. Now that we have a transliteration dictionary created, we just add a dictionary key-value pair.

```
cyrillic_translit[u'\u0436'] = 'zh'
```

Of course, rather than do each pair one by one, it would probably be easier to write the dictionary in a Python module or paste it in from a word processor. The full Cyrillic transliteration dictionary is here:

```
cyrillic_translit={u'\u0410': 'A', u'\u0430': 'a',
u'\u0411': 'B', u'\u0431': 'b',
u'\u0412': 'V', u'\u0432': 'v',
u'\u0413': 'G', u'\u0433': 'g',
u'\u0414': 'D', u'\u0434': 'd',
u'\u0415': 'E', u'\u0435': 'e',
u'\u0416': 'Zh', u'\u0436': 'zh',
u'\u0417': 'Z', u'\u0437': 'z',
u'\u0418': 'I', u'\u0438': 'i',
u'\u0419': 'I', u'\u0439': 'i',
u'\u041a': 'K', u'\u043a': 'k',
u'\u041b': 'L', u'\u043b': 'l',
u'\u041c': 'M', u'\u043c': 'm',
u'\u041d': 'N', u'\u043d': 'n',
u'\u041e': 'O', u'\u043e': 'o',
u'\u041f': 'P', u'\u043f': 'p',
u'\u0420': 'R', u'\u0440': 'r',
u'\u0421': 'S', u'\u0441': 's',
u'\u0422': 'T', u'\u0442': 't',
u'\u0423': 'U', u'\u0443': 'u',
u'\u0424': 'F', u'\u0444': 'f',
u'\u0425': 'Kh', u'\u0445': 'kh',
u'\u0426': 'Ts', u'\u0446': 'ts',
u'\u0427': 'Ch', u'\u0447': 'ch',
u'\u0428': 'Sh', u'\u0448': 'sh',
u'\u0429': 'Shch', u'\u0449': 'shch',
u'\u042a': '"', u'\u044a': '"',
u'\u042b': 'Y', u'\u044b': 'y',
u'\u042c': "'", u'\u044c': "'",
u'\u042d': 'E', u'\u044d': 'e',
u'\u042e': 'Iu', u'\u044e': 'iu',
u'\u042f': 'Ia', u'\u044f': 'ia'}
```

Now that we have the transliteration dictionary, we can simply loop through every character in the source page and convert those Unicode characters in the dictionary. If we turn it into a procedure, then we can reuse it for other webpages.

```
def transliterate(word, translit_table):
    converted_word = ''
    for char in word:
        transchar = ''
        if char in translit_table:
            transchar = translit_table[char]
        else:
            transchar = char
        converted_word += transchar
    return converted_word
```

We can then call this function using the newly created dictionary and the webpage downloaded earlier.

```
#we will run it with the cyrillic_translit dictionary and the webpage
converted_content = transliterate(content, cyrillic_translit)
converted_content[200:310]
```

Here is what we end up with:

```
u'="list-right">\r\n</li>
    <li>
<p class="name"><a name="n1"></a>A-Aku Tulikovich</p>
<p class="cont">\r\nRodilsia\xa0v 1913 g.'</p>
```

Still not perfect. Python did not convert the special character '\xa0' that signifies a *non-breaking space*. But with the transliteration dictionary, any characters that pop up can just be added to the dictionary and they will be converted. First we need to find out what that character is. We could search for it on the Internet or we can just print it:

```
#Let's find out what u'\xa0' is
print u'\xa0'

#it's not nothing but a non-breaking space
#it would be better if our transliteration dictionary could change it into a
 space

cyrillic_translit[u'\xa0'] = ' '
```

With this fix, all the Cyrillic and special characters are gone, making it much easier to read the file and deal with it. For the last part of the lesson, we will modify methods used in the lesson "Intro to Beautiful Soup"[326] to get a list of transliterated names from the webpage.

[326] Jeri Wieringa, 'Intro to Beautiful Soup', *The Programming Historian* (2012).

Transliterated List of Names

There may be cases where it is best to transliterate the entire file but if the goal is to transliterate and extract just a part of the data in the file, it would be best to extract first and transliterate later. That way Python will only transliterate a small part of the file rather than having to loop through the whole of the HTML. Speed is not a huge issue when dealing with a handful of web pages but Memorial's site has thousands of pages. The difference between looping through thousands of whole pages and just looping through a small part of each of those pages can add up. But, of course, it would have been anti-climactic to have all the names before the transliteration dictionary and also more difficult for non-Cyrillic readers to understand the rest of the lesson. So now we need to find a way to get just the names from the page. Here is the first bit of HTML from the converted_content string, containing parts of two database entries:

```
converted_content[200:1000]
```

This code prints out characters 200 to 1000 of the HTML, which happens to include the entire first entry and the beginning of the second:

```
u'="list-right">\r\n</li>
    <li>
<p class="name"><a name="n1"></a>A-Aku Tulikovich</p>
<p</li>
    <li>class="cont">\r\nRodilsia v 1913 g., Kamchatskaia gub., Tigil\'skii
r-n, stoibishsha Utkholok; koriak-kochevnik; malogramotnyi; b/p; \r\n\r\n
Arestovan12 noiabria 1938 g.\r\n
Prigovoren: Koriakskii okrsud 8 aprelia 1939 g., ob</li>
</ul>


v.: po st. 58-2-8-9-10-11 UK RSFSR.\r\n
Prigovor: 20 let. Opredeleniem Voen

noi kollegii VS SSSR ot 17 oktiabria 1939 g. mera snizhena do 10 let.\r\nRea
bili

tirovan 15 marta 1958 g. Reabilitirovan opredeleniem Voennoi kollegii VS SSS
R\r\

n
<p class="author">Istochnik: Baza dannykh o zhertvakh repressii Kamchatskoi<
/p>
obl.
<ul>
    <li>\r\n</li>
    <li>
<p class="name"><a name="n2"></a>Aab Avgust Mikhailovich</p>
p>
<p class="cont">\r\nRodilsia v 1899 g., Saratovskaia obl., Grimm s.; nemets;
</p>
obrazovanie nachal\'noe;'
```

Each entry includes lots of information: name (last, first and patronymic), date of birth, place of birth, profession, date of arrest, date of sentencing and so on. If we wanted the detailed information about each person, we would have to parse the page ourselves and extract that information using the string manipulation techniques from the lesson “Manipulating Strings in Python.”[327] However, for just the names it will be quicker to use the HTML parsing module Beautiful Soup. If you have not installed Beautiful Soup, see “Installing Python Modules with pip”[328] and read “Intro to Beautiful Soup”[329] for an overview of how this tool works. In the transliterator module, we will load Beautiful Soup and then turn our converted page into a *Beautiful Soup object*.

```
#Load Beautiful Soup
from bs4 import BeautifulSoup

#convert the page
converted_soup = BeautifulSoup(converted_content)
```

The lesson “Intro to Beautiful Soup” teaches how to grab sections of a web page by their tags. But we can also select sections of the page by *attributes*, HTML code that modifies elements. Looking at the HTML from this page, notice that the text of our names are enclosed in the tag <p class=“name”>. The class attribute allows the page’s Cascading Style Sheets (CSS)[330] settings to change the look of all elements that share the “name” *class* at once. CSS itself is an important tool for web designers. For those interested in learning more on this aspect of CSS, I recommend Code Academy’s[331] interactive lessons in its web fundamentals track. In mining data from the web, though, attributes like class give us a pattern to separate out certain values.

What we want is to get the elements where the class attribute’s value is “name”. When dealing with most types of attributes, Beautiful Soup can select parts of the page using the same syntax as HTML. The class attribute makes things a little tricky because Python uses “class” to define new types of objects. Beautiful Soup gets around this by making us search for class followed by an underscore: `class_=“value”`. Beautiful Soup objects’ `.find_all()` method will generate a Python list of Beautiful Soup objects that match the HTML tags or attributes set as *parameters*. The method `.get_text()` extracts just the text from Beautiful Soup objects, so `“ <p class=“name”><a name=“n1”></a>A-Aku Tulikovich</p> “.get_text()` will become “*A-Aku Tulikovich*”. We need to use

[327] William J. Turkel and Adam Crymble, ‘Manipulating Strings in Python’, *The Programming Historian* (2012).

[328] Fred Gibbs ‘Installing Python Modules with pip’, *The Programming Historian* (2013).

[329] Jeri Wieringa, ‘Intro to Beautiful Soup’, *The Programming Historian* (2012).

[330] ‘CSS Tutorial’, *W3Schools*: http://www.w3schools.com/css/

[331] ‘CSS: Coding with Style’, *Code Academy*: https://www.codecademy.com/courses/css-coding-with-style/0/1

`.get_text()` on each item in the list, then append it to a new list containing just the names:

```
#creating the final names list
names = []

#creating the list with .find_all() and looping through it
for entry in converted_soup.find_all(class_="name"):
    names.append(entry.get_text())
```

To make sure it worked, let's check the number of names and then see if they look like we expect:

```
#check the number of names
len(names)

> 190

#see the first twenty names in the list
names[:20]

> [u'A-Aku Tulikovich ', u'Aab Avgust Mikhailovich', u'Aab Avgust Khristiano
vich', u'Aab Aleksandr Aleksandrovich', u"Aab Aleksandr Khrist'ianovich", u"
Aab Al'bert Viktorovich", u"Aab Al'brekht Aleksandrovich", u'Aab Amaliia And
reevna', u'Aab Amaliia Ivanovna', u'Aab Angelina Andreevna', u'Aab Andrei An
dreevich', u'Aab Andrei Filippovich', u'Aab Arvid Karlovich', u"Aab Arnol'd
Aleksandrovich", u'Aab Artur Avgustovich', u"Aab Artur Vil'gel'movich", u"Aa
b Aelita Arnol'dovna", u'Aab Viktor Aleksandrovich', u'Aab Viktor Aleksandro
vich', u"Aab Viktor Vil'gel'movich"]
```

The 'u' in front of each of the names indicates that they are *unicode objects* in Python, not *strings*. But when Python needs a string, it will automatically change any unicode to be a string if it only uses ASCII characters or else throw a "unicodedecode error". Fortunately, because we have transliterated all the Cyrillic characters, this list fits Python's needs. If we had not parsed the transliterated page, that would be easy to handle with the transliterate function from earlier. All it would take is to use the transliterate function on the text from each item in the list before appending it to the final list.

Transliteration can only do so much. Except for proper names, it can tell you little about the content of the source being transliterated. Yet the ability to transliterate automatically is of great use when dealing with lots of names or for people who prefer or need to use ASCII characters. It is a simple tool but one that can be an enormous time saver.

About the Author

Seth Bernstein is a postdoctoral fellow at the Higher School of Economics in Moscow. He defended his doctoral dissertation, 'Communist Upbringing under Stalin: The Political Socialization and Militarization of Soviet Youths, 1934-1941' at the University of Toronto, in 2013.

Part Four: Analyzing Data

You have your materials, you've cleaned them up. Now you want to see what they can tell you. The lessons in this section deal with different approaches to analyzing historical data. This is where research meets computation. What comes out the other side is up to you.

30. Counting Word Frequencies with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Normalizing Textual Data with Python'.[332]

Lesson Goals

Your list is now clean enough that you can begin analyzing its contents in meaningful ways. Counting the frequency of specific words in the list can provide illustrative data. Python has an easy way to count frequencies, but it requires the use of a new type of variable: the *dictionary*. Before you begin working with a dictionary, consider the processes used to calculate frequencies in a list.

Files Needed For This Lesson

`obo.py`

If you do not have these files, you can download a (zip - http://programminghistorian.org/assets/programming-historian3.zip) file from the previous lesson.

Frequencies

Now we want to count the frequency of each word in our list. You've already seen that it is easy to process a list by using a `for` loop. Try saving and executing the following example. Recall that += tells the program to append something to the end of an existing variable.

[332] William J. Turkel and Adam Crymble, 'Normalizing Textual Data with Python', *The Programming Historian* (2012).

```
# count-list-items-1.py

wordstring = 'it was the best of times it was the worst of times '
wordstring += 'it was the age of wisdom it was the age of foolishness'

wordlist = wordstring.split()

wordfreq = []
for w in wordlist:
    wordfreq.append(wordlist.count(w))

print "String\n" + wordstring +"\n"
print "List\n" + str(wordlist) + "\n"
print "Frequencies\n" + str(wordfreq) + "\n"
print "Pairs\n" + str(zip(wordlist, wordfreq))
```

Here, we start with a string and split it into a list, as we've done before. We then create an (initially empty) list called *wordfreq*, go through each word in the *wordlist*, and count the number of times that word appears in the whole list. We then add each word's count to our *wordfreq* list. Using the `zip` operation, we are able to match the first word of the word list with the first number in the frequency list, the second word and second frequency, and so on. We end up with a list of word and frequency pairs. The `str` function converts any object to a string so that it can be printed.

You should get something like this:

```
String
it was the best of times it was the worst of times it was the age of wisdom
it was the age of foolishness

List
['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was',
'the', 'worst', 'of', 'times', 'it', 'was', 'the', 'age',
'of', 'wisdom', 'it', 'was', 'the', 'age', 'of',
'foolishness']

Frequencies
[4, 4, 4, 1, 4, 2, 4, 4, 4, 1, 4, 2, 4, 4, 4, 2, 4, 1, 4,
4, 4, 2, 4, 1]

Pairs
[('it', 4), ('was', 4), ('the', 4), ('best', 1), ('of', 4),
('times', 2), ('it', 4), ('was', 4), ('the', 4),
('worst', 1), ('of', 4), ('times', 2), ('it', 4),
('was', 4), ('the', 4), ('age', 2), ('of', 4),
('wisdom', 1), ('it', 4), ('was', 4), ('the', 4),
('age', 2), ('of', 4), ('foolishness', 1)]
```

It will pay to study the above code until you understand it before moving on. Python also includes a very convenient tool called a `list` `comprehension`,[333] which can be used to do the same thing as the `for` loop more economically.

```
# count-list-items-1.py

wordstring = 'it was the best of times it was the worst of times '
wordstring += 'it was the age of wisdom it was the age of foolishness'
wordlist = wordstring.split()

wordfreq = [wordlist.count(w) for w in wordlist] # a list comprehension

print "String\n" + wordstring +"\n"
print "List\n" + str(wordlist) + "\n"
print "Frequencies\n" + str(wordfreq) + "\n"
print "Pairs\n" + str(zip(wordlist, wordfreq))
```

If you study this list comprehension carefully, you will discover that it does exactly the same thing as the `for` loop in the previous example, but in a condensed manner. Either method will work fine, so use the version that you are most comfortable with.

At this point we have a list of pairs, where each pair contains a word and its frequency. This list is a bit redundant. If 'the' occurs 500 times, then this list contains five hundred copies of the pair ('the', 500). The list is also ordered by the words in the original text, rather than listing the words in order from most to least frequent. We can solve both problems by converting it into a dictionary, then printing out the dictionary in order from the most to the least commonly occurring item.

Python Dictionaries

Both strings and lists are sequentially ordered, which means that you can access their contents by using an index, a number that starts at 0. If you have a list containing strings, you can use a pair of indexes to access first a particular string in the list, and then a particular character within that string. Study the examples below.

```
s = 'hello world'
print s[0]
-> h

print s[1]
-> e

m = ['hello', 'world']
print m[0]
-> hello

print m[1]
-> world
```

[333] 'List Comprehensions', *Python*: https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

```
print m[0][1]
-> e

print m[1][0]
-> w
```

To keep track of frequencies, we're going to use another type of Python object, a dictionary. The dictionary is an *unordered* collection of objects. That means that you can't use an index to retrieve elements from it. You can, however, look them up by using a key (hence the name "dictionary"). Study the following example.

```
d = {'world': 1, 'hello': 0}
print d['hello']
-> 0

print d['world']
-> 1

print d.keys()
-> ['world', 'hello']
```

Dictionaries might be a bit confusing to a new programmer. Try to think of it like a language dictionary. If you don't know (or remember) exactly how "bijection" differs from "surjection" you can look the two terms up in the *Oxford English Dictionary*. The same principle applies when you `print d['hello']`; except, rather than print a literary definition it prints the value associated with the keyword 'hello', as defined by you when you created the dictionary named *d*. In this case, that value is "0".

Note that you use curly braces to define a dictionary, but square brackets to access things within it. The `keys` operation returns a list of keys that are defined in the dictionary.

Word-Frequency Pairs

Building on what we have so far, we want a function that can convert a list of words into a dictionary of word-frequency pairs. The only new command that we will need is `dict`, which makes a dictionary from a list of pairs. Copy the following and add it to the `obo.py` module.

```
# Given a list of words, return a dictionary of
# word-frequency pairs.

def wordListToFreqDict(wordlist):
    wordfreq = [wordlist.count(p) for p in wordlist]
    return dict(zip(wordlist,wordfreq))
```

We are also going to want a function that can sort a dictionary of word-frequency pairs by descending frequency. Copy this and add it to the `obo.py` module, too.

```
# Sort a dictionary of word-frequency pairs in
# order of descending frequency.

def sortFreqDict(freqdict):
    aux = [(freqdict[key], key) for key in freqdict]
    aux.sort()
    aux.reverse()
    return aux
```

We can now write a program which takes a URL and returns word-frequency pairs for the web page, sorted in order of descending frequency. Copy the following program into Komodo Edit, save it as `html-to-freq.py` and execute it. Study the program and its output carefully before continuing.

```
#html-to-freq.py

import urllib2, obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
html = response.read()
text = obo.stripTags(html).lower()
wordlist = obo.stripNonAlphaNum(text)
dictionary = obo.wordListToFreqDict(wordlist)
sorteddict = obo.sortFreqDict(dictionary)

for s in sorteddict: print str(s)
```

Removing Stop Words

When we look at the output of our `html-to-freq.py` program, we see that a lot of the most frequent words in the text are function words like "the", "of", "to" and "and".

```
(192, 'the')
(105, 'i')
(74, 'to')
(71, 'was')
(67, 'of')
(62, 'in')
(53, 'a')
(52, 'and')
(50, 'you')
(50, 'he')
(40, 'that')
(39, 'his')
(36, 'it')
```

These words are usually the most common in any English language text, so they don't tell us much that is distinctive about Bowsey's trial. In general, we are more interested in finding the words that will help us differentiate this text from texts that are about different subjects. So we're going to filter out the common function words. Words that are ignored like this are known as stop words. We're going to use the following list, adapted from one posted online by computer scientists at Glasgow.[334] Copy it and put it at the beginning of the `obo.py` library that you are building.

```
stopwords = ['a', 'about', 'above', 'across', 'after', 'afterwards']
stopwords += ['again', 'against', 'all', 'almost', 'alone', 'along']
stopwords += ['already', 'also', 'although', 'always', 'am', 'among']
stopwords += ['amongst', 'amoungst', 'amount', 'an', 'and', 'another']
stopwords += ['any', 'anyhow', 'anyone', 'anything', 'anyway', 'anywhere']
stopwords += ['are', 'around', 'as', 'at', 'back', 'be', 'became']
stopwords += ['because', 'become', 'becomes', 'becoming', 'been']
stopwords += ['before', 'beforehand', 'behind', 'being', 'below']
stopwords += ['beside', 'besides', 'between', 'beyond', 'bill', 'both']
stopwords += ['bottom', 'but', 'by', 'call', 'can', 'cannot', 'cant']
stopwords += ['co', 'computer', 'con', 'could', 'couldnt', 'cry', 'de']
stopwords += ['describe', 'detail', 'did', 'do', 'done', 'down', 'due']
stopwords += ['during', 'each', 'eg', 'eight', 'either', 'eleven', 'else']
stopwords += ['elsewhere', 'empty', 'enough', 'etc', 'even', 'ever']
stopwords += ['every', 'everyone', 'everything', 'everywhere', 'except']
stopwords += ['few', 'fifteen', 'fifty', 'fill', 'find', 'fire', 'first']
stopwords += ['five', 'for', 'former', 'formerly', 'forty', 'found']
stopwords += ['four', 'from', 'front', 'full', 'further', 'get', 'give']
stopwords += ['go', 'had', 'has', 'hasnt', 'have', 'he', 'hence', 'her']
stopwords += ['here', 'hereafter', 'hereby', 'herein', 'hereupon', 'hers']
stopwords += ['herself', 'him', 'himself', 'his', 'how', 'however']
stopwords += ['hundred', 'i', 'ie', 'if', 'in', 'inc', 'indeed']
stopwords += ['interest', 'into', 'is', 'it', 'its', 'itself', 'keep']
stopwords += ['last', 'latter', 'latterly', 'least', 'less', 'ltd', 'made']
stopwords += ['many', 'may', 'me', 'meanwhile', 'might', 'mill', 'mine']
stopwords += ['more', 'moreover', 'most', 'mostly', 'move', 'much']
stopwords += ['must', 'my', 'myself', 'name', 'namely', 'neither', 'never']
stopwords += ['nevertheless', 'next', 'nine', 'no', 'nobody', 'none']
stopwords += ['noone', 'nor', 'not', 'nothing', 'now', 'nowhere', 'of']
stopwords += ['off', 'often', 'on','once', 'one', 'only', 'onto', 'or']
stopwords += ['other', 'others', 'otherwise', 'our', 'ours', 'ourselves']
stopwords += ['out', 'over', 'own', 'part', 'per', 'perhaps', 'please']
stopwords += ['put', 'rather', 're', 's', 'same', 'see', 'seem', 'seemed']
stopwords += ['seeming', 'seems', 'serious', 'several', 'she', 'should']
stopwords += ['show', 'side', 'since', 'sincere', 'six', 'sixty', 'so']
stopwords += ['some', 'somehow', 'someone', 'something', 'sometime']
stopwords += ['sometimes', 'somewhere', 'still', 'such', 'system', 'take']
stopwords += ['ten', 'than', 'that', 'the', 'their', 'them', 'themselves']
stopwords += ['then', 'thence', 'there', 'thereafter', 'thereby']
stopwords += ['therefore', 'therein', 'thereupon', 'these', 'they']
stopwords += ['thick', 'thin', 'third', 'this', 'those', 'though', 'three']
stopwords += ['three', 'through', 'throughout', 'thru', 'thus', 'to']
stopwords += ['together', 'too', 'top', 'toward', 'towards', 'twelve']
stopwords += ['twenty', 'two', 'un', 'under', 'until', 'up', 'upon']
stopwords += ['us', 'very', 'via', 'was', 'we', 'well', 'were', 'what']
stopwords += ['whatever', 'when', 'whence', 'whenever', 'where']
stopwords += ['whereafter', 'whereas', 'whereby', 'wherein', 'whereupon']
stopwords += ['wherever', 'whether', 'which', 'while', 'whither', 'who']
stopwords += ['whoever', 'whole', 'whom', 'whose', 'why', 'will', 'with']
stopwords += ['within', 'without', 'would', 'yet', 'you', 'your']
stopwords += ['yours', 'yourself', 'yourselves']
```

Now getting rid of the stop words in a list is as easy as using another list comprehension. Add this function to the `obo.py` module, too.

[334] 'stop words', http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words

```
# Given a list of words, remove any that are
# in a list of stop words.

def removeStopwords(wordlist, stopwords):
    return [w for w in wordlist if w not in stopwords]
```

Putting it All Together

Now we have everything we need to determine word frequencies for web pages. Copy the following to Komodo Edit, save it as `html-to-freq-2.py` and execute it.

```
# html-to-freq-2.py

import urllib2
import obo

url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

response = urllib2.urlopen(url)
html = response.read()
text = obo.stripTags(html).lower()
fullwordlist = obo.stripNonAlphaNum(text)
wordlist = obo.removeStopwords(fullwordlist, obo.stopwords)
dictionary = obo.wordListToFreqDict(wordlist)
sorteddict = obo.sortFreqDict(dictionary)

for s in sorteddict: print str(s)
```

If all went well, your output should look like this:

```
(25, 'house')
(20, 'yes')
(20, 'prisoner')
(19, 'mr')
(17, 'man')
(15, 'akerman')
(14, 'mob')
(13, 'black')
(12, 'night')
(11, 'saw')
(9, 'went')
(9, 'sworn')
(9, 'room')
(9, 'pair')
(9, 'know')
(9, 'face')
(8, 'time')
(8, 'thing')
(8, 'june')
(8, 'believe')
...
```

Suggested Readings

Lutz, Learning Python[335]

Ch. 9: Tuples, Files, and Everything Else
Ch. 11: Assignment, Expressions, and print
Ch. 12: if Tests
Ch. 13: while and for Loops

Pilgrim, Diving into Python[336]

Ch. 7: Regular Expressions

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each chapter you can download the "programming-historian" zip file to make sure you have the correct code.

programming-historian-3
zip: http://programminghistorian.org/assets/programming-historian3.zip

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Creating and Viewing HTML Files with Python'.[337]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[335] Mark Lutz, *Learning Python* (5th edition), O'Reilly: 2013.

[336] Pilgrim, 'Regular Expressions', *Diving into Python:* http://www.diveintopython.net/regular_expressions/index.html

[337] William J. Turkel and Adam Crymble, 'Creating and Viewing HTML Files with Python', *The Programming Historian*, 2012.

31. Counting Frequencies from Zotero Items

Spencer Roberts – 2013

Editor's Note: This is the third of three lessons on the 'Zotero API'. You may find it easier to complete this tutorial if you have already completed the previous one: 'Adding New Items to Zotero.[338]

Lesson Goals

In 'Counting Word Frequencies with Python'[339] you learned how to count the frequency of specific words in a list using python. In this lesson, we will expand on that topic by showing you how to get information from Zotero HTML items, save the content from those items, and count the frequencies of words. It may be beneficial to look over the previous lesson before we begin.

Files Needed For This Lesson

`obo.py`

If you do not have these files, you can download a zip file [http://programminghistorian.org/assets/programming-historian3.zip].

Modifying the obo.py Module

Before we begin, we need to adjust `obo.py` in order to use this module to interact with different html files. The *stripTags* function in the `obo.py` module must be updated to the following, because it was previously designed for Old Bailey Online[340] content only. First, we need to remove the line that instructs the program to begin at the end of the header, then we will tell it where to begin. Open the `obo.py` file in your text editor and follow the instructions below:

[338] Amanda Morton, 'Adding New Items to Zotero', *the Programming Historian* (2013).

[339] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

[340] Tim Hitchcock, Robert Shoemaker, Clive Emsley, Sharon Howard and Jamie McLaughlin, *et al.*, *The Old Bailey Proceedings Online, 1674-1913* (www.oldbaileyonline.org, version 7.0, 24 March 2012).

```
def stripTags(pageContents):
    #remove the following line
    #startLoc = pageContents.find("<hr/><h2>")

    #modify the following line
    #pageContents = pageContents[startLoc:]
    #so that it looks like this
    pageContents = pageContents[0:]

    inside = 0
    text = ' '
    for char in pageContents:
        if char == '<':
            inside = 1
        elif (inside == 1 and char =='>'):
            inside = 0
        elif inside == 1:
            continue
        else:
            text += char

    return text
```

Remember to save your changes before we continue.

Get Items from Zotero and Save Local Copy

After we have modified the `obo.py` file, we can create a program designed to request the top two items from a collection within a Zotero library, retrieve their associated URLs, read the web pages, and save the content to a local copy. This particular program will only work on webpage-type items with html content (for instance, entering the URLs of JSTOR or Google Books pages will not result in an analysis of the actual content).

First, create a new .py file and save it in your programming historian directory. Make sure your copy of the `obo.py` file is in the same location. Once you have saved your file, we can begin by importing the libraries and program data we will need to run this program:

```
#Get urls from Zotero items, create local copy, count frequencies
import obo
from libZotero import zotero
import urllib2
```

Next, we need to tell our program where to find the items we will be using in our analysis. Using the sample Zotero library from which we retrieved items in Amanda Morton's lesson 'Intro to the Zotero API',[341] or using your personal library, we will pull the first two top-level items from either the library or from a specific collection within the library. (To find your collection key, mouseover the RSS button on that collection's page and use the second alpha-numeric sequence in the URL. If you are trying to connect

[341] Amanda Morton, 'Intro to the Zotero API', *The Programming Historian* (2013).

to an individual user library, you must change the word group to the word user, replace the six-digit number with your user ID, and insert your own API key.)

```
#links to Zotero library
zlib = zotero.Library('group', '155975', '<null>', 'f4Bfk3OTYb7bukNwfcKXKNLG
')

#specifies subcollection - leave blank to use whole library
collectionKey = 'I253KRDT'

#retrieves top two items from library
items = zlib.fetchItemsTop({'limit': 2, 'collectionKey': collectionKey, 'con
tent': 'json,bib,coins'})
```

Now we can instruct our program to retrieve the URL from each of our items, create a filename using that URL, and save a copy of the html on the page.

```
#retrieves url from each item, creates a filename from the url,
#saves a local copy
for item in items:
    url = item.get('url')
    filename = url.split('/')[-1] + '.html'           #splits url at last /
    filename = filename.split('=')[-1]                #splits url at last =
    filename = filename.replace('.html.html', '.html') #removes double .html
    print 'Saving local copy of ' + filename

    response = urllib2.urlopen(url)
    webContent = response.read()
    f = open(filename,'w')
    f.write(webContent)
    f.close()
```

Running this portion of the program will result in the following:

```
Saving local copy of PastsFutures.html
Saving local copy of 29.html
```

Get Item URLs from Zotero and Count Frequencies

Now that we've retrieved our items and created local html files, we can use the next portion of our program to retrieve the URLs, read the web pages, create a list of words, count their frequencies, and display them. Most of this should be familiar to you from the ‘Counting Word Frequencies with Python’ lesson.[342]

[342] William J. Turkel and Adam Crymble, ‘Counting Word Frequencies with Python’, *The Programming Historian* (2012).

```
#retrieves url from each item, creates a filename from the url
for item in items:
    itemTitle = item.get('title')
    url = item.get('url')
    filename = url.split('/')[-1] + '.html'           #splits url at last /
    filename = filename.split('=')[-1]                #splits url at last =
    filename = filename.replace('.html.html', '.html') #removes double .html
    print '\n' + itemTitle +'\nFilename: ' + filename + '\nWord Frequencies\n'
    response = urllib2.urlopen(url)
    html = response.read()
```

This section of code grabs the URL from our items, removes the unnecessary portions, and creates and prints a filename. For the items in our sample collection, the output looks something like this:

```
 The Pasts and Futures of Digital History
Filename: PastsFutures.html
Word Frequencies

History and the Web, From the Illustrated Newspaper to Cyberspace: Visual Te
chnologies and Interaction in the Nineteenth and Twenty-First Centuries
Filename: 29.html
Word Frequencies
```

Now we can go ahead and create our list of words and their frequencies. Enter the following:

```
#strips HTML tags, strips nonAlpha characters, removes stopwords
    text = obo.stripTags(html).lower()
    fullwordlist = obo.stripNonAlphaNum(text)
    wordlist = obo.removeStopwords(fullwordlist, obo.stopwords)

#counts frequencies
    dictionary = obo.wordListToFreqDict(wordlist)
    sorteddict = obo.sortFreqDict(dictionary)

#displays list of words and frequencies
    for s in sorteddict: print str(s)
```

Your final output will include a long list of words accompanied by their frequency within the html file:

```
Saving local copy of PastsFutures.html
Saving local copy of 29.html

The Pasts and Futures of Digital History
Filename: PastsFutures.html
Word Frequencies

(51, 'history')
(43, 'new')
(31, '9')
(27, 'historians')
(24, 'digital')
(23, 'social')
(21, 'narrative')
(16, 'media')
(15, 'time')
(13, 'possibilities')
(13, 'past')
(12, 'science')
...

History and the Web, From the Illustrated Newspaper to Cyberspace: Visual Techno
logies and Interaction in the Nineteenth and Twenty-First Centuries
Filename: 29.html
Word Frequencies

(52, 'new')
(49, 'history')
(46, 'media')
(44, 'ndash')
(34, 'figure')
(34, 'digital')
(24, 'visual')
(24, 'museum')
(24, 'http')
(23, 'edu')
(22, 'web')
(22, 'text')
(22, 'barnum')
(21, 'users')
(21, 'information')
...
```

About the Author

Spencer Roberts is a Research Assistant and former Digital History Research Fellow at the Roy Rosenzweig Center for History and New Media, and a PhD graduate student at George Mason University in the Department of History.

32. Getting Started with Topic Modeling and MALLET

Shawn Graham, Scott Weingart, and Ian Milligan – 2012

Editor's Note

This lesson requires you to use the command line. If you have no previous experience using the command line you may find it helpful to work through the Programming Historian Bash Command Line lesson.[343]

Lesson Goals

In this lesson you will first learn what *topic modeling* is and why you might want to employ it in your research. You will then learn how to install and work with the MALLET *natural language processing* toolkit to do so. MALLET involves modifying an *environment variable* (essentially, setting up a short-cut so that your computer always knows where to find the MALLET program) and working with the *command line* (ie, by typing in commands manually, rather than clicking on icons or menus). We will run the topic modeller on some example files, and look at the kinds of outputs that MALLET installed. This will give us a good idea of how it can be used on a corpus of texts to identify topics found in the documents without reading them individually.

Please see the MALLET users' discussion list[344] for the full range of things one can do with the software.

(We would like to thank Robert Nelson and Elijah Meeks for hints and tips in getting MALLET to run for us the first time, and for their examples of what can be done with this tool.)

What is Topic Modeling And For Whom is this Useful?

A *topic modeling* tool takes a single text (or corpus) and looks for patterns in the use of words; it is an attempt to inject semantic meaning into vocabulary. Before you begin with topic modeling, you should ask yourself whether or not it is likely to be useful for your project. Matthew Kirschenbaum's *Distant Reading*[345] (a talk given at the 2009 National Science Foundation Symposium on the Next Generation of Data Mining and Cyber-Enabled Discovery for Innovation) and Stephen Ramsay's

[343] Ian Milligan and James Baker, 'Introduction to the Bash Command Line', *The Programming Historain* (2014).

[344] 'The Mallet Development Mailling List': http://mallet.cs.umass.edu/mailinglist.php

[345] Matthew G. Kirschenbaum, 'The Remaking of Reading: Data Mining and the Digital Humanities': http://www.csee.umbc.edu/~hillol/NGDM07/abstracts/talks/MKirschenbaum.pdf

Reading Machines[346] are good places for beginning to understand in which circumstances a technique such as this could be most effective. As with all tools, just because you can use it, doesn't necessarily mean that you should. If you are working with a small number of documents (or even a single document) it may well be that simple frequency counts are sufficient, in which case something like Voyant Tools[347] might be appropriate. However, if you have hundreds of documents from an archive and you wish to understand something of what the archive contains without necessarily reading every document, then topic modeling might be a good approach.

Topic models represent a family of computer programs that extract *topics* from *texts*. A topic to the computer is a list of words that occur in statistically meaningful ways. A text can be an email, a blog post, a book chapter, a journal article, a diary entry – that is, any kind of unstructured text. By unstructured we mean that there are no computer-readable annotations that tell the computer the semantic meaning of the words in the text.

Topic modeling programs do not know anything about the meaning of the words in a text. Instead, they assume that any piece of text is composed (by an author) by selecting words from possible baskets of words where each basket corresponds to a topic. If that is true, then it becomes possible to mathematically decompose a text into the probable baskets from whence the words first came. The tool goes through this process over and over again until it settles on the most likely distribution of words into baskets, which we call topics.

There are many different topic modeling programs available; this tutorial uses one called MALLET. If one used it on a series of political speeches for example, the program would return a list of topics and the keywords composing those topics. Each of these lists is a topic according to the algorithm. Using the example of political speeches, the list might look like:

Job Jobs Loss Unemployment Growth
Economy Sector Economics Stock Banks
Afghanistan War Troops Middle-East Taliban Terror
Election Opponent Upcoming President
et cetera

By examining the keywords we can discern that the politician who gave the speeches was concerned with the economy, jobs, the Middle East, the upcoming election, and so on.

[346] Stephen Ramsay, *Reading Machines: Towards an Algorithmic Criticism* (University of Illinois, 2011).
[347] 'Voyant Tools': http://voyant-tools.org/

As Scott Weingart warns, there are many dangers that face those who use topic modeling without fully understanding it.[348] For instance, we might be interested in word use as a proxy for placement along a political spectrum. Topic modeling could certainly help with that, but we have to remember that the proxy is not in itself the thing we seek to understand – as Andrew Gelman demonstrates in his mock study of zombies using Google Trends.[349] Ted Underwood and Lisa Rhody (see Further Reading) argue that we as historians would be better to think of these categories as discourses; however for our purposes here we will continue to use the word: topic.

Note: You will sometimes come across the term "*LDA*" when looking into the bibliography of topic modeling. LDA and Topic Model are often used synonymously, but the LDA technique is actually a special case of topic modeling created by David Blei and friends in 2002.[350] It was not the first technique now considered topic modeling, but it is by far the most popular. The myriad variations of topic modeling have resulted in an alphabet soup of techniques and programs to implement them that might be confusing or overwhelming to the uninitiated; ignore them for now. They all work in much the same way. MALLET uses LDA.

Examples of topic models employed by historians:

Rob Nelson, Mining the Dispatch.[351]

Cameron Blevins, "Topic Modeling Martha Ballard's Diary" *Historying*, April 1, 2010.[352]

David J Newman and Sharon Block, "Probabilistic topic decomposition of an eighteenth century American newspaper," *Journal of the American Society for Information Science and Technology* vol. 57, no. 6 (April 1, 2006): 753-767.

Installing MALLET

There are many tools one could use to create topic models, but at the time of this writing (summer 2012) the simplest tool to run your text through is called MALLET.[353] MALLET uses an implementation of *Gibbs sampling*,[354] a statistical technique meant to quickly construct a sample distribution, to create its topic models. MALLET requires using the command line – we'll

[348] Scott Weingart, 'The Myth of Text Analytics and Unobtusive Measurement' (6 May 2012): http://www.scottbot.net/HIAL/?p=16713

[349] Andrew Gelman, 'How many zombies do you know? Using indirect survey methods to measure alien attacks and outbreaks of the undead', arXiv:1003.6087: http://arxiv.org/abs/1003.6087

[350] 'Latent Dirichlet allocation', *Wikipedia*: https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

[351] Rob Nelson, 'Mining the *Dispatch*': http://dsl.richmond.edu/dispatch/

[352] Cameron Blevins, 'Topic Modeling Martha Ballard's Diary', *Historying* (1 April 2010): http://www.cameronblevins.org/posts/topic-modeling-martha-ballards-diary/

[353] 'Machine Learning for LanguagE Toolkit': http://mallet.cs.umass.edu/index.php

[354] 'Gibbs sampling', *Wikipedia*: https://en.wikipedia.org/wiki/Gibbs_sampling

talk about that more in a moment, although you typically use the same few commands over and over.

The installation instructions are different for Windows and Mac. Follow the instructions appropriate for you below:

Windows Instructions

Go to the MALLET project page, and download MALLET.[355] (As of this writing, we are working with version 2.0.7.)

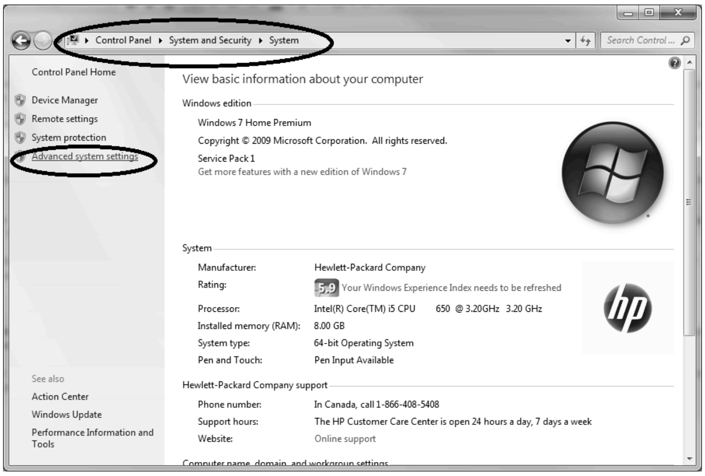
You will also need the Java developer's kit[356] – that is, not the regular Java that's on every computer, but the one that lets you program things. Install this on your computer.

Unzip MALLET into your `C:` directory . This is important: it cannot be anywhere else. You will then have a directory called `C:\mallet-2.0.7` or similar. For simplicity's sake, rename this directory just `mallet`.

MALLET uses an *environment variable* to tell the computer where to find all the various components of its processes when it is running. It's rather like a shortcut for the program. A programmer cannot know exactly where every user will install a program, so the programmer creates a variable in the code that will always stand in for that location. We tell the computer, once, where that location is by setting the environment variable. If you moved the program to a new location, you'd have to change the variable.

To create an environment variable in Windows 7, click on your `Start Menu -> Control Panel -> System -> Advanced System Settings` (Figures 1,2,3). Click new and type `MALLET_HOME` in the variable name box. It must be like this – all caps, with an underscore – since that is the shortcut that the programmer built into the program and all of its subroutines. Then type the exact path (location) of where you unzipped MALLET in the variable value, e.g., `c:\mallet`.

To see if you have been successful, please read on to the next section.



View basic information about your computer

Windows edition

Windows 7 Home Premium

Copyright © 2009 Microsoft Corporation. All rights reserved.

Service Pack 1

Get more features with a new edition of Windows 7

System

Manufacturer:	Hewlett-Packard Company
Rating:	5.9 Your Windows Experience Index needs to be refreshed
Processor:	Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz 3.20 GHz
Installed memory (RAM):	8.00 GB
System type:	64-bit Operating System
Pen and Touch:	Pen Input Available

Hewlett-Packard Company support

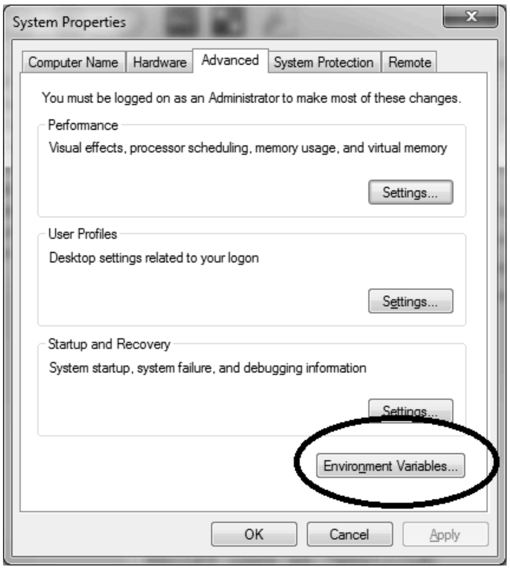
Phone number:	In Canada, call 1-866-408-5408
Support hours:	The HP Customer Care Center is open 24 hours a day, 7 days a week
Website:	Online support

Computer name, domain, and workgroup settings

[355] 'Machine Learning for LanguagE Toolkit': http://mallet.cs.umass.edu/index.php

[356] 'Java Developer's Kit': http://www.oracle.com/technetwork/java/javase/downloads/index.html

Advanced System Settings on Windows



You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

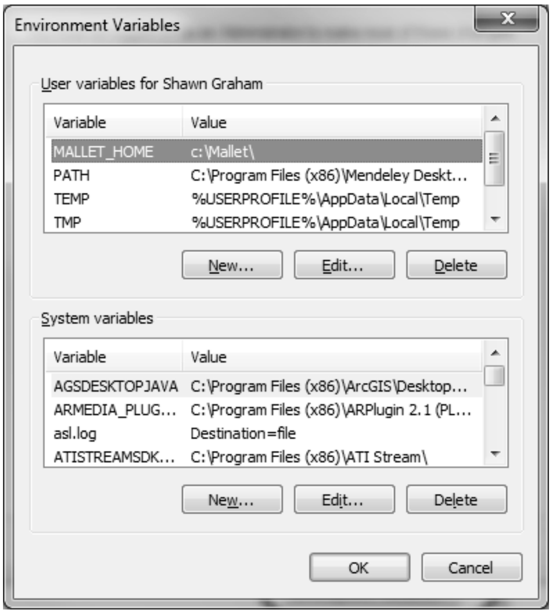
User Profiles

Desktop settings related to your logon

Startup and Recovery

System startup, system failure, and debugging information

Environment Variables Location



User variables for Shawn Graham

Variable	Value
MALLET_HOME	c:\Mallet\
PATH	C:\Program Files (x86)\Mendeley Deskt...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

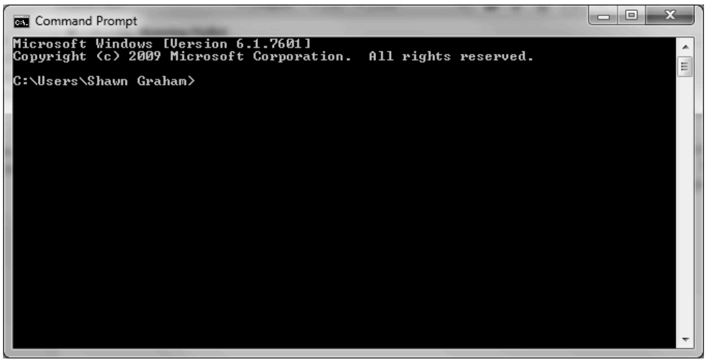
System variables

Variable	Value
AGSDESKTOPJAVA	C:\Program Files (x86)\ArcGIS\Desktop...
ARMEDIA_PLUG...	C:\Program Files (x86)\ARPlugin 2.1 (PL...
asl.log	Destination=file
ATISTREAMSDK...	C:\Program Files (x86)\ATI Stream\

Environment Variable

Running MALLET using the Command Line

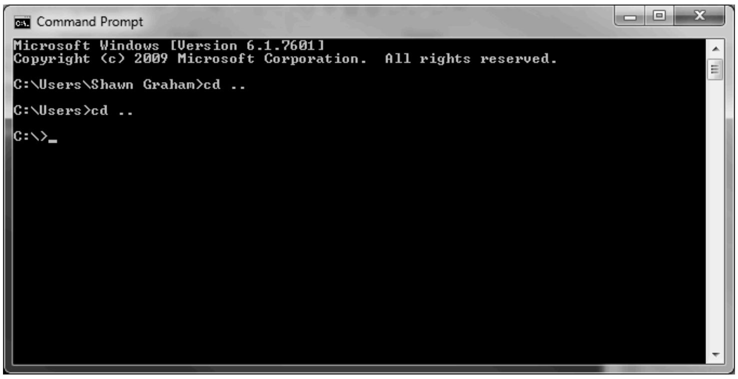
MALLET is run from the command line, also known as *Command Prompt* (Figure 4). If you remember MS-DOS, or have ever played with a Unix computer Terminal, this will be familiar. The command line is where you can type commands directly, rather than clicking on icons and menus.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Shawn Graham>
```

Command Prompt on Windows

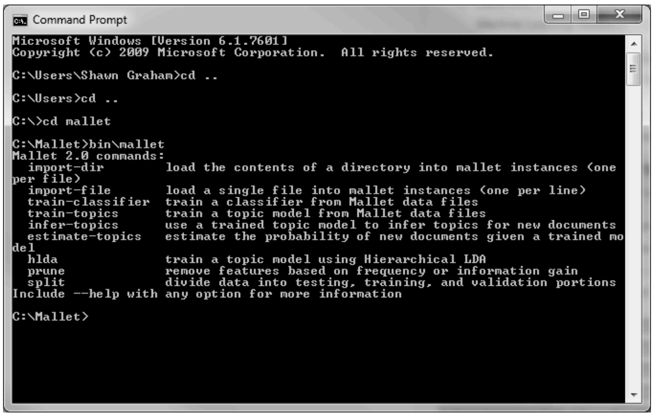
Click on your `Start Menu -> All Programs -> Accessories -> Command Prompt`.
You'll get the command prompt window, which will have a cursor at `c:\user\user>` (or similar; see Figure 4).
Type `cd ..` (That is: cd-space-period-period) to *change directory*. Keep doing this until you're at the `C:\` . (as in Figure 5)



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Shawn Graham>cd ..

C:\Users>cd ..

C:\>_
```

Navigating to the C: Directory in Command Prompt

Then type `cd mallet`and you are in the MALLETdirectory. Anything you type in the command prompt window is a *command*. There are commands like `cd` (change directory) and `dir` (list directory contents) that the computer understands. You have to tell the computer explicitly that 'this is a MALLET command' when you want to use MALLET. You do this by telling the computer to grab its instructions from the MALLET *bin*, a subfolder in MALLET that contains the core operating routines.
Type `bin\mallet` as in Figure 6. If all has gone well, you should be presented with a list of MALLET commands – congratulations! If you get an error message, check your typing. Did you use the wrong slash? Did you set up the environment variable correctly? Is MALLET located at `C:\mallet` ?



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Shawn Graham>cd ..

C:\Users>cd ..

C:\>cd mallet

C:\Mallet>bin\mallet
Mallet 2.0 commands:
  import-dir         load the contents of a directory into mallet instances (one
per file)
  import-file        load a single file into mallet instances (one per line)
  train-classifier   train a classifier from Mallet data files
  train-topics       train a topic model from Mallet data files
  infer-topics       use a trained topic model to infer topics for new documents
  estimate-topics    estimate the probability of new documents given a trained mo
del
  hlda               train a topic model using Hierarchical LDA
  prune              remove features based on frequency or information gain
  split              divide data into testing, training, and validation portions
Include --help with any option for more information

C:\Mallet>
```

Command Prompt MALLET Installed

You are now ready to skip ahead to the next section.

Mac Instructions

Many of the instructions for OS X installation are similar to Windows, with a few differences. In fact, it is a bit easier.

Download and install MALLET (*mallet-2.0.7.tar.gaz*as of Summer 2012).[357] Download the Java Development Kit.[358]

Unzip MALLET into a directory on your system (for ease of following along with this tutorial, your `/user/` directory works but anywhere is okay). Once it is unzipped, open up your Terminal window (in the `Applications` directory in your Finder. Navigate to the directory where you unzipped MALLET using the Terminal (it will be `mallet-2.0.7` . If you unzipped it into your `/user/` directory as was suggested in this lesson, you can navigate to the correct directory by typing `cd mallet-2.0.7`). cd is short for "change directory" when working in the Terminal.

The same command will suffice to run commands from this directory, except you need to append `./` (period-slash) before each command. This needs to be done before all MALLET commands when working on a Mac.

Going forward, the commands for MALLET on a Mac will be nearly identical to those on Windows, except for the direction of slashes (there are a few other minor differences that will be noted when they arise). If on Windows a command would be `\bin\mallet`, on a Mac you would instead type:

```
./bin/mallet
```

[357] 'MAchine Learning for LanguagE Toolkit': http://mallet.cs.umass.edu/download.php

[358] 'Java Development Kit': http://www.oracle.com/technetwork/java/javase/downloads/index.html

A list of commands should appear. If it does, congratulations – you've installed it correctly!

Typing in MALLET Commands

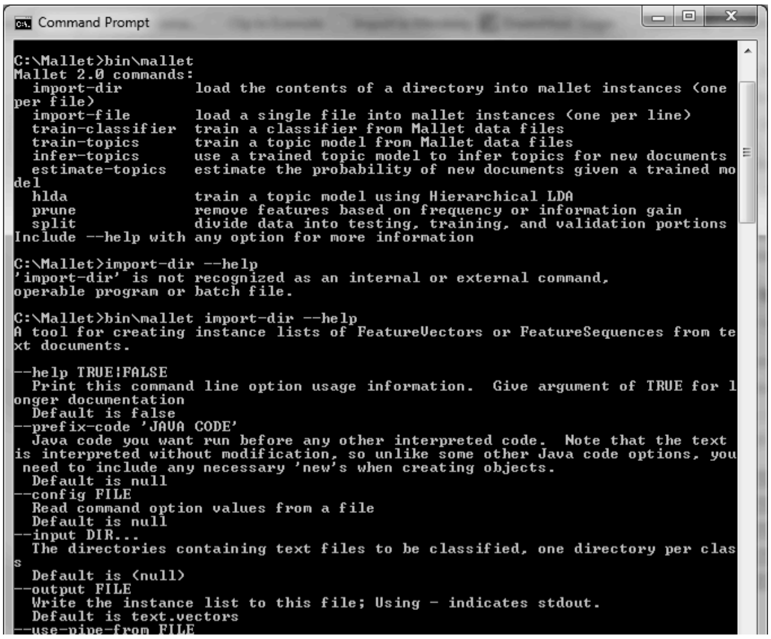
Now that you have MALLET installed, it is time to learn what commands are available to use with the program. There are nine MALLET commands you can use (see Figure 6 above). Sometimes you can combine multiple instructions. At the Command Prompt or Terminal (depending on your operating system), try typing:

```
import-dir --help
```

You are presented with the error message that `import-dir` is not recognized as an internal or external command, operable program, or batch file. This is because we forgot to tell the computer to look in the MALLET `bin` for it. Try again, with

```
bin\mallet import-dir --help
```

Remember, the direction of the slash matters (See Figure 7, which provides an entire transcript of what we have done so far in the tutorial). We checked to see that we had installed MALLET by typing in `bin\mallet`. We then made the mistake with `import-dir` a few lines further down. After that, we successfully called up the help file, which told us what `import-dir` does, and it listed all of the potential *parameters* you can set for this tool.



```
C:\Mallet>bin\mallet
Mallet 2.0 commands:
  import-dir        load the contents of a directory into mallet instances (one
per file)
  import-file       load a single file into mallet instances (one per line)
  train-classifier  train a classifier from Mallet data files
  train-topics      train a topic model from Mallet data files
  infer-topics      use a trained topic model to infer topics for new documents
  estimate-topics   estimate the probability of new documents given a trained mo
del
  hlda              train a topic model using Hierarchical LDA
  prune             remove features based on frequency or information gain
  split             divide data into testing, training, and validation portions
Include --help with any option for more information

C:\Mallet>import-dir --help
'import-dir' is not recognized as an internal or external command,
operable program or batch file.

C:\Mallet>bin\mallet import-dir --help
A tool for creating instance lists of FeatureVectors or FeatureSequences from te
xt documents.

--help TRUE|FALSE
  Print this command line option usage information.  Give argument of TRUE for l
onger documentation
  Default is false
--prefix-code 'JAVA CODE'
  Java code you want run before any other interpreted code.  Note that the text
is interpreted without modification, so unlike some other Java code options, you
 need to include any necessary 'new's when creating objects.
  Default is null
--config FILE
  Read command option values from a file
  Default is null
--input DIR...
  The directories containing text files to be classified, one directory per clas
s
  Default is (null)
--output FILE
  Write the instance list to this file; Using - indicates stdout.
  Default is text.vectors
--use-pipe-from FILE
```

The help menu in MALLET

Note: there is a difference in MALLET commands between a single hyphen and a double hyphen. A single hyphen is simply part of the name; it replaces a space (e.g., `import-dir` rather than import dir), since spaces offset multiple commands or parameters. These parameters let us tweak the file that is created when we import our texts into MALLET. A double hyphen (as with `–help` above) modifies, adds a sub-command, or specifies some sort of parameter to the command.

For Windows users, if you got the error *'exception in thread "main" java.lang.NoClassDefFoundError:'* it might be because you installed MALLET somewhere other than in the `C:\` directory. For instance, installing MALLET at `C:\Program Files\mallet` will produce this error message. The second thing to check is that your environment variable is set correctly. In either of these cases, check the Windows installation instructions and double check that you followed them properly.

Working with data

MALLET comes pre-packaged with sample `.txt` files with which you can practice. Type `dir` at the `C:\mallet> prompt`, and you are given the listing of the MALLET directory contents. One of those directories is called `sample-data`. You know it is a directory because it has the word <dir> beside it.

Type `cd sample-data`. Type `dir` again. Using what you know, navigate to first the `web` then the `en` directories. You can look inside these `.txt` files by typing the full name of the file (with extension).

Note that you cannot now run any MALLET commands from this directory. Try it:

```
bin\mallet import-dir --help
```

You get the error message. You will have to navigate back to the main MALLET folder to run the commands. This is because of the way MALLET and its components are structured.

Importing data

In the `sample data` directory, there are a number of `.txt` files. Each one of these files is a single document, the text of a number of different web pages. The entire folder can be considered to be a corpus of data. To work with this corpus and find out what the topics are that compose these individual documents, we need to transform them from several individual text files into a single MALLET format file. MALLET can import more than one file at a time. We can import the entire directory of text files using the `import` command. The commands below import the directory, turn it into a MALLET file, keep the original texts in the order in which they were listed, and strip out the *stop words* (words such as *and*, *the*, *but*, and *if* that occur in such frequencies that they obstruct analysis) using the default English `stop-words` dictionary. Try the following (swapping in the correct pathway to the sample data).

```
bin\mallet import-dir --input pathway\to\the\directory\with\the\files --outp
ut tutorial.mallet --keep-sequence --remove-stopwords
```

If you type `dir` now (or `ls` for Mac), you will find a file called `tutorial.mallet`. (If you get an error message, you can hit the cursor up key on your keyboard to recall the last command you typed, and look carefully for typos). This file now contains all of your data, in a format that MALLET can work with.

For Mac

Mac instructions are similar to those above for Windows, but keep in mind that Unix file paths (which are used by Mac) are different: for example, if the directory was in one's home directory, one would type

```
./bin/mallet import-dir --input /users/username/database/ --output tutorial.
mallet --keep-sequence --remove-stopwords
```

Issues with Big Data

If you're working with extremely large file collections – or indeed, very large files – you may run into issues with your *heap space*, your computer's working memory. This issue will initially arise during the import sequence, if it is relevant. By default, MALLET allows for 1GB of memory to be used. If you run into the following error message, you've run into your limit:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

If your system has more memory, you can try increasing the memory allocated to your *Java virtual machine*. To do so, you need to edit the code in the `mallet` file found in the `bin` subdirectory of your MALLET folder. Using Komodo Edit, (See Mac, Windows, Linux for installation instructions),[359] open the `Mallet.bat` file (`C:\Mallet\bin\mallet.bat`) if you are using Windows, or the `mallet` file (`~/Mallet/bin/mallet`) if you are using Linux or OS X.

Find the following line:

```
MEMORY=1g
```

You can then change the 1g value upwards – to 2g, 4g, or even higher depending on your system's RAM, which you can find out by looking up the machine's system information.

Save your changes. You should now be able to avoid the error. If not, increase the value again.

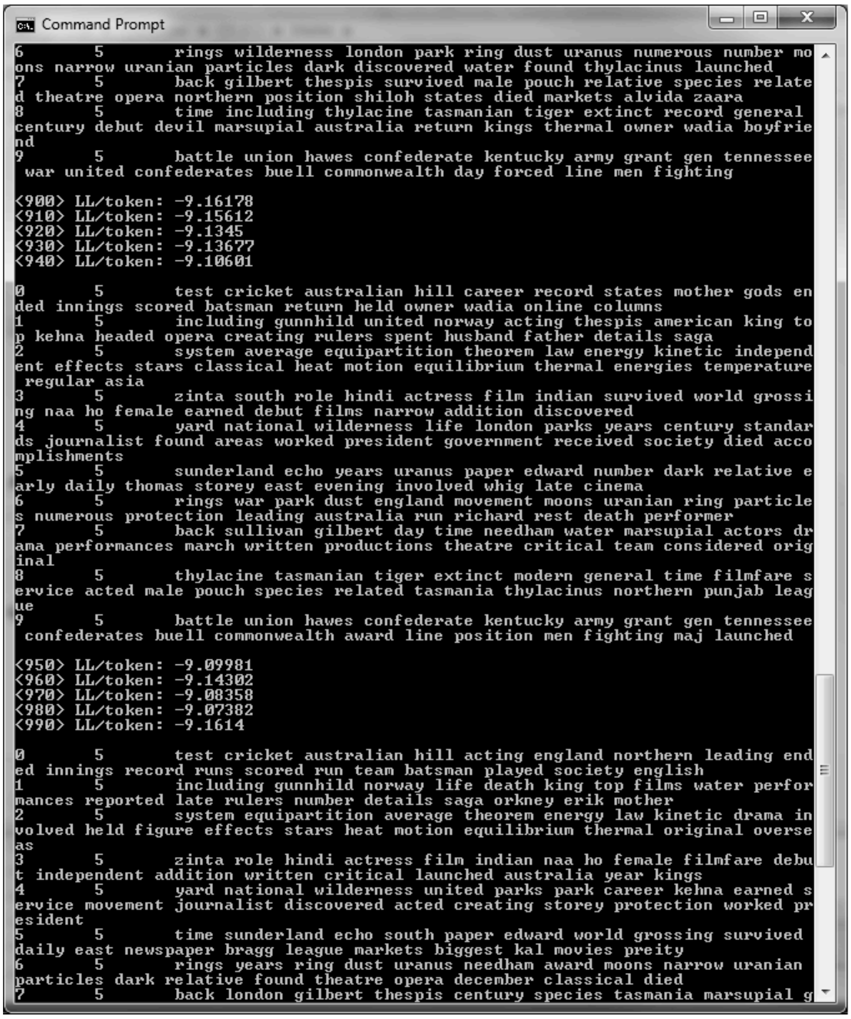
Your first topic model

At the command prompt in the MALLET directory, type:

```
bin\mallet train-topics  --input tutorial.mallet
```

[359] See: William J. Turkel and Adam Crymble, 'Setting Up an Integrated Development Environment for Python' (Mac, Linux, or Windows), *The Programming Historian* (2012).

This command opens your `tutorial.mallet` file, and runs the topic model routine on it using only the default settings. As it iterates through the routine, trying to find the best division of words into topics, your command prompt window will fill with output from each run. When it is done, you can scroll up to see what it was outputting (as in Figure 8).



```
6       5       rings wilderness london park ring dust uranus numerous number mo
ons narrow uranian particles dark discovered water found thylacinus launched
7       5       back gilbert thespis survived male pouch relative species relate
d theatre opera northern position shiloh states died markets alvida zaara
8       5       time including thylacine tasmanian tiger extinct record general
century debut devil marsupial australia return kings thermal owner wadia boyfrie
nd
9       5       battle union hawes confederate kentucky army grant gen tennessee
 war united confederates buell commonwealth day forced line men fighting

<900> LL/token: -9.16178
<910> LL/token: -9.15612
<920> LL/token: -9.1345
<930> LL/token: -9.13677
<940> LL/token: -9.10601

0       5       test cricket australian hill career record states mother gods en
ded innings scored batsman return held owner wadia online columns
1       5       including gunnhild united norway acting thespis american king to
p kehna headed opera creating rulers spent husband father details saga
2       5       system average equipartition theorem law energy kinetic independ
ent effects stars classical heat motion equilibrium thermal energies temperature
 regular asia
3       5       zinta south role hindi actress film indian survived world grossi
ng naa ho female earned debut films narrow addition discovered
4       5       yard national wilderness life london parks years century standar
ds journalist found areas worked president government received society died acco
mplishments
5       5       sunderland echo years uranus paper edward number dark relative e
arly daily thomas storey east evening involved whig late cinema
6       5       rings war park dust england movement moons uranian ring particle
s numerous protection leading australia run richard rest death performer
7       5       back sullivan gilbert day time needham water marsupial actors dr
ama performances march written productions theatre critical team considered orig
inal
8       5       thylacine tasmanian tiger extinct modern general time filmfare s
ervice acted male pouch species related tasmania thylacinus northern punjab leag
ue
9       5       battle union hawes confederate kentucky army grant gen tennessee
 confederates buell commonwealth award line position men fighting maj launched

<950> LL/token: -9.09981
<960> LL/token: -9.14302
<970> LL/token: -9.08358
<980> LL/token: -9.07382
<990> LL/token: -9.1614

0       5       test cricket australian hill acting england northern leading end
ed innings record runs scored run team batsman played society english
1       5       including gunnhild norway life death king top films water perfor
mances reported late rulers number details saga orkney erik mother
2       5       system equipartition average theorem energy law kinetic drama in
volved held figure effects stars heat motion equilibrium thermal original overse
as
3       5       zinta role hindi actress film indian naa ho female filmfare debu
t independent addition written critical launched australia year kings
4       5       yard national wilderness united parks park career kehna earned s
ervice movement journalist discovered acted creating storey protection worked pr
esident
5       5       time sunderland echo south paper edward world grossing survived
daily east newspaper bragg league markets biggest kal movies preity
6       5       rings years ring dust uranus needham award moons narrow uranian
particles dark relative found theatre opera december classical died
7       5       back london gilbert thespis century species tasmania marsupial g
```

Basic Topic Model Output

The computer is printing out the key words, the words that help define a statistically significant topic, per the routine. In Figure 8, the first topic it prints out might look like this (your key words might look a bit different):

```
0    5    test cricket Australian hill acting England northern leading ended
 innings record runs scored run team batsman played society English
```

If you are a fan of cricket, you will recognize that all of these words could be used to describe a cricket match. What we are dealing with here is a topic related to Australian cricket. If you go to `C:\mallet\sample-data\web\en\hill.txt`, you will see that this file is a brief biography of the noted Australian cricketer Clem Hill. The 0 and the 5 we will talk about later in the lesson. Note that MALLET includes an element of randomness, so the keyword lists will look different every time the program is run, even if on the same set of data.

Go back to the main MALLET directory, and type `dir`. You will see that there is no output file. While we successfully created a topic model, we did not save the output! At the command prompt, type

```
bin\mallet train-topics  --input tutorial.mallet --num-topics 20 --output-st
ate topic-state.gz --output-topic-keys tutorial_keys.txt --output-doc-topics
 tutorial_compostion.txt
```

Here, we have told MALLET to create a topic model (`train-topics`) and everything with a double hyphen afterwards sets different parameters

This command

opens your `tutorial.mallet` file

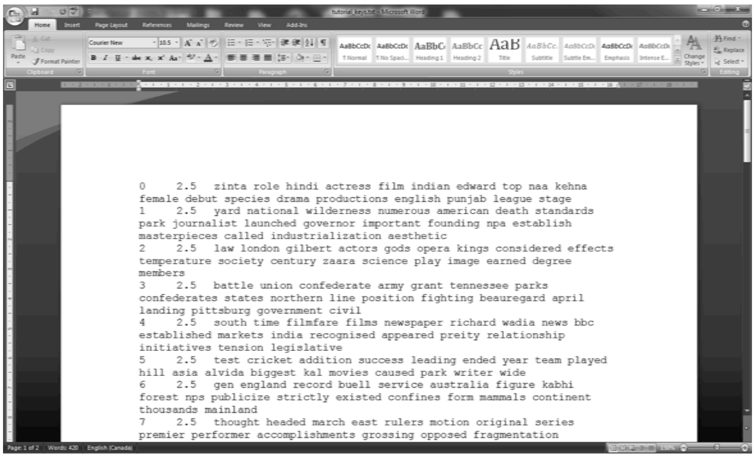
trains MALLET to find 20 topics

outputs every word in your corpus of materials and the topic it belongs to into a compressed file (`.gz`; see www.gzip.org on how to unzip this)

outputs a text document showing you what the top key words are for each topic (`tutorial_keys.txt`)

and outputs a text file indicating the breakdown, by percentage, of each topic within each original text file you imported (`tutorial_composition.txt`). (To see the full range of possible parameters that you may wish to tweak, type `bin\mallet train-topics –help` at the prompt.)

Type `dir`. Your outputted files will be at the bottom of the list of files and directories in `C:\Mallet`. Open `tutorial_keys.txt` in a word processor (Figure 9). You are presented with a series of paragraphs. The first paragraph is topic 0; the second paragraph is topic 1; the third paragraph is topic 2; etc. (The output begins counting at 0 rather than 1; so if you ask it to determine 20 topics, your list will run from 0 to 19). The second number in each paragraph is the *Dirichlet parameter* for the topic. This is related to an option which we did not run, and so its default value was used (this is why every topic in this file has the number 2.5).



```
0     2.5   zinta role hindi actress film indian edward top naa kehna
female debut species drama productions english punjab league stage
1     2.5   yard national wilderness numerous american death standards
park journalist launched governor important founding npa establish
masterpieces called industrialization aesthetic
2     2.5   law london gilbert actors gods opera kings considered effects
temperature society century zaara science play image earned degree
members
3     2.5   battle union confederate army grant tennessee parks
confederates states northern line position fighting beauregard april
landing pittsburg government civil
4     2.5   south time filmfare films newspaper richard wadia news bbc
established markets india recognised appeared preity relationship
initiatives tension legislative
5     2.5   test cricket addition success leading ended year team played
hill asia alvida biggest kal movies caused park writer wide
6     2.5   gen england record buell service australia figure kabhi
forest nps publicize strictly existed confines form mammals continent
thousands mainland
7     2.5   thought headed march east rulers motion original series
premier performer accomplishments grossing opposed fragmentation
```

Keywords shown in a Word Processor

If when you ran the topic model routine you had included

```
--optimize-interval 20
```

as below

```
bin\mallet train-topics  --input tutorial.mallet  --num-topics 20 --optimiz
e-interval 20 --output-state topic-state.gz  --output-topic-keys tutorial_ke
ys.txt --output-doc-topics tutorial_composition.txt
```

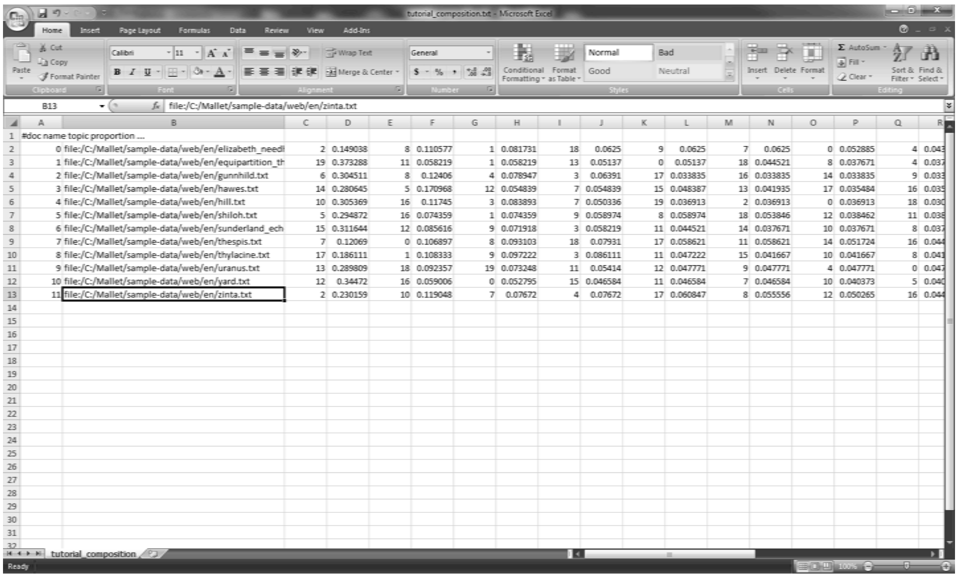
the output might look like this:

```
0 0.02995 xi ness regular asia online cinema established alvida acclaim veen
r commercial
```

That is, the first number is the topic (topic 0), and the second number gives an indication of the *weight* of that topic. In general, including `-optimize-interval` leads to better topics.

The composition of your documents

What topics compose your documents? The answer is in the `tutorial_composition.txt` file. To stay organized, import the `tutorial_composition.txt` file into a spreadsheet (Excel, Open Office, etc). You will have a spreadsheet with a #doc, source, topic, proportion columns. All subsequent columns run topic, proportion, topic, proportion, etc., as in figure 10.



B13 file:/C:/Mallet/sample-data/web/en/zinta.txt

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	#doc name topic proportion																	
2	0	file:/C:/Mallet/sample-data/web/en/elizabeth_needl	2	0.149038	8	0.110577	1	0.081731	18	0.0625	9	0.0625	7	0.0625	0	0.052885	4	0.043
3	1	file:/C:/Mallet/sample-data/web/en/equipartition_th	19	0.373288	11	0.058219	1	0.058219	13	0.05137	0	0.05137	18	0.044521	8	0.037671	4	0.037
4	2	file:/C:/Mallet/sample-data/web/en/gunnhild.txt	6	0.304511	8	0.12406	4	0.078947	3	0.06391	17	0.033835	16	0.033835	14	0.033835	9	0.033
5	3	file:/C:/Mallet/sample-data/web/en/hawes.txt	14	0.280645	5	0.170968	12	0.054839	7	0.054839	15	0.048387	13	0.041935	17	0.035484	16	0.035
6	4	file:/C:/Mallet/sample-data/web/en/hill.txt	10	0.305369	16	0.11745	3	0.083893	7	0.050336	19	0.036913	2	0.036913	0	0.036913	18	0.030
7	5	file:/C:/Mallet/sample-data/web/en/shiloh.txt	5	0.294872	16	0.074359	1	0.074359	9	0.058974	8	0.058974	18	0.053846	12	0.038462	11	0.038
8	6	file:/C:/Mallet/sample-data/web/en/sunderland_ech	15	0.311644	12	0.085616	9	0.071918	3	0.058219	11	0.044521	14	0.037671	10	0.037671	8	0.037
9	7	file:/C:/Mallet/sample-data/web/en/thespis.txt	7	0.12069	0	0.106897	8	0.093103	18	0.07931	17	0.058621	11	0.058621	14	0.051724	16	0.044
10	8	file:/C:/Mallet/sample-data/web/en/thylacine.txt	17	0.186111	1	0.108333	9	0.097222	3	0.086111	11	0.047222	15	0.041667	10	0.041667	8	0.041
11	9	file:/C:/Mallet/sample-data/web/en/uranus.txt	13	0.289809	18	0.092357	19	0.073248	11	0.05414	12	0.047771	9	0.047771	4	0.047771	0	0.047
12	10	file:/C:/Mallet/sample-data/web/en/yard.txt	12	0.34472	16	0.059006	0	0.052795	15	0.046584	11	0.046584	7	0.046584	10	0.040373	5	0.040
13	11	file:/C:/Mallet/sample-data/web/en/zinta.txt	2	0.230159	10	0.119048	7	0.07672	4	0.07672	17	0.060847	8	0.055556	12	0.050265	16	0.044

Topic Composition

You can see that doc# 0 (ie, the first document loaded into MALLET), `elizabeth_needham.txt` has topic 2 as its principal topic, at about 15%; topic 8 at 11%, and topic 1 at 8%. As we read along that first column of topics, we see that `zinta.txt` also has topic 2 as its largest topic, at 23%.

The topic model suggests a connection between these two documents that you might not at first have suspected.

If you have a corpus of text files that are arranged in chronological order (e.g., `1.txt` is earlier than `2.txt`), then you can graph this output in your spreadsheet program, and begin to see changes over time, as Robert Nelson has done in Mining the Dispatch.[360]

How do you know the number of topics to search for? Is there a *natural* number of topics? What we have found is that one has to run the train-topics with varying numbers of topics to see how the composition file breaks down. If we end up with the majority of our original texts all in a very limited number of topics, then we take that as a signal that we need to increase the number of topics; the settings were too coarse. There are computational ways of searching for this, including using MALLETs `hlda command`, but for the reader of this tutorial, it is probably just quicker to cycle through a number of iterations (but for more see Griffiths, T. L., & Steyvers, M. (2004). Finding scientific topics. Proceedings of the National Academy of Science, 101, 5228-5235).

Getting your own texts into MALLET

The `sample data` folder in MALLET is your guide to how you should arrange your texts. You want to put everything you wish to topic model into a single folder within `c:\mallet`, ie `c:\mallet\mydata`. Your texts should be in `.txt` format (that is, you create them with Notepad, or in Word choose `Save As -> MS Dos text`). You have to make some decisions. Do you want to explore topics at a paragraph by paragraph level? Then each `txt` file should contain one paragraph. Things like page numbers or other identifiers can be indicated in the name you give the file, e.g., `pg32_paragraph1.txt`. If you are working with a diary, each text file might be a single entry, e.g., `april_25_1887.txt`. (Note that when naming folders or files, do not leave spaces in the name. Instead use underscores to represent spaces). If the texts that you are interested in are on the web, you might be able to automate this process.[361]

Further Reading about Topic Modeling

To see a fully worked out example of topic modeling with a body of materials culled from webpages, see Mining the Open Web with Looted Heritage Draft.[362]

[360] Robert Nelson, 'Mining the *Dispatch*', http://dsl.richmond.edu/dispatch/

[361] See Shawn Graham, 'Mining a Day of Archaeology', *Electric Archaeology* (9 July 2012): http://electricarchaeology.ca/2012/07/09/mining-a-day-of-archaeology/

[362] Shawn Graham, 'Mining the Open Web with 'Looted Heritage' – Draft', *Electric Archaeology* (8 June 2012), http://electricarchaeology.ca/2012/06/08/mining-the-open-web-with-looted-heritage-draft/

You can grab the data for yourself at Figshare.com,[363] which includes a number of `.txt` files. Each individual `.txt` file is a single news report.

For extensive background and bibliography on topic modeling you may wish to begin with Scott Weingart's Guided Tour to Topic Modeling[364]
Ted Underwood's 'Topic modeling made just simple enough' is an important discussion on interpreting the meaning of topics.[365]
Lisa Rhody's post on interpreting topics is also illuminating. 'Some Assembly Required' *Lisa @ Work* August 22, 2012.[366]
Clay Templeton, 'Topic Modeling in the Humanities: An Overview | Maryland Institute for Technology in the Humanities', n.d.[367]
David Blei, Andrew Ng, and Michael Jordan, 'Latent dirichlet allocation,' The Journal of Machine Learning Research 3 (2003).[368]
Finally, also consult David Mimno's bibliography of topic modeling articles.[369] They're tagged by topic to make finding the right one for a particular application that much easier. Also take a look at his recent article on Computational Historiography from *ACM Transactions on Computational Logic*[370] which goes through a hundred years of Classics journals to learn something about the field. While the article should be read as a good example of topic modeling, his 'Methods' section is especially important, in that it discusses preparing text for this sort of analysis.

About the Author

Shawn Graham is associate professor of digital humanities and history at Carleton University. Scott Weingart is a historian of science and doctoral candidate at Indiana University. Ian Milligan is an assistant professor of history at the University of Waterloo.

[363] Shawn Graham, 'Full Text of 207 Reports from 1st Quarter of 2012 on Looted Heritage', *Figshare*: https://figshare.com/articles/looted_heritage_reports_txt.zip/91828

[364] Scott Weingart, 'Topic Modelilng for Humanists: A Guided Tour' (25 July 2012): http://www.scottbot.net/HIAL/?p=19113

[365] Ted Underwood, 'Topic modeling made just simple enough', *The Stone and the Shell* (7 April 2012): http://tedunderwood.com/2012/04/07/topic-modeling-made-just-simple-enough/

[366] Lisa Marie Rhody, 'Some Assembly Required: Understanding and Interpreting Topics in LDA Models of Figurative Language' (22 August 2012): http://www.lisarhody.com/some-assembly-required/

[367] Clay Templeton, 'Topic Modeling in the Humanities: An Overview': http://mith.umd.edu/topic-modeling-in-the-humanities-an-overview/

[368] David Blei, Andrew Ng, and Michael Jordan, 'Latent dirichlet allocation', *The Journal of Machine Learning Research*, vol. 3 (2003): http://dl.acm.org/citation.cfm?id=944937

[369] David Mimno, 'Topic Modeling Bibliography': http://mimno.infosci.cornell.edu/topics.html

[370] David Mimno, 'Computational Historiography: Data Mining in a Century of Classics Journals': http://www.perseus.tufts.edu/publications/02-jocch-mimno.pdf

33. Corpus Analysis with Antconc

Heather Froehlich – 2015

Introduction

Corpus analysis is a form of text analysis which allows you to make comparisons between textual objects at a large scale (so-called 'distant reading'). It allows us to see things that we don't necessarily see when reading as humans. If you've got a collection of documents, you may want to find patterns of grammatical use, or frequently recurring phrases in your corpus. You also may want to find statistically likely and/or unlikely phrases for a particular author or kind of text, particular kinds of grammatical structures or a lot of examples of a particular concept across a large number of documents in context. Corpus analysis is especially useful for testing intuitions about texts and/or triangulating results from other digital methods.

By the end of this tutorial, you will be able to:

create/download a corpus of texts
conduct a keyword-in-context search
identify patterns surrounding a particular word
use more specific search queries
look at statistically significant differences between corpora
make multi-modal comparisons using corpus lingiustic methods

You have done this sort of thing before, if you have ever...

searched in a PDF or a word doc for all examples a specific term
Used Voyant Tools[371] for looking at patterns in one text
Followed Programming Historian's Introduction to Python tutorials

In many ways Voyant is a gateway into conducting more sophisticated, replicable analysis, as the DIY aesthetic of Python or R scripting may not appeal to everyone. AntConc[372] fills this void by being a standalone software package for linguistic analysis of texts, freely available for Windows, Mac OS, and Linux and is highly maintained by its creator, Laurence Anthony. There are other concordance software packages

[371] 'Voyant Tools': http://voyant-tools.org/

[372] 'AntConc': http://www.laurenceanthony.net/software/antconc/

available, but it is freely available across platforms and very well maintained. See the concordance bibliography for other resources.[373]

This tutorial explores several different ways to approach a corpus of texts. It's important to note that corpus linguistic approaches are rarely, if ever, a one-size-fits all affair. So, as you go through each step, it's worth thinking about what you're doing and how it can help you answer a specific question with your data. Although I present this tutorial in a building-block approach of 'do this then that to achieve x', it's not always necessary to follow the exact order outlined here. This lessons provides an outline of some of the methods available, rather than a recipe for success.

Tutorial downloads

1. Software: AntConc.[374]
   Unzip the download if necessary, and launch the application. Screen shots below may vary slightly from the version you have (and by operationg system, of course), but the procedures are more or less the same across platforms and recent versions of AntConc. This tutorial is written with a (much older) version of AntConc in mind, as I find it easier to use in an introductory context. You are welcome to use the most recent version, but if you wish to follow along with the screenshots provided, you can download the version used here, version 3.2.4.

2. Sample Corpus: Download the zip file of movie reviews: (https://db.tt/2PsC23px).

A broad outline of this tutorial:

Working with plain text files
The AntConc user interface, loading corpora
Keyword-in-context searching
Advanced keyword-in-context searching
Collocates and word lists
Comparing corpora
Discussion: Making meaningful comparisons
Further resources

Working with Plain Text Files

Antconc works only with plain-text files with the file appendix .txt (eg Hamlet.txt).

Antconc **will not** read .doc, .docx, .pdf, files. You will need to convert these into .txt files.

It will read XML files that are saved as .txt files (it's OK if you don't know what an XML file is).

[373] Heather Froehlich, 'An Introductory Bibliography to Corpus Linguistics': http://hfroehli.ch/2014/05/11/intro-bibliography-corpus-linguistics/

[374] 'AntConc': http://www.laurenceanthony.net/software/antconc/

Visit your favorite website for news, and navigate to a news article (doesn't matter which one, as long as it is primarily text). Highlight all text in the article (header, byline, etc), and right-click "copy".

Open a text editor such as Notepad (on Windows) or TextEdit (on Mac) and paste in your text.

Other free options for text editors include Notepad++ (Windows)[375] or TextWrangler (Mac),[376] which offer more advanced features, and are especially good for doing a lot of text clean-up. By text clean-up, I mean removing extratextual information such as "boilerplate", which appears regularly throughout. If you keep this information, it's going to throw your data off; text analysis software will address these words in word counts, statistical analyses, and lexical relationships. For example, you might want to remove standard headers and footers which will appear on every page. Please see "Cleaning Data with OpenRefine"[377] for more on how to automate this task. On smaller corpora it may be more feasible to do this yourself, plus you'll get a much better sense of your corpus this way.

Save the article as a .txt file to the desktop. You may want to do some follow-up text cleanup on other information, such as author by-line or title (remove them, then save the file again.) Remember that anything you leave in the text file can and will be addressed by text analysis software.

Go to your desktop and check to see you can find your text file.

Repeating this a lot is how you would build a corpus of plain text files; this process is called *corpus construction*, which very often involves addressing questions of sampling, representativeness and organization. Remember, *each file you want to use in your corpus must be a plain text file for Antconc to use it.* It is customary to name files with the .txt suffix so that you know what kind of file it is.

As you might imagine, it can be rather tedious to build up a substantial corpus one file at a time, especially if you intend to process a large set of documents. It is very common, therefore, to use webscraping (using a small program to automatically grab files from the web for you) to construct your corpus. To learn more about the concepts and techniques for webscraping, see the *Programming Historian* tutorials scraping with Beautiful Soup and automatic downloading with wget.[378] Rather than build a corpus one document at a time, we're going to use a prepared corpus of positive and negative movie reviews, borrowed from the Natural Language Processing Toolkit.[379] The NLTK movie review corpus has 2000 reviews, organized by

[375] 'Notepad++': https://notepad-plus-plus.org/

[376] 'TextWrangler': http://www.barebones.com/products/textwrangler/

[377] Seth van Hooland, Ruben Verborgh, and Max de Wilde, 'Cleaning Data with OpenRefine', *The Programming Historian* (2013).

[378] Jeri Wieringa, 'Intro to Beautiful Soup' (2012); Ian Milligan 'Automated Downloading with Wget', *The Programming Historian* (2012).

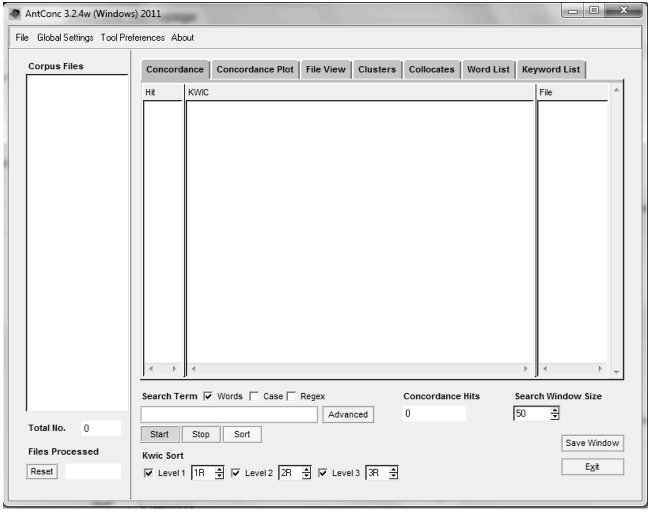
[379] 'Natural Language Toolkit': http://www.nltk.org/

positive and negative outcomes; today we will be addressing a small subset of them (200 positive, 200 negative).

Corpus construction is a subfield in its own right. Please see Representativeness in Corpus Design," *Literary and Linguistic Computing*, 8 (4): 243-257 and *Developing Linguistic Corpora: a Guide to Good Practice* for more information.[380]

Getting Started with AntConc: The AntConc user interface, loading corpora

When AntConc launches, it will look like this.



Total No. 0

Concordance Hits 0

Search Window Size 50

Search Term ☑ Words ☐ Case ☐ Regex

Kwic Sort ☑ Level 1 1R ☑ Level 2 2R ☑ Level 3 3R

On the left-hand side, there is a window to see all corpus files loaded (which we'll use momentarily).

There are 7 tabs across the top:

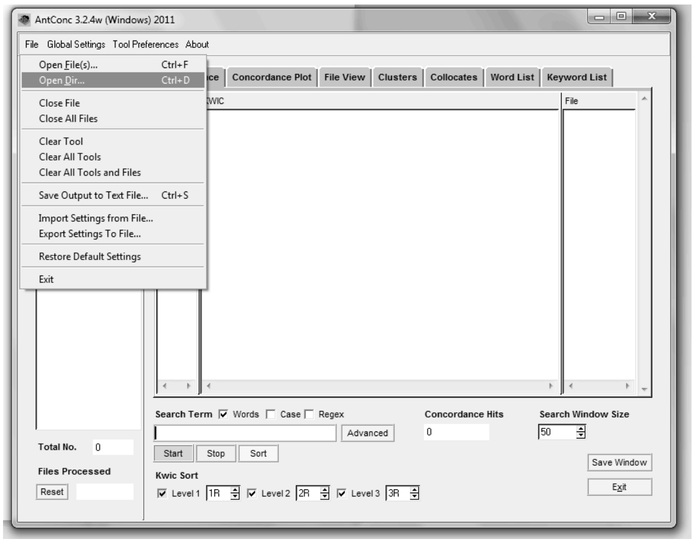
1. **Concordance**: This will show you what's known as a Keyword in Context view (abbreviated KWIC, more on this in a minute), using the search bar below it.
2. **Concordance Plot**: This will show you a very simple visualization of your KWIC search, where each instance will be represented as a little black line from beginning to end of each file containing the search term.
3. **File View**: This will show you a full file view for larger context of a result.
4. **Clusters**: This view shows you words which very frequently appear together.
5. **Collocates**: Clusters show us words which _definitely _appear together in a corpus; collocates show words which are statistically likely to appear together.
6. **Word list**: All the words in your corpus.
7. **Keyword List**: This will show comparisons between two corpora.

[380] Douglas Biber, 'Representativeness in Corpus Design', *Literary & Linguistic Computing*, vol. 8, no. 4 (1993), 243-257; Martin Wynne, 'Developing Linguistic Corpora: A Guide to Good Practice'.

As an introduction, this tutortial barely scratches the surface of what you can do with AntConc. We will focus on the Concordance, Collocates, Keywords, and Word List functions.

Loading Corpora

Like opening a file elsewhere, we're going to start with File > Open, but instead of opening just ONE file we want to open the directory of all our files. AntConc allows you to open entire directories, so if you're comfortable with this concept, you can just open the folder 'all reviews' and jump to Basic Analysis, below



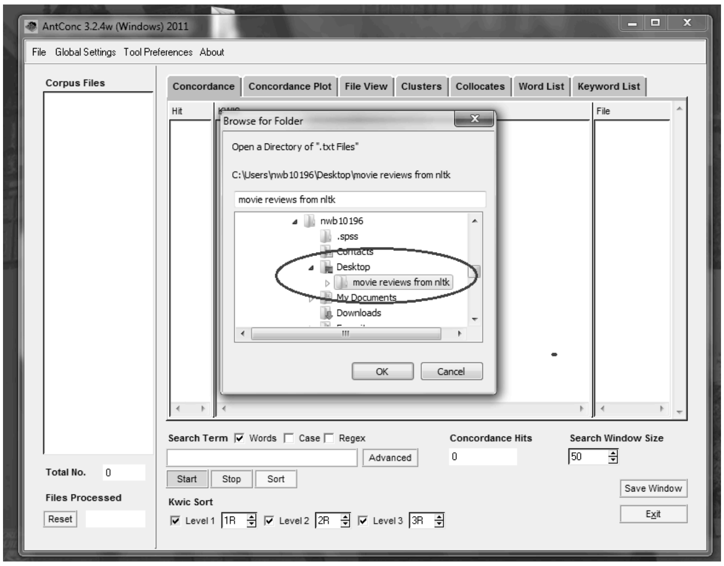
Concordance Hits: 0

Search Window Size: 50

Kwic Sort: ☑ Level 1 1R ☑ Level 2 2R ☑ Level 3 3R

Total No. 0

Remember we've put our files on the desktop, so navigate there in the dropdown menu.

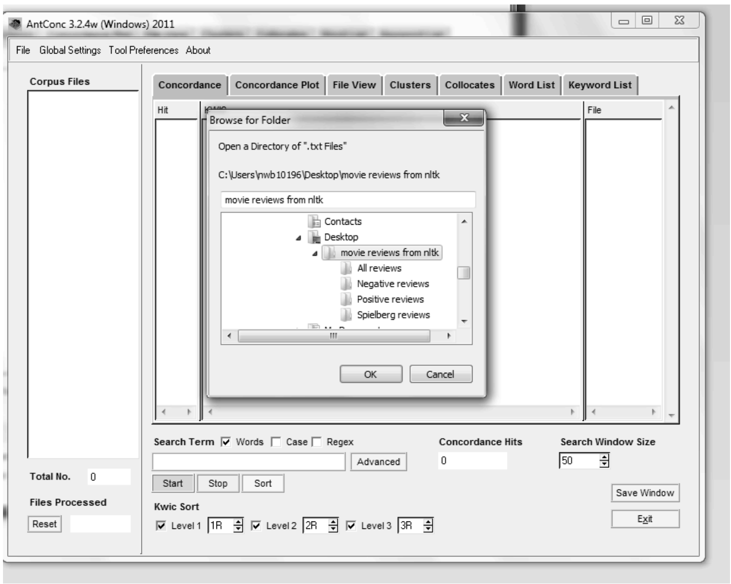


Open a Directory of "*.txt Files"

C:\Users\nwb10196\Desktop\movie reviews from nltk

movie reviews from nltk

From the Desktop you want to navigate to our folder "movie reviews from nltk":



AntConc 3.2.4w (Windows) 2011

Browse for Folder

Open a Directory of ".txt Files"

C:\Users\nwb10196\Desktop\movie reviews from nltk

movie reviews from nltk

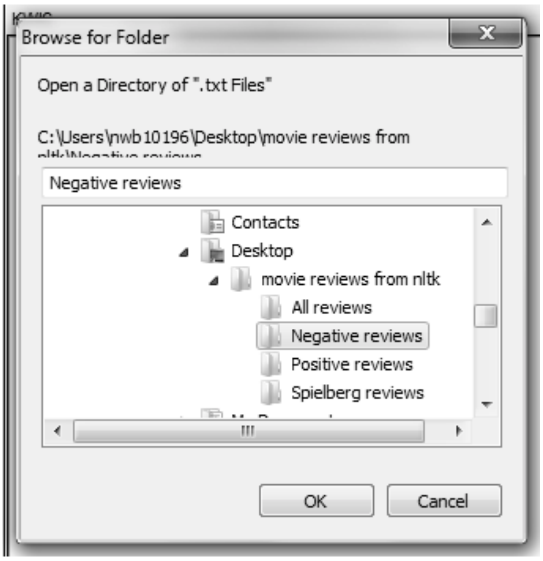
Total No. 0

Concordance Hits 0

Search Window Size 50

☑ Level 1 1R ☑ Level 2 2R ☑ Level 3 3R

First you will select "Negative Reviews" and hit OK. 200 texts should load in the lefthand column Corpus Files – watch the Total No. box!



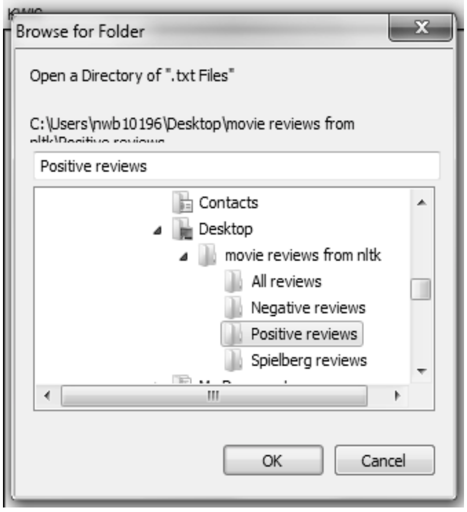
Browse for Folder

Open a Directory of ".txt Files"

C:\Users\nwb10196\Desktop\movie reviews from nltk\Negative reviews

Negative reviews

Then you're going to repeat the process to load the folder "Positive Reviews". You should now have 400 texts in the Corpus Files column.

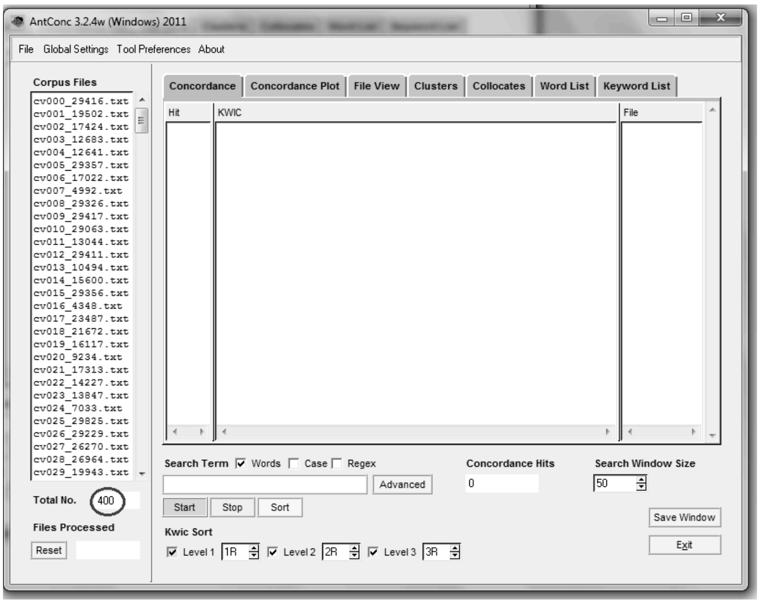


Browse for Folder

Open a Directory of "*.txt Files"

C:\Users\nwb10196\Desktop\movie reviews from nltk\Positive reviews

Positive reviews



AntConc 3.2.4w (Windows) 2011

Corpus Files

```
cv000_29416.txt
cv001_19502.txt
cv002_17424.txt
cv003_12683.txt
cv004_12641.txt
cv005_29357.txt
cv006_17022.txt
cv007_4992.txt
cv008_29326.txt
cv009_29417.txt
cv010_29063.txt
cv011_13044.txt
cv012_29411.txt
cv013_10494.txt
cv014_15600.txt
cv015_29356.txt
cv016_4348.txt
cv017_23487.txt
cv018_21672.txt
cv019_16117.txt
cv020_9234.txt
cv021_17313.txt
cv022_14227.txt
cv023_13847.txt
cv024_7033.txt
cv025_29825.txt
cv026_29229.txt
cv027_26270.txt
cv028_26964.txt
cv029_19943.txt
```

Total No. 400

Search Term ☑ Words □ Case □ Regex

Concordance Hits 0

Search Window Size 50

Kwic Sort ☑ Level 1 1R ☑ Level 2 2R ☑ Level 3 3R

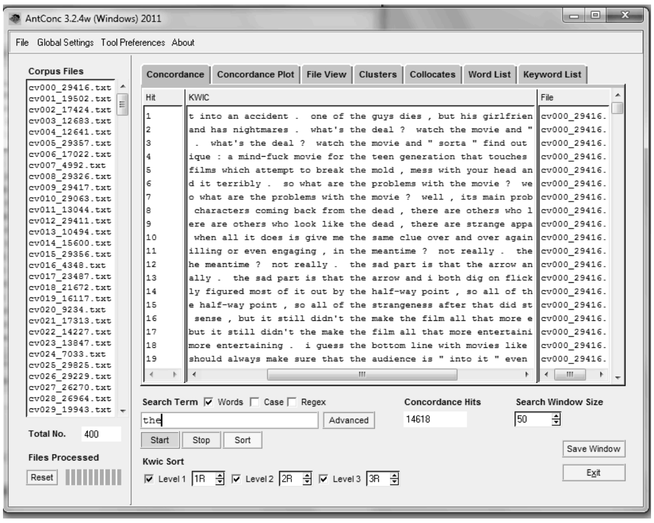
all reviews loaded

Searching Keywords in Context

Start with a basic search

One of the things corpus tools like Antconc are very good at are finding patterns in language which we have a hard time identifying as readers. Small boring words like *the, I, he, she, a, an, is, have, will* are especially difficult to keep track of as readers, because they're so common, but computers happen to be very good at them. These words are called function words, though they commonly known as 'stopwords' in digital humanities; they are often very distinct measures of authorial and generic style. As a result, they can be quite powerful search terms on their own or when combined with more content-driven terms, helping the researcher identify patterns they may not have been aware of previously.

In the search box at the bottom, type the and click "start". The Concordance view will show you every time the word the appears in our corpus of movie reviews, and some context for it. This is called a "Key Words in Context" viewer.



Corpus Files

cv000_29416.txt
cv001_19502.txt
cv002_17424.txt
cv003_12683.txt
cv004_12641.txt
cv005_29357.txt
cv006_17022.txt
cv007_4992.txt
cv008_29326.txt
cv009_29417.txt
cv010_29063.txt
cv011_13044.txt
cv012_29411.txt
cv013_10494.txt
cv014_15600.txt
cv015_29356.txt
cv016_4348.txt
cv017_23487.txt
cv018_21672.txt
cv019_16117.txt
cv020_9234.txt
cv021_17313.txt
cv022_14227.txt
cv023_13847.txt
cv024_7033.txt
cv025_29825.txt
cv026_29229.txt
cv027_26270.txt
cv028_26964.txt
cv029_19943.txt

Total No. 400

Files Processed

Hit	KWIC	File
1	t into an accident . one of the guys dies , but his girlfrien	cv000_29416.
2	and has nightmares . what's the deal ? watch the movie and "	cv000_29416.
3	. what's the deal ? watch the movie and " sorta " find out	cv000_29416.
4	ique : a mind-fuck movie for the teen generation that touches	cv000_29416.
5	films which attempt to break the mold , mess with your head an	cv000_29416.
6	d it terribly . so what are the problems with the movie ? we	cv000_29416.
7	o what are the problems with the movie ? well , its main prob	cv000_29416.
8	characters coming back from the dead , there are others who l	cv000_29416.
9	ere are others who look like the dead , there are strange appa	cv000_29416.
10	when all it does is give me the same clue over and over again	cv000_29416.
11	illing or even engaging , in the meantime ? not really . the	cv000_29416.
12	he meantime ? not really . the sad part is that the arrow an	cv000_29416.
13	ally . the sad part is that the arrow and i both dig on flick	cv000_29416.
14	ly figured most of it out by the half-way point , so all of th	cv000_29416.
15	e half-way point , so all of the strangeness after that did st	cv000_29416.
16	sense , but it still didn't the make the film all that more e	cv000_29416.
17	but it still didn't the make the film all that more entertaini	cv000_29416.
18	more entertaining . i guess the bottom line with movies like	cv000_29416.
19	should always make sure that the audience is " into it " even	cv000_29416.

Search Term ☑ Words ☐ Case ☐ Regex

the

Concordance Hits 14618

Search Window Size 50

Kwic Sort

☑ Level 1 1R ☑ Level 2 2R ☑ Level 3 3R

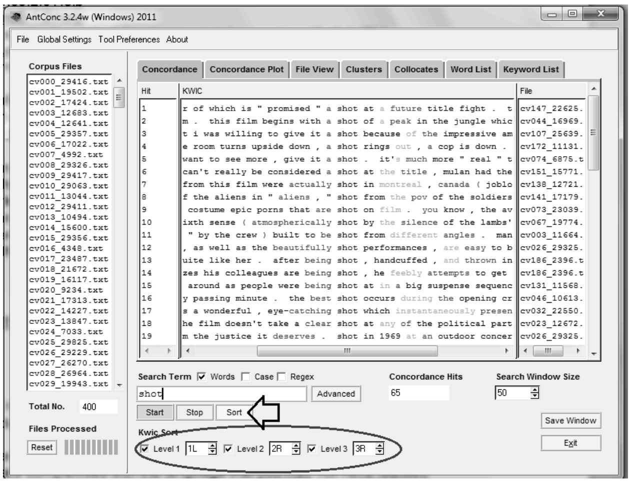
(14618 times, according to the Concordance Hits box in the bottom centre.)

As above, the KWIC list is a good way to start looking for patterns. Even though it's still a lot of information, what kinds of words appear near "the"?

Try a similar search for "a". Both "a" and "the" are articles, but one is a definite article and one an indefinite article - and the results you get will be illustrative of that.

Now that you're comfortable with looking at a KWIC line, try doing it again with "shot": this will produce examples of both shot the noun ('line up the **shot**') and the verb 'this scene was **shot** carefully')

What do you see? I understand this can be a difficult to read way of identifiying patterns. Try pressing the yellow "sort" button. What happens now?



Corpus Files

cv000_29416.txt
cv001_19502.txt
cv002_17424.txt
cv003_12683.txt
cv004_12641.txt
cv005_29357.txt
cv006_17022.txt
cv007_4992.txt
cv008_29326.txt
cv009_29417.txt
cv010_29063.txt
cv011_13044.txt
cv012_29411.txt
cv013_10494.txt
cv014_15600.txt
cv015_29356.txt
cv016_4348.txt
cv017_23487.txt
cv018_21672.txt
cv019_16117.txt
cv020_9234.txt
cv021_17313.txt
cv022_14227.txt
cv023_13847.txt
cv024_7033.txt
cv025_29825.txt
cv026_29229.txt
cv027_26270.txt
cv028_26964.txt
cv029_19943.txt

Total No. 400

Files Processed

Hit	KWIC	File
1	r of which is " promised " a shot at a future title fight . t	cv147_22625.
2	m . this film begins with a shot of a peak in the jungle whic	cv044_16969.
3	t i was willing to give it a shot because of the impressive am	cv107_25639.
4	e room turns upside down , a shot rings out , a cop is down .	cv172_11131.
5	want to see more , give it a shot . it's much more " real " t	cv074_6875.t
6	can't really be considered a shot at the title , mulan had the	cv151_15771.
7	from this film were actually shot in montreal , canada (joblo	cv138_12721.
8	f the aliens in " aliens , " shot from the pov of the soldiers	cv141_17179.
9	costume epic porns that are shot on film . you know , the av	cv073_23039.
10	ixth sense (atmospherically shot by the silence of the lambs'	cv067_19774.
11	" by the crew) built to be shot from different angles . man	cv003_11664.
12	, as well as the beautifully shot performances , are easy to b	cv026_29325.
13	uite like her . after being shot , handcuffed , and thrown in	cv186_2396.t
14	zes his colleagues are being shot , he feebly attempts to get	cv186_2396.t
15	around as people were being shot at in a big suspense sequenc	cv131_11568.
16	y passing minute . the best shot occurs during the opening cr	cv046_10613.
17	s a wonderful , eye-catching shot which instantaneously presen	cv032_22550.
18	he film doesn't take a clear shot at any of the political part	cv023_12672.
19	m the justice it deserves . shot in 1969 at an outdoor concer	cv026_29325.

Search Term ☑ Words ☐ Case ☐ Regex

shot

Concordance Hits 65

Search Window Size 50

Kwic Sort

☑ Level 1 1L ☑ Level 2 2R ☑ Level 3 3R

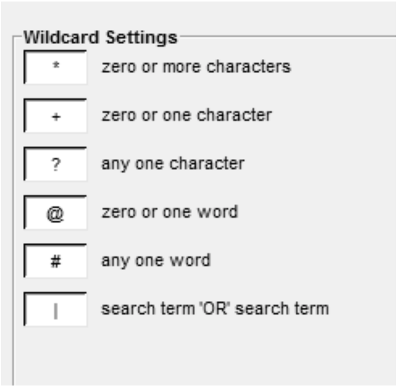
(This might be easier to read!) You can adjust the way AntConc sorts information by changing the parameters in the red circle: L corresponds with 'left' and R corresponds with 'right'; you can extend these up to ±5 in either direction. The default is 1 left, 2 right, 3 right, but you can change that to search 3 left, 2 left, 1 right (to get phrases and/or trigrams that end in the search term in question, for example) by clicking the arrow buttons up or down. If you don't want to include a sorting option you can skip it (as in the default: 1L, 2R, 3R) or include it as a 0. Less linear sorting practices are available, such as 4 left, 3 right, 5 right, which includes a lot of other contextual information. These parameters can be slow to respond, but be patient. If you're not sure what the resulting search is, just press 'sort' to see what's happened and adjust accordingly. ### Search Operators

The * operator (wildcard)

The * operator (which finds zero or more characters) can help, for instance, find both the singular and the plural forms of nouns.

Task: Search for qualit*, then sort this search. What tends to precede and follow quality & qualities? (Hint: they're different words, and have different contexts. Again- look for patterns in usage using the KWIC!)

For a full list of available wildcard operators and what they mean, go to Global Settings > Wildcard Settings.



Wildcard Settings

*	zero or more characters
+	zero or one character
?	any one character
@	zero or one word
#	any one word
|	search term 'OR' search term

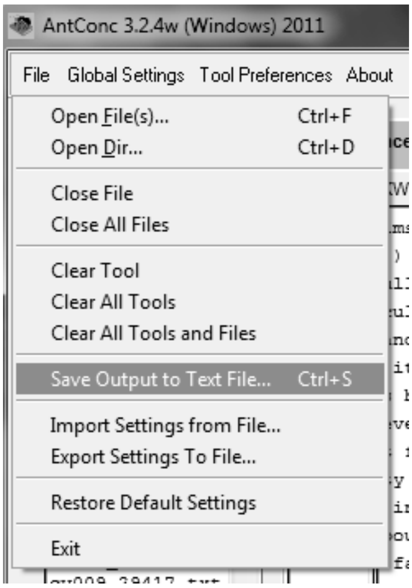
To find out the difference between * and ?, search for th*n and th?n. These two search queries look very similiar, but show very different results.

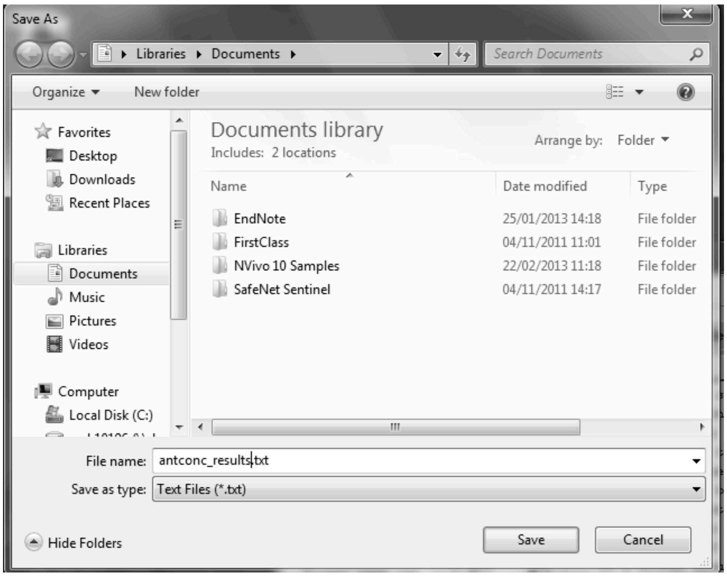
The ? operator is more specific than the * operator:
wom?n – both women and woman
m?n – man and men, but also min
contrast to m*n: not helpful, because you'll get mean, melon, etc.

Task: Compare these two searches: wom?n and m?n

1. sort each search in a meaningful way (eg. by search term then 1L then 2L)
2. File > Save output to text file (& append with .txt.

HINT: During the course of exploring in your research, you may generate many such files for reference; it's helpful to use descriptive filenames that describe what's in them (such as "wom?n-results.text", not "antconc_results.txt").



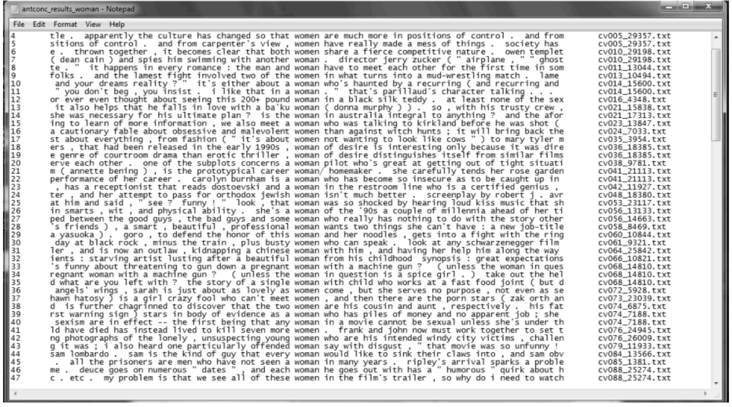


Name	Date modified	Type
EndNote	25/01/2013 14:18	File folder
FirstClass	04/11/2011 11:01	File folder
NVivo 10 Samples	22/02/2013 11:18	File folder
SafeNet Sentinel	04/11/2011 14:17	File folder

File name: antconc_results.txt

Save as type: Text Files (*.txt)

And now you can open the plain text file in your text editor; you might have to widen the application window to make it readable:



```
4      tle .  apparently the culture has changed so that women are much more in positions of control .  and from       cv005_29357.txt
5      sitions of control .  and from carpenter's view , women have really made a mess of things .  society has        cv005_29357.txt
6      e .  thrown together , it becomes clear that both women share a fierce competitive nature .  owen templet       cv010_29198.txt
7      ( dean cain ) and spies him swimming with another woman .  director jerry zucker ( " airplane , " " ghost       cv010_29198.txt
8      te . "  it happens in every romance : the man and woman have to meet each other for the first time in som       cv011_13044.txt
9      folks .  and the lamest fight involved two of the women in what turns into a mud-wrestling match .  lame        cv013_10494.txt
10      and your dreams reality ? "  it's either about a woman who's haunted by a recurring ( and recurring and        cv014_15600.txt
11      " you don't beg , you insist .  i like that in a woman . "  that's parillaud's character talking . . .         cv014_15600.txt
12     or ever even thought about seeing this 200+ pound woman in a black silk teddy .  at least none of the sex       cv016_4348.txt
13      it also helps that he falls in love with a ba'ku woman ( donna murphy ) ) .  so , with his trusty crew ,       cv021_15838.txt
14     she was necessary for his ultimate plan ?  is the woman in australia integral to anything ?  and the afor       cv021_17313.txt
15     ing to learn of more information , we also meet a woman who was talking to kirkland before he was shot (        cv023_13847.txt
16     a cautionary fable about obsessive and malevolent women than against witch hunts ; it will bring back the       cv024_7033.txt
17     st about everything , from fashion ( " it's about women not wanting to look like cows " ) to mary tyler m       cv035_3954.txt
18     ers , that had been released in the early 1990s , woman of desire is interesting only because it was dire      cv036_18385.txt
19     e genre of courtroom drama than erotic thriller , woman of desire distinguishes itself from similar films       cv036_18385.txt
20     erve each other .  one of the subplots concerns a woman pilot who's great at getting out of tight situati       cv038_9781.txt
21     m ( annette bening ) , is the prototypical career woman/ homemaker .  she carefully tends her rose garden       cv041_21113.txt
22     performance of her career .  carolyn burnham is a woman who has become so insecure as to be caught up in        cv041_21113.txt
23      , has a receptionist that reads dostoevski and a woman in the restroom line who is a certified genius ,        cv042_11927.txt
24     ter , and her attempt to pass for orthodox jewish woman isn't much better .  screenplay by robert j . avr       cv048_18380.txt
25     at him and said , " see ?  funny ! "  look , that woman was so shocked by hearing loud kiss music that sh       cv053_23117.txt
26     in smarts , wit , and physical ability .  she's a woman of the '90s a couple of millennia ahead of her ti       cv056_13133.txt
27     ped between the good guys , the bad guys and some woman who really has nothing to do with the story other      cv056_14663.txt
28     's friends ) , a smart , beautiful , professional woman wants two things she can't have : a new job-title       cv058_8469.txt
29     a yasuoka ) .  goro , to defend the honor of this woman and her noodles , gets into a fight with the ring       cv060_10844.txt
30      day at black rock , minus the train , plus busty women who can speak .  look at any schwarzenegger film        cv061_9321.txt
31     ler , and is now an outlaw , kidnapping a chinese woman with him , and having her help him along the way        cv064_25842.txt
32     ients : starving artist lusting after a beautiful woman from his childhood  synopsis : great expectations      cv066_10821.txt
33     's funny about threatening to gun down a pregnant woman with a machine gun ?   ( unless the woman in ques      cv068_14810.txt
34     regnant woman with a machine gun ?   ( unless the woman in question is a spice girl . )  take out the hel      cv068_14810.txt
35     d what are you left with ?  the story of a single woman with child who works at a fast food joint ( but d      cv068_14810.txt
36      angels' wings , sarah is just about as lovely as women come , but she serves no purpose , not even as se       cv072_5928.txt
37     hawn hatosy ) is a girl crazy fool who can't meet women , and then there are the porn stars ( zak orth an      cv073_23039.txt
38     d  is further chagrinned to discover that the two women are his cousin and aunt , respectively .  his fat      cv074_6875.txt
39     rst warning sign ) stars in body of evidence as a woman who has piles of money and no apparent job ; she       cv074_7188.txt
40      sexism are in effect -- the first being that any woman in a movie cannot be sexual unless she's under th      cv074_7188.txt
41     ld have died has instead lived to kill seven more women .  frank and john now must work together to set t      cv076_24945.txt
42     ng photographs of the lonely , unsuspecting young women who are his intended windy city victims , challen      cv076_26009.txt
43     g it was ; i also heard one particularly offended woman say with disgust , " that movie was so unfunny !       cv079_11933.txt
44     sam lombardo .  sam is the kind of guy that every woman would like to sink their claws into , and sam obv      cv084_13566.txt
45      .  all the prisoners are men who have not seen a woman in many years .  ripley's arrival sparks a proble      cv085_1381.txt
46     me .  deuce goes on numerous " dates " , and each woman he goes out with has a " humorous " quirk about h      cv088_25274.txt
47     c . etc .  my problem is that we see all of these women in the film's trailer , so why do i need to watch      cv088_25274.txt
```

Do this for each of the two searches and then look at the two text files side by side. What do you notice?

The | operator ("or")

Task: Search on she|he.

Now search for these separately: how many instances of she vs he?

There are many fewer instances of she – why? That's a research question! A good follow-up questions might be to sort the she|he search for patterns, and look to see if particular verbs follow each.

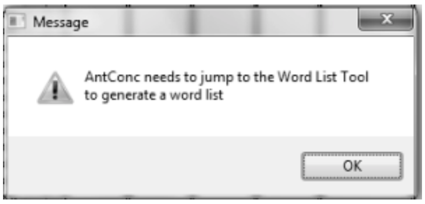
Task: Practice searching a word of your choice, sorting in different ways, using wildcard(s), and finally exporting. Guiding focus question here: what kinds of patterns do you see? Can you explain them?

Collocates and word lists

Having looked at the KWIC lines for patterns, don't you wish there was a way for the computer to give you a list of words which appear most frequently in company with your keyword?

Good news - there is a way to get this information, and it's available from the Collocates tab. Click that, and AntConc will tell you it needs to create a word list. Hit OK; it will do it automatically.

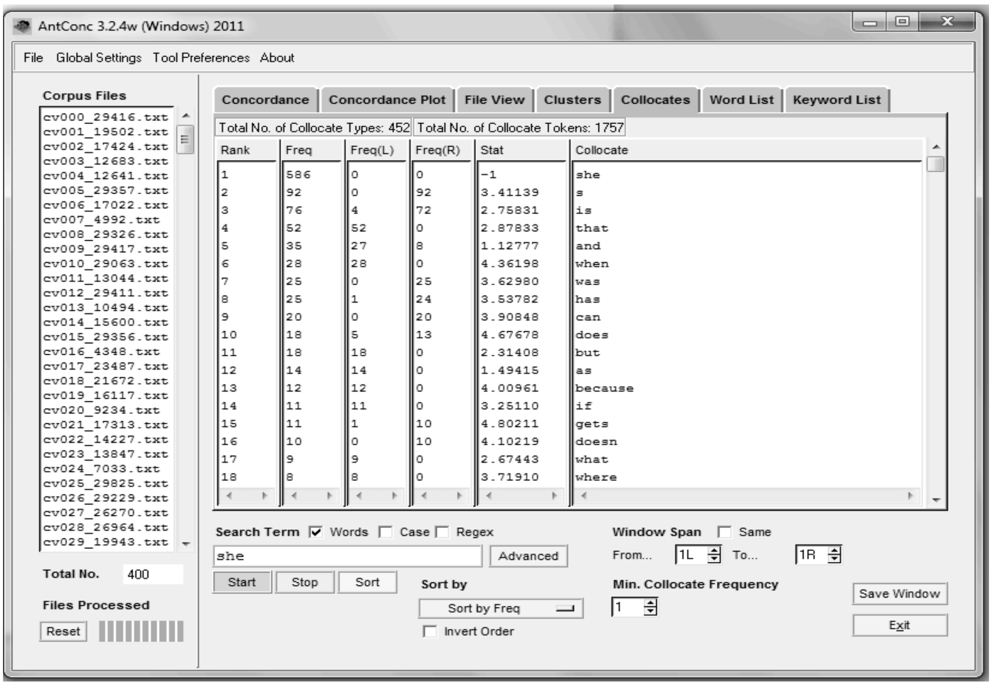
NOTE: You will only get this notice when you haven't created a word list yet.



Message

AntConc needs to jump to the Word List Tool to generate a word list

Try generating collocates for she.



Corpus Files

cv000_29416.txt
cv001_19502.txt
cv002_17424.txt
cv003_12683.txt
cv004_12641.txt
cv005_29357.txt
cv006_17022.txt
cv007_4992.txt
cv008_29326.txt
cv009_29417.txt
cv010_29063.txt
cv011_13044.txt
cv012_29411.txt
cv013_10494.txt
cv014_15600.txt
cv015_29356.txt
cv016_4348.txt
cv017_23487.txt
cv018_21672.txt
cv019_16117.txt
cv020_9234.txt
cv021_17313.txt
cv022_14227.txt
cv023_13847.txt
cv024_7033.txt
cv025_29825.txt
cv026_29229.txt
cv027_26270.txt
cv028_26964.txt
cv029_19943.txt

Total No. 400

Files Processed

Total No. of Collocate Types: 452 Total No. of Collocate Tokens: 1757

Rank	Freq	Freq(L)	Freq(R)	Stat	Collocate
1	586	0	0	-1	she
2	92	0	92	3.41139	s
3	76	4	72	2.75831	is
4	52	52	0	2.87833	that
5	35	27	8	1.12777	and
6	28	28	0	4.36198	when
7	25	0	25	3.62980	was
8	25	1	24	3.53782	has
9	20	0	20	3.90848	can
10	18	5	13	4.67678	does
11	18	18	0	2.31408	but
12	14	14	0	1.49415	as
13	12	12	0	4.00961	because
14	11	11	0	3.25110	if
15	11	1	10	4.80211	gets
16	10	0	10	4.10219	doesn
17	9	9	0	2.67443	what
18	8	8	0	3.71910	where

Search Term ☑ Words □ Case □ Regex

she

Sort by

Sort by Freq

□ Invert Order

Window Span □ Same

From... 1L To... 1R

Min. Collocate Frequency

1

she with collocates

The unsorted results will seem to start with function words (words that build phrases) then go down to content words (words that build meaning)– these small boring words are the most frequent words in English,[381] which are largely phrase builders. Later versions of AntConc often include the search term as the first hit, presumably because the search term you are looking for shows up in the text and we are looking for words which are likely to appear with this word.

Some people might want to remove these small words by using a stopword list; this is a common step in topic modelling. Personally I don't encourage this practice because addressing highly-frequent words is where computers shine! As readers we tend not to notice them very much. Computers, especially software like Antconc, can show us where these words do and do not appear and that can be quite interesting, especially in very large collections of text - as explored earlier in the tutorial, with *the*, *a*, *she* and *he*.

Additionally you may have a single letter 's' appear, quite high as well - that represents the possessive *'s* (the apostrophe won't be counted), but AntConc considers that s indicative of another word. Another example of this is *'t* appearing with *do*, as they contract as *don't*. Because these so commonly appear together, this makes them highly likely collocates.

Task: Generate collocates for m?n and wom?n. Now sort them by frequency to 1L.

[381] 'Word frequency data: Corpus of Contemporary American English': http://www.wordfrequency.info/free.asp

This tells us about what makes a man or woman 'movie-worthy':
– women have to be 'beautiful' or 'pregnant' or 'sophisticated'
– men have to be somehow outside the norm – 'holy' or 'black' or 'old'

This is not necessarily telling us about the movies but about the way those movies are written about in reviews, and can lead us to ask more nuanced questions, like "How are women in romantic comedies described in reviews written by men compared to those written by women?"

Comparing corpora

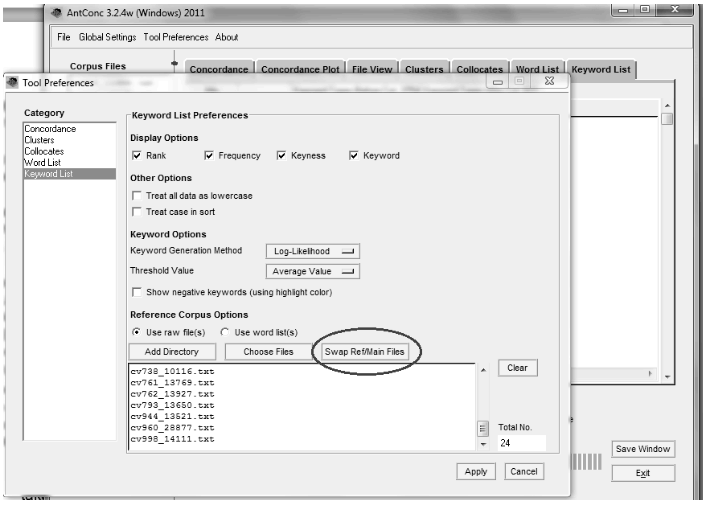
One of the most powerful types of analysis is comparing your corpus to a larger reference corpus.

I've pulled out reviews of movies with which Steven Spielberg is associated (as director or producer). We can compare them to a reference corpus of movies by a range of directors.

Be sure to think carefully about what a reference corpus for your own research might look like (eg. a study of Agatha Christie's language in her later years would work nicely as an analysis corpus for comparison to a reference corpus of all her novels). Remember, again, that corpus construction is a subfield in its own right.

Settings > Tool preferences > Keyword List
Under 'Reference Corpus' make sure "Use raw files" is checked
Add Directory > open the folder containing the files that make up the reference corpus

Ensure you have a whole list of files



Keyword List Preferences

Display Options

☑ Rank ☑ Frequency ☑ Keyness ☑ Keyword

Other Options

□ Treat all data as lowercase
□ Treat case in sort

Keyword Options

Keyword Generation Method: Log-Likelihood
Threshold Value: Average Value
□ Show negative keywords (using highlight color)

Reference Corpus Options

● Use raw file(s) ○ Use word list(s)

```
cv738_10116.txt
cv761_13769.txt
cv762_13927.txt
cv793_13650.txt
cv944_13521.txt
cv960_28877.txt
cv998_14111.txt
```

Total No. 24

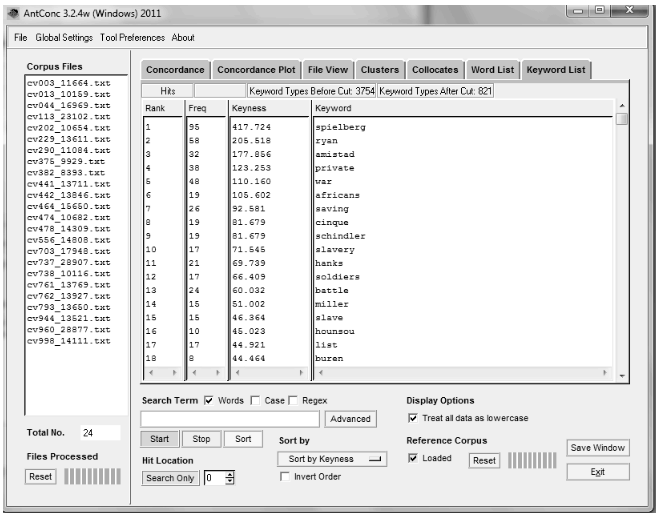
Hit Load (& wait …) then once the 'Loaded' box is checked, hit Apply.
You can also opt to swap reference corpus & main files (SWAP REF/MAIN FILES). It is worth looking at what both results show. > If you're using a later version of AntConc, the Swap Ref/Main files option may be marked as

'swap with target files', and you will need to ensure the target and reference corpora have been loaded (press the load button each time you upload, or swap, a corpus).

In Keyword List, just hit Start (with nothing typed in the search box). If you've just swapped the reference corpus and the target files, you may be prompted to create a new word list before AntConc will calculate the keywords. We see a list of Keywords that have words that are much more "unusual" – more statistically unexpected – in the corpus we are looking at when compared to the reference corpus.

> Keyness: this is the frequency of a word in the text when compared with its frequency in a reference corpus, "such that the statistical probability as computed by an appropriate procedure is smaller than or equal to a p value specified by the user."[382]

What are our keywords?



Corpus Files

cv003_11664.txt
cv013_10159.txt
cv044_16969.txt
cv113_23102.txt
cv202_10654.txt
cv229_13611.txt
cv290_11084.txt
cv375_9929.txt
cv382_8393.txt
cv441_13711.txt
cv442_13846.txt
cv464_15650.txt
cv474_10682.txt
cv478_14309.txt
cv556_14808.txt
cv703_17948.txt
cv737_28907.txt
cv738_10116.txt
cv761_13769.txt
cv762_13927.txt
cv793_13650.txt
cv944_13521.txt
cv960_28877.txt
cv998_14111.txt

Total No. 24

Files Processed

Keyword Types Before Cut: 3754 Keyword Types After Cut: 821

Rank	Freq	Keyness	Keyword
1	95	417.724	spielberg
2	58	205.518	ryan
3	32	177.856	amistad
4	38	123.253	private
5	48	110.160	war
6	19	105.602	africans
7	26	92.581	saving
8	19	81.679	cinque
9	19	81.679	schindler
10	17	71.545	slavery
11	21	69.739	hanks
12	17	66.409	soldiers
13	24	60.032	battle
14	15	51.002	miller
15	15	46.364	slave
16	10	45.023	hounsou
17	17	44.921	list
18	8	44.464	buren

Search Term ☑ Words ☐ Case ☐ Regex

Sort by: Sort by Keyness ☐ Invert Order

Hit Location: Search Only 0

Display Options: ☑ Treat all data as lowercase

Reference Corpus: ☑ Loaded

spielberg vs movie reviews

Discussion: Making meaningful comparisons

Keep in mind that the way your organize your text files makes a difference to the kinds of questions you can ask and the kinds of results you will get. Remember that we are comparing 'negative' and 'positive' reviews quite flatly here. You could, for instance, make other comparisons with different subsets of reviews, which yield very different kinds of questions.

Of course, the files you put in your corpus will shape your results. Again, the question of representativeness and sampling are highly relevant here – it's not always necessary or even ideal to use *all* of a dataset at once, even if you do have it. At this juncture, it's really worth interrogating how these methods help produce research questions.

[382] 'Definition of Key-ness': http://www.lexically.net/downloads/version6/HTML/index.html?keyness_definition.htm

When thinking about how movie reviews work as a genre, you could consider, for example...

Movie reviews vs music reviews
Movie reviews vs book reviews
Movie reviews vs news articles about sport
Movie reviews vs news articles in general

Each of these comparisons will tell you something different, and can produce different research questions, such as:

How are movie reviews different than other kinds of media reviews?
How are movie reviews different than other kinds of published writing?
How do movie reviews compare to other specific kinds of writing, such as sport writing?
How do movie reviews have in common with music reviews?

And of course you could flip those questions to make further research questions:

How are book reviews different to movie reviews?
How are music reviews different than movie reviews?
What do published newspaper articles have in common?
How are movie reviews similar to other kinds of published writing?

In summary: it's worth thinking about:

Why you might want to compare two corpora
What kinds of queries make meaningful research questions
Principles of corpora construction: sampling & ensuring you can get something representative

Further resources for this tutorial

A short bibliography on corpus linguistics.[383]

A more step-by-step version of this tutorial, assuming no computer knowledge[384]

About the Author

Heather Froehlich is a PhD student at the University of Strathclyde (Glasgow, UK), where she studies gender in Early Modern London plays using computers. Her thesis draws heavily from sociohistoric linguistics and corpus stylistics, though she sustains an interest in digital methods for literary and linguistic inquiry.

[383] Heather Froehlich, 'An Introductory Bibliography to Corpus Linguistics': http://hfroehli.ch/2014/05/11/intro-bibliography-corpus-linguistics/

[384] Heather Froehlich, 'Getting Started with Antconc': http://hfroehli.ch/workshops/getting-started-with-antconc/

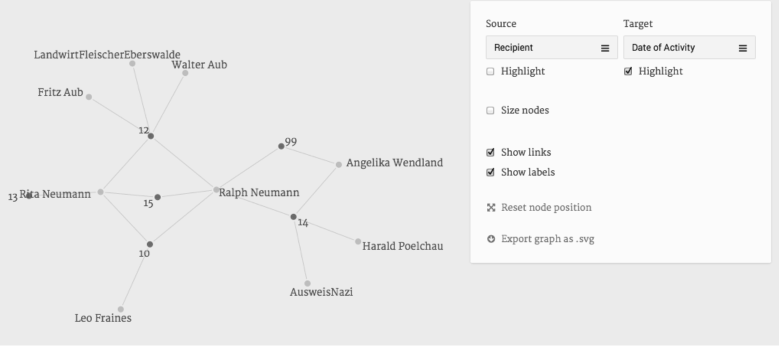
34. From Hermeneutics to Data to Networks: Data Extraction and Network Visualization of Historical Sources

Marten Düring – 2015

Table of contents:

Introduction

Network visualizations can help humanities scholars reveal hidden and complex patterns and structures in textual sources. This tutorial explains how to extract network data (people, institutions, places, etc) from historical sources through the use of non-technical methods developed in Qualitative Data Analysis (QDA) and Social Network Analysis (SNA), and how to visualize this data with the platform-independent and particularly easy-to-use *Palladio.*[385]



Source: Recipient
Target: Date of Activity
□ Highlight ☑ Highlight
□ Size nodes
☑ Show links
☑ Show labels

A network visualization in Palladio and what you will be able to create by the end of this tutorial

The graph above shows an excerpt from the network of Ralph Neumann, particularly his connections to people who helped him and his sister during

[385] 'Palladio': http://palladio.designhumanities.org/#/

their life in the underground in Berlin 1943-1945. You could easily modify the graph and ask: Who helped in which way? Who helped when? Who is connected to whom?

Generally, network analysis provides the tools to explore highly complex constellations of relations between entities. Think of your friends: You will find it very easy to map out who are close and who don't get along well. Now imagine you had to explain these various relationships to somebody who does not know any of your friends. Or you wanted to include the relationships between your friends' friends. In situations like this language and our capacity to comprehend social structures quickly reach their limits. Graph visualizations can be means to effectively communicate and explore such complex constellations. Generally you can think of Social Network Analysis as a means to transform complexity from a problem to an object of research. Often, nodes in a network represent humans connected to other humans by all imaginable types of social relations. But pretty much anything can be understood as a node: A film, a place, a job title, a point in time, a venue. Similarly, the concept of a tie (also called edge) between nodes is just as flexible: two theaters could be connected by a film shown in both of them, or by co-ownership, geographical proximity, or being in business in the same year. All this depends on your research interests and how you express them in form of nodes and relations in a network.

This tutorial can not replace any of the many existing generic network analysis handbooks, such as John Scott's *Social Network Analysis.*[386] For a great general introduction to the field and all its pitfalls for humanists I recommend*Scott Weingart's blog post series "Networks Demystified"*[387] as well as*Claire Lemercier's paper "Formal network methods in history: why and how?"*.[388] You may also want to explore the bibliography and event calendar over at *Historical Network Research*[389] to get a sense of how historians have made use of networks in their research.

This tutorial will focus on data extraction from unstructured text and shows one way to visualize it using Palladio. It is purposefully designed to be as simple and robust as possible. For the limited scope of this tutorial it will suffice to say that an actor refers to the persons, institutions, etc. which are the object of study and which are connected by relations. Within the context of a network visualization or computation (also called graph), we call them nodes and we call the connections ties. In all cases it is important to remember that nodes and ties are drastically simplified models used to represent the complexities of past events, and in themselves do not always suffice to generate insight. But it is likely that the graph will highlight interesting aspects, challenge your hypothesis and/or lead you to

[386] John Scott, *Social Network Analysis*, 2013.

[387] Scott Weingart, 'Demystifying Networks' (14 December 2011): http://www.scottbot.net/HIAL/?p=6279

[388] Claire Lemercier, 'Formal network method in history: why and how?' (2011): https://hal.archives-ouvertes.fr/file/index/docid/649316/filename/lemercier_A_zg.pdf

[389] 'Historical Network Research': http://historicalnetworkresearch.org/

generate new ones. *Network diagrams become meaningful when they are part of a dialogue with data and other sources of information.*

Many network analysis projects in the social sciences rely on pre-existing data sources or data that was created for the purpose of network analysis. Examples include email logs, questionnaires or trade relations which make it relatively easy to identify who is connected to whom and how. It is considerably more difficult to extract network data from unstructured text. This forces us to somehow marry the complexities of hermeneutics with the rigor of formal data analysis. The term "friend" might serve as an example: Depending on the context it can signify anything from an insult to an expression of love. Context knowledge and analysis of the text will help you identify what it stands for in any given case. A formal category system should represent the different meanings inasmuch detail as necessary for your purposes.

In other words, the challenge is to systematize text interpretation. Networks created from pre-existing data sets need to be considered within the context in which they were created (e.g. wording of questions in a questionnaire and selected target groups). Networks created from unstructured text pose challenges on top of this: interpretations are highly individual and depend on viewpoints and context knowledge.

About the case study

The case study I use for this tutorial is a first-person narrative of Ralph Neumann, a Jewish survivor of the Holocaust. You can find the text *online*.[390] The coding scheme which I will introduce below is a simplified version of the one I developed during my PhD project on covert support networks during the Second World War. My research was driven by three questions: To what extent can social relationships help explain why ordinary people took the risks associated with helping? How did such relationships enable people to provide these acts of help given that only very limited resources were available to them? How did social relationships help Jewish refugees to survive in the underground?

In this project network visualisations helped me to discover hitherto forgotten yet highly important contact brokers, highlight the overall significance of Jewish refugees as contact brokers and generally to navigate through a total of some 5,000 acts of help which connected some 1,400 people between 1942 and 1945.

Developing a coding scheme

In visualizing network relationships, one of the first and most difficult challenges is to decide who should be part of the network and which relations between the selected actors are to be coded. It will probably take

[390] Ralph Neuman, 'Memories from My Early Life in Germany 1926-1946': http://www.gdw-berlin.de/fileadmin/bilder/publ/publikationen_in_englischer_sprache/2006_Neuman_eng.pdf

some time to figure this out and will likely be an iterative process since you will need to balance your research interests and hypotheses with the availability of information in your texts and represent both in a rigid and necessarily simplifying coding scheme.

The main questions during this process are: Which aspects of relationships between two actors are relevant? Who is part of the network? Who is not? Which attributes matter? What do you aim to find?

I found the following answers to these:

What defines a relationship between two actors?

Any action which directly contributed to the survival of persecuted persons in hiding. This included e.g. non-Jewish communists but excluded bystanders who chose not to denunciate refugees or mere acquaintances between actors (for lack of sufficient coverage in the sources). Actors were coded as either providers or recipients of an act of help independently of their status as refugees. There is no simple and robust way to handle ambiguities and doubt at the moment. I therefore chose to collect verifiable data only.

Who is part of the network? Who is not?

Anyone who is mentioned as a helper, involved in helping activities, involved in activities which aimed to suppress helping behaviour. In fact, some helping activities turned out to be unconnected to my case studies but in other cases this approach revealed hitherto unexpected cross-connections between networks.

Which types of relationships do you observe?

Rough categorizations of: Form of help, intensity of relationships, duration of help, time of help, time of first meeting (both coded in 6-months steps).

Which attributes are relevant?

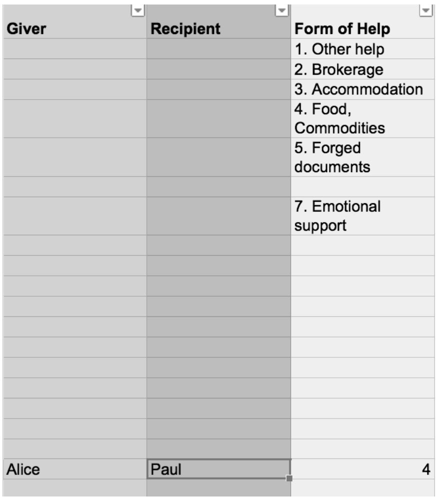
Mainly racial status according to National Socialist legislation.

What do you aim to find?

A deeper understanding of who helps whom how, and discovery of patterns in the data that correspond to network theory. A highly productive interaction between my sources and the visualized data made me stick with this.

Note that coding schemes in general are not able to represent the full complexity of sources in all their subtleties and ambivalence. The purpose of the coding scheme is to develop a model of the relationships you are interested in. As such, the types of relations and the attributes are abstracted and categorized renditions of the complexities conveyed in the text(s). This also means that in many cases network data and visualizations will only make sense once reunited with their original context, in my case the primary sources from which I extracted it.

The translation of text interpretation into data collection has its roots in sociological Qualitative Data Analysis. It is important that you and others can retrace your steps and understand how you define your relations. It is very helpful to define them abstractly and to provide examples from your sources to further illustrate your choices. Any data you produce can only be as clear and coherent as your coding practices. Clarity and coherence increase during the iterative process of creating coding schemes and by testing it on a variety of different sources until it fits.



Giver	Recipient	Form of Help
		1. Other help
		2. Brokerage
		3. Accommodation
		4. Food, Commodities
		5. Forged documents
		7. Emotional support
Alice	Paul	4

A first stab at the coding scheme

The figure above shows a snapshot with sample data of the coding scheme I used during my project. In this case Alice helps Paul. We can express this as a relation between the actors “Alice” and “Paul” which share a relation of the category “Form of Help”. Within this category we find the subcategory “4. Food, Commodities” which further describes their relation.

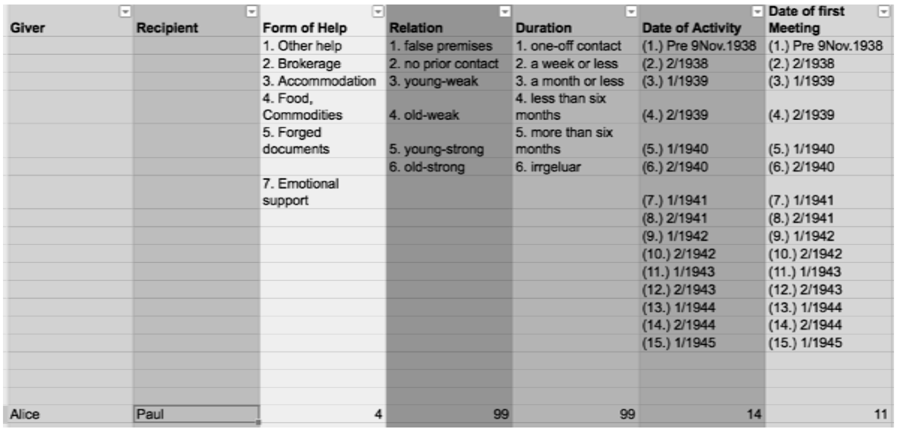
All major network visualization tools let you specify whether a network is directed like this one or undirected. In directed networks, relations describe an exchange from one actor to another, in our case this is “help”. By convention, the active nodes are mentioned first (in this case Alice) in the dataset. In a visualization of a directed network, you will see arrows going from one actor to another. Relations can also be reciprocal, for example when Alice helps Bob and Bob helps Alice.

Quite often, however, it doesn’t make sense to work with directionality, for example when two actors are simply part of the same organization. In this case the network should be undirected and would be represented by a simple line between the two actors.

I wanted to know how often actors gave help and how often they received it. I was particularly interested in the degree of Jewish self-help, which is why a directed network approach and the role of "Giver" and "Recipient" make sense. The third column in the coding scheme is optional and further describes the kind of relationship between Alice and Paul. As a category I chose "Form of Help" which reflects the most common ways in which support was given.

The categories and subcategories emerged during a long process of coding different types of texts and different types of support networks. During this process I learned, for example, which relevant forms of help are rarely described and therefore not traceable, such as the provision of support-related information. Expect having to adapt your coding scheme frequently in the beginning and brace yourself for re-coding your data a few times until it consistenly corresponds with your sources and interests.

As it stands, the coding scheme conveys the information that Alice provided food or other commodities for Paul, as indicated by the value 4 which corresponds to the subcategory "4. Food, Commodities" in the category "Form of Help". Human relationships are however significantly more complex than this and characterized by different and ever-changing layers of relations. To an extent, we can represent some of this complexity by collecting *multiplex* relationships. Consider this sample sentence: *"In September 1944 Paul stayed at his friend Alice's place; they had met around Easter the year before."*

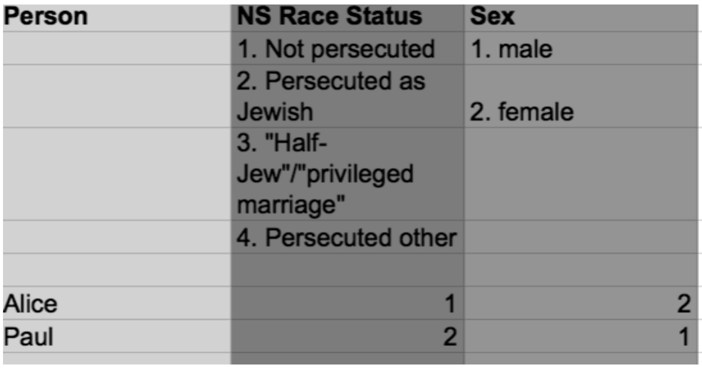


Giver	Recipient	Form of Help	Relation	Duration	Date of Activity	Date of first Meeting
		1. Other help	1. false premises	1. one-off contact	(1.) Pre 9Nov.1938	(1.) Pre 9Nov.1938
		2. Brokerage	2. no prior contact	2. a week or less	(2.) 2/1938	(2.) 2/1938
		3. Accommodation	3. young-weak	3. a month or less	(3.) 1/1939	(3.) 1/1939
		4. Food, Commodities	4. old-weak	4. less than six months	(4.) 2/1939	(4.) 2/1939
		5. Forged documents	5. young-strong	5. more than six months	(5.) 1/1940	(5.) 1/1940
			6. old-strong	6. irrgeluar	(6.) 2/1940	(6.) 2/1940
		7. Emotional support			(7.) 1/1941	(7.) 1/1941
					(8.) 2/1941	(8.) 2/1941
					(9.) 1/1942	(9.) 1/1942
					(10.) 2/1942	(10.) 2/1942
					(11.) 1/1943	(11.) 1/1943
					(12.) 2/1943	(12.) 2/1943
					(13.) 1/1944	(13.) 1/1944
					(14.) 2/1944	(14.) 2/1944
					(15.) 1/1945	(15.) 1/1945
Alice	Paul	4	99	99	14	11

A representation of the sample sentence.

The coding scheme in the figure above describes the relationships between helpers and recipients of help in greater detail. "Relation" for example gives a rough categorization of how well two actors knew each other, "Duration" captures how long an act of help lasted, "Date of Activity" indicates when an act of help occurred and "Date of first Meeting" should be self explanatory. The value "99" here specifies "unknown" since the sample sentence does not describe the intensity of the relationship between Alice and Paul in greater detail. Note that this scheme focuses exclusively on collecting acts of help, not on capturing the development of relationships between people (which were not covered in my sources). Explicit choices like this define the value of the data during analysis.

It is also possible to collect information on the actors in the network; so-called attribute data uses pretty much the same format. The figure below shows sample data for Alice and Paul.



Person	NS Race Status	Sex
	1. Not persecuted	1. male
	2. Persecuted as Jewish	2. female
	3. "Half-Jew"/"privileged marriage"	
	4. Persecuted other	
Alice	1	2
Paul	2	1

Sample attribute data

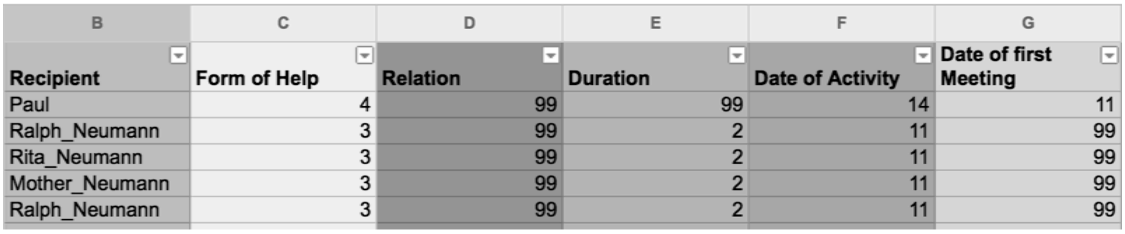
If we read the information now stored in the coding scheme we learn that Alice provided accommodation for Paul ("Form of Help": 4), that we do not know how close they were ("Relation": 99) or how long he stayed ("Duration": 99). We do know however that this took place some time in the second half of 1944 ("Date of Activity": 14) and that they had met for the first time in the first half of 1943 ("Date of first Meeting": 11). The date of first meeting can be inferred from the words *"around Easter the year before".* If in doubt, I always chose to enter "99" representing "unknown".

But what if Alice had also helped Paul with emotional support (another subcategory of "Form of Help") while he was staying with her? To acknowledge this, I coded one row which describes the provision of accommodation and a second below which describes the provision of emotional support. Note that not all network visualization tools will allow you to represent parallel edges and will either ignore the second act of help which occurred or try to merge the two relations. Both NodeXL and Palladio can handle this however and it is rumoured that a future release of Gephi will as well. If you encounter this problem and if none of the two tools are an option for you, I would recommend to set up a relational database and work with specific queries for each visualization.

The process of designing such a coding scheme forces you to become explicit about your assumptions, interests and the materials at your disposal, something valuable beyond data analysis. Another side effect of extracting network data from text is that you will get to know your sources very well: Sentences following the model of "Person A is connected to Persons B, C and D through relation type X at time Y" will probably be rare. Instead it will take close reading, deep context knowledge and interpretation to find out who is connected to whom in which way. This means that coding data in this way, will raise many questions and will force you to study your sources more deeply and more rigorously than if you had worked through them the “traditional” way.

Visualize network data in Palladio

Once you have come up with a coding scheme and encoded your sources you are ready to visualize the network relationships. First make sure that all empty cells are filled with either a number representing a type of tie or with "99" for “unknown”. Create a new copy of your file (Save as..) and delete the codes for the different categories so that your sheet looks something like the image below.



B	C	D	E	F	G
Recipient	Form of Help	Relation	Duration	Date of Activity	Date of first Meeting
Paul	4	99	99	14	11
Ralph_Neumann	3	99	2	11	99
Rita_Neumann	3	99	2	11	99
Mother_Neumann	3	99	2	11	99
Ralph_Neumann	3	99	2	11	99

Sample attribute data ready to be exported for visualization or computation

All spreadsheet editors let you export tables as either .csv (comma-separated values) or as .txt files. These files can be imported into all of the commonly used network visualization tools (see the list at the end of the tutorial). For your first steps however I suggest that you try out Palladio, a very easy-to-use data visualization tool in active development by Stanford University. It runs in browsers and is therefore platform-independent. Please note that Palladio, although quite versatile, is designed more for quick visualizations than sophisticated network analysis.

The following steps will explain how to visualize network data in Palladio but I also recommend that you take a look at their own training materials and explore their sample data. Here however I use a slightly modified sample dataset based on the coding scheme presented earlier (you can also download it and use it to explore other tools).[391]

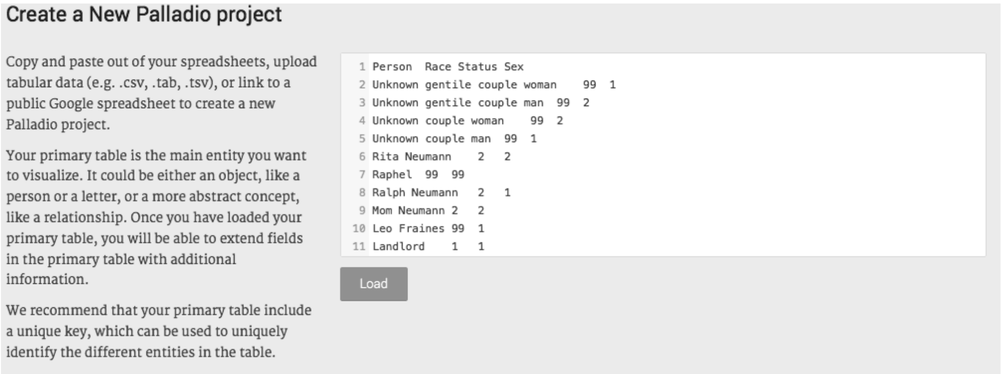
Step by Step:

1. **Palladio.** Go to http://palladio.designhumanities.org/.

2. **Start.** On their website click the “Start” button.

3. **Load attribute data.** From your data sheet, copy the attribute data (*Sample dataset*, Sheet 2) and paste it in the white section of the page, now click “Load”.

[391] Referred to hereafter as ‘Sample dataset’. Available from: https://docs.google.com/spreadsheets/d/1LzbWsG73m74t3p6xE7lutfVWuOdzOIfN55FbhCCRZvk/edit#gid=0. Link also available in the online version of the tutorial.



Create a New Palladio project

Copy and paste out of your spreadsheets, upload tabular data (e.g. .csv, .tab, .tsv), or link to a public Google spreadsheet to create a new Palladio project.

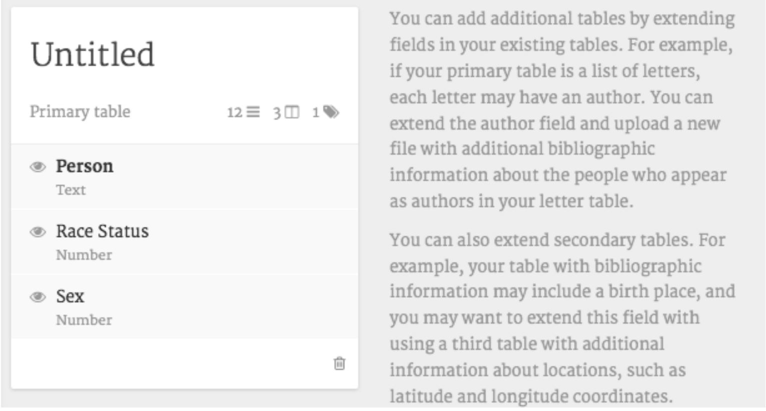
Your primary table is the main entity you want to visualize. It could be either an object, like a person or a letter, or a more abstract concept, like a relationship. Once you have loaded your primary table, you will be able to extend fields in the primary table with additional information.

We recommend that your primary table include a unique key, which can be used to uniquely identify the different entities in the table.

```
Person  Race Status Sex
Unknown gentile couple woman    99  1
Unknown gentile couple man  99  2
Unknown couple woman    99  2
Unknown couple man  99  1
Rita Neumann    2   2
Raphel  99  99
Ralph Neumann   2   1
Mom Neumann 2   2
Leo Fraines 99  1
Landlord    1   1
```

Loading attribute data into Palladio

4. Edit attributes. Change the title of the table to something more meaningful, such as “People”. Now you see the columns “Person”, “Race Status” and “Sex” which correspond to the columns in the sample data. Next you need to make sure that Palladio understands that there are actions associated with the people you just entered in the database.



Untitled

Primary table 12 ≡ 3 ◫ 1 🏷

Person
Text

Race Status
Number

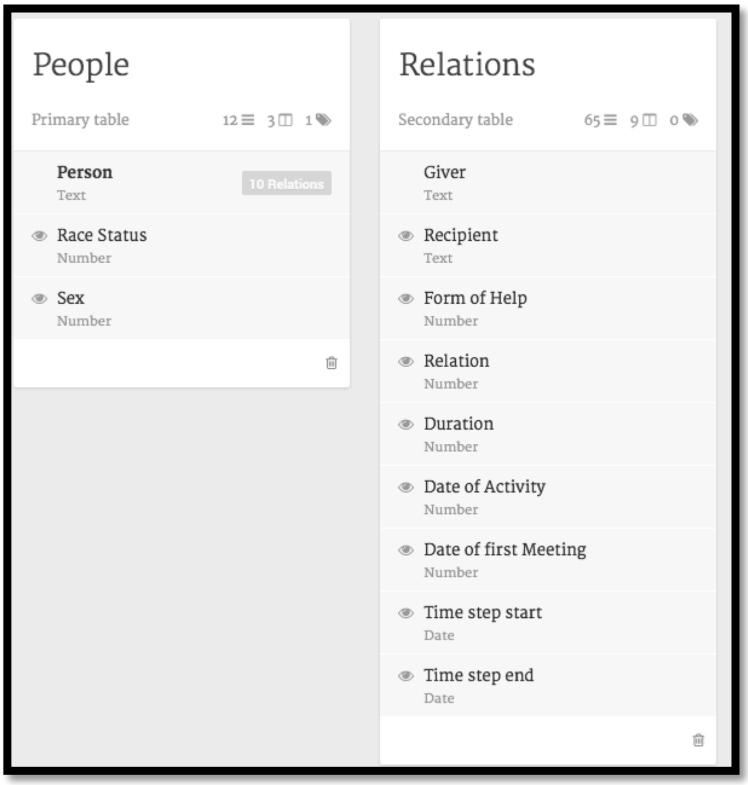
Sex
Number

You can add additional tables by extending fields in your existing tables. For example, if your primary table is a list of letters, each letter may have an author. You can extend the author field and upload a new file with additional bibliographic information about the people who appear as authors in your letter table.

You can also extend secondary tables. For example, your table with bibliographic information may include a birth place, and you may want to extend this field with using a third table with additional information about locations, such as latitude and longitude coordinates.

View of attribute data in Palladio

5. Load relational data. To do this, click on “Person” and “Add a new table”. Now paste all the relational data (*Sample data*, Sheet 1) in the appropriate field. Palladio expects unique identifiers to link the relational information to the actor attribute information. Make sure this lines up well and that you avoid any irritating characters such as “/”. Palladio will prompt you with error messages if you do. Click “Load data”, close the overlay window and go back to the main data overview. You should see something like this:



People

Primary table 12 ≡ 3 ◫ 1

Person
Text
10 Relations

Race Status
Number

Sex
Number

Relations

Secondary table 65 ≡ 9 ◫ 0

Giver
Text

Recipient
Text

Form of Help
Number

Relation
Number

Duration
Number

Date of Activity
Number

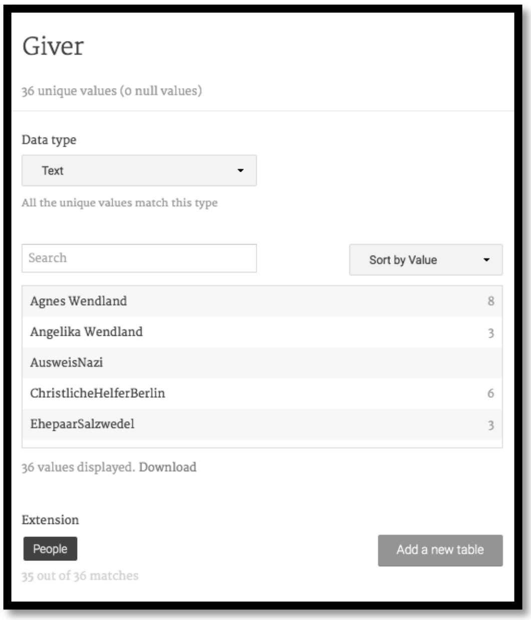
Date of first Meeting
Number

Time step start
Date

Time step end
Date

Loading relational data

6. Link attributes and relations. Next, we need to explicitly link the two tables we created. In our case, peoples' first- and last names work as IDs so we need to connect them. To do this click on the corresponding occurrences in the new table. In the sample files these are "Giver" and "Recipient". Click on "Extension" (at the bottom) and select "People", the table which contains all the people attribute information. Do the same for "Recipient".



Giver

36 unique values (0 null values)

Data type

Text

All the unique values match this type

Search

Sort by Value

Agnes Wendland	8
Angelika Wendland	3
AusweisNazi	
ChristlicheHelferBerlin	6
EhepaarSalzwedel	3

36 values displayed. Download

Extension

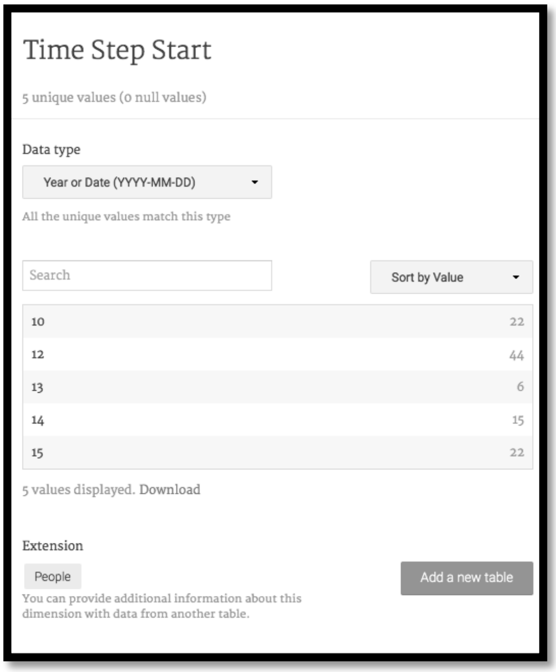
People

Add a new table

35 out of 36 matches

Linking People to Relations

7. Identify temporal data. Palladio has nice time visualization features. You can use it if you have start and end points for each relation. The sample data contains two columns with suitable data. Click on “Time Step Start” and select the data type “Year or Date”. Do the same for “Time Step End”. The Palladio team recommends that your data is in the YYYY-MM-DD format, but my more abstract time steps worked well. If you were to load geographical coordinates (not covered by this tutorial but here: *Palladio Simple Map Scenario*)[392] you would select the “Coordinates” data type.



Time Step Start

5 unique values (0 null values)

Data type

Year or Date (YYYY-MM-DD)

All the unique values match this type

Search

Sort by Value

10	22
12	44
13	6
14	15
15	22

5 values displayed. Download

Extension

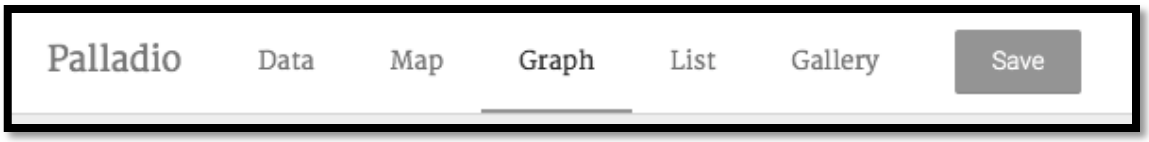
People

You can provide additional information about this dimension with data from another table.

Add a new table

Changing the data type to 'Year or Date'

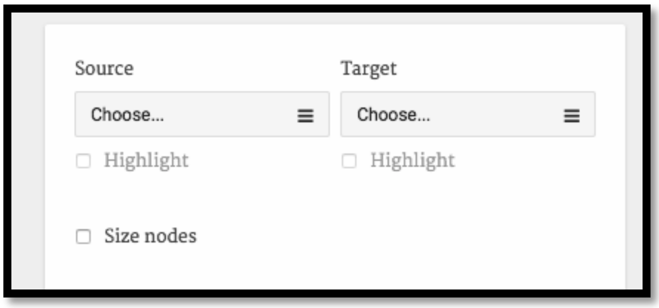
8. Open the Graph tool. You are now done with loading the data. Click “Graph” to load the visualization interface.



Load the Graph tool

9. Specify source and target nodes. First off Palladio asks you to specify the “Source” and “Target” nodes in the network. Let’s start with “Givers” and “Recipients”. You will now see the graph and can begin to study it in greater detail.

[392] ‘Scenario #1: Simple Map’: http://hdlab.stanford.edu/doc/scenario-simple-map.pdf

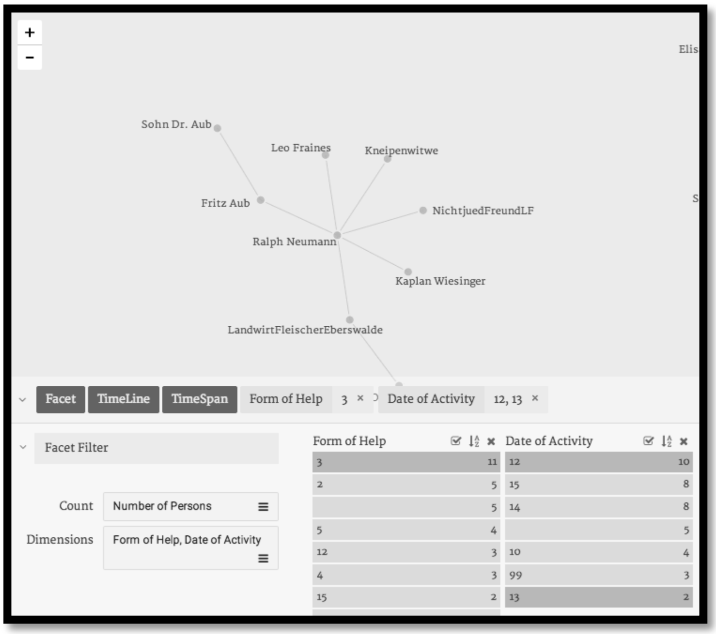


Select 'Giver' as source and 'Recipient' as target.

10. Highlight nodes. Continue by ticking the "Highlight" boxes. This will give you an immediate sense of who acted as a provider of help, who merely received help and which actors were both givers and recipients of help.

11. Facet filter. Next up, try the faceted filter (Figure 13). You will recognize the columns which describe the different acts of help. Start by selecting "3" in the "Form of Help" column. This will reduce the graph to only provisions of accommodation. Next, select values from the "Date of Activity" column to further narrow down your query. This will show you who provided accommodation and how this changes over time. Re-select all values in a column by clicking on the check box next to the column name. Take your time to explore the dataset – how does it change over time? When you are done, make sure to delete the Facet filter using the small red trashcan.

Network visualizations can be incredibly suggestive. Remember that whatever you see is a different representation of your data coding (and the choices you made along the way) and that there will be errors you might have to fix. Either of the graphs I worked with would have looked differently had I chosen different time steps or included people who merely knew each other but did not engage in helping behavior.



Form of Help 3 × Date of Activity 12, 13 ×

Count: Number of Persons

Dimensions: Form of Help, Date of Activity

Form of Help	
3	11
2	5
	5
5	4
12	3
4	3
15	2

Date of Activity	
12	10
15	8
14	8
	5
10	4
99	3
13	2

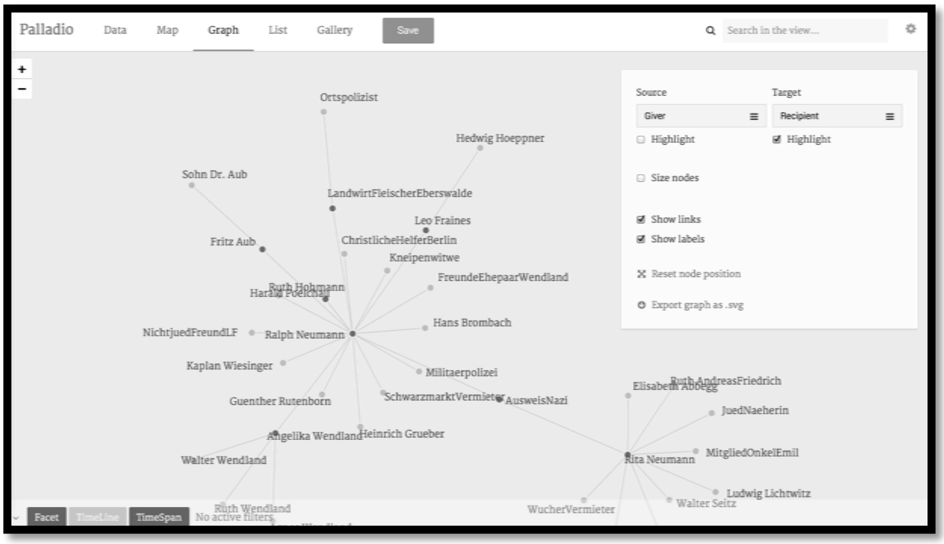
The Facet filter in Palladio

12. Bipartite network visualization. Now this is nice. But there is something else which makes Palladio a great tool to start out with network visualization: It makes it very easy to produce *bipartite, or 2-mode networks.*[393] What you have seen until now is a so-called unipartite or 1-mode network: It represents relations between source and target nodes of one type (for example "people") through one or more types of relations, Figures 13 and 14 are examples of this type of graph.

Network analysis however gives you a lot of freedom to rethink what source and targets are. Bipartite networks have two different types of nodes, an example could be to select "people" as the first node type and "point in time" as the second. Figure 15 shows a bipartite network and reveals which recipients of help were present in the network at the same time. Compare this graph to Figure 16 which shows which givers of help were present at the same time. This points at a high rate of fluctuation among helpers, an observation which holds true for all of the networks I studied. While humans are very good at processing people-to-people networks, we find it harder to process these more abstract networks. Give it a try and experiment with different bipartite networks: Click again on "Target" but this time select "Form of Help" or "Sex" or any other category.

Note that if you wanted to see "Giver" and "Recipients" as one node type and "Date of Activity" as the second, you would need to create one column with all the persons and a second with the points in time during which they were present in your spreadsheet editor and import this data into Palladio. Also, at this stage Palladio does not yet let you represent attribute data for example by coloring the nodes, but all other tools have this functionality.

[393] 'Bipartite graphs', *Wikipedia*: https://en.wikipedia.org/wiki/Bipartite_graph#Examples



Source: Giver ≡
Target: Recipient ≡

□ Highlight
☑ Highlight

□ Size nodes

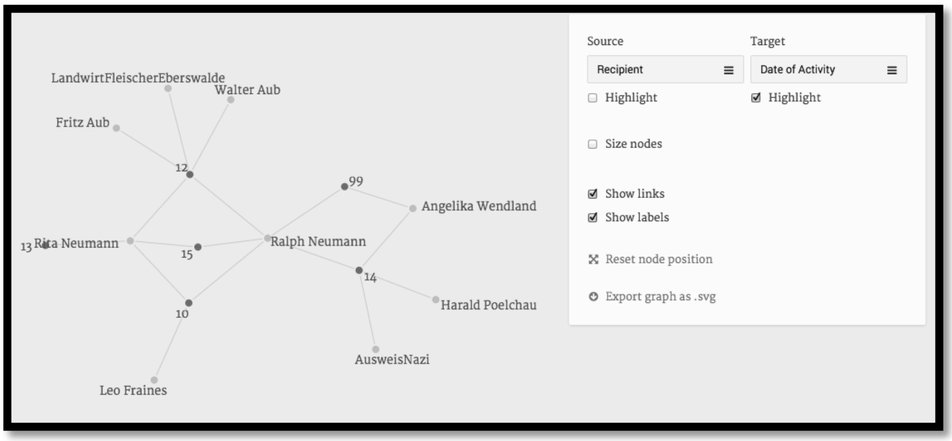
☑ Show links

☑ Show labels

Reset node position

Export graph as .svg

Visualization of a unipartite network: Givers and Recipients of help



Source: Recipient ≡
Target: Date of Activity ≡

□ Highlight
☑ Highlight

□ Size nodes

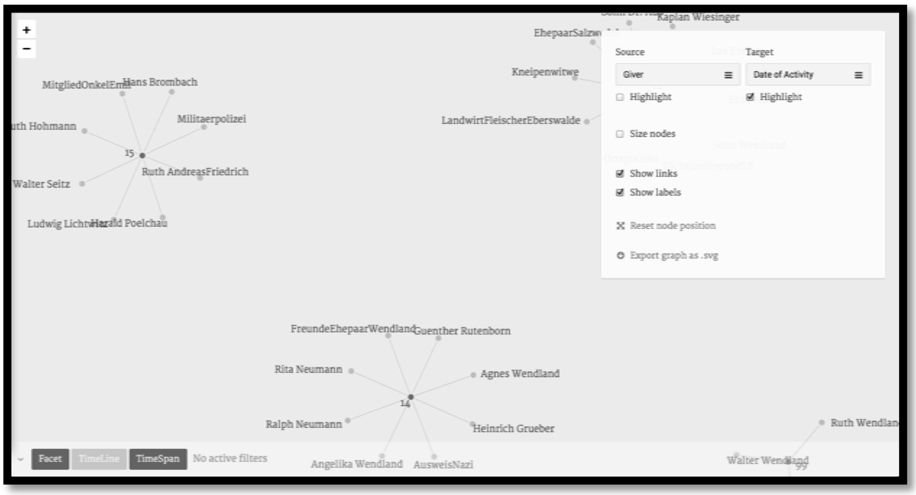
☑ Show links

☑ Show labels

Reset node position

Export graph as .svg

Visualization of a bipartite network: Recipients and Date of Activity



Source: Giver ≡
Target: Date of Activity ≡

□ Highlight
☑ Highlight

□ Size nodes

☑ Show links

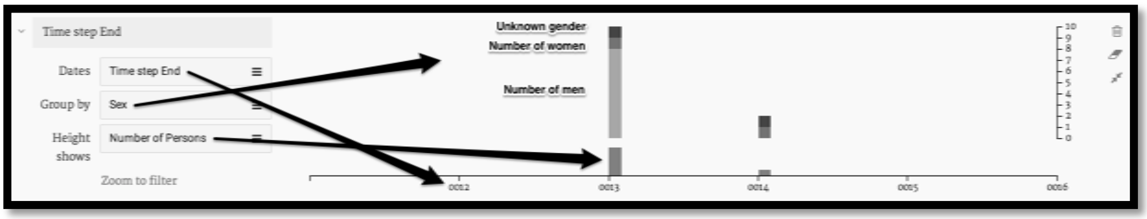
☑ Show labels

Reset node position

Export graph as .svg

Visualization of a bipartite network: Givers and Date of Activity

13. Timeline. The Timeline feature provides a relatively easy way to visualize changes in your network. Figure 17 shows the distribution of men and women in the network over time. The first column on the y-axis corresponds to the “Dates” field and represents the different time steps. The bars represent the “Sex” attribute: Unknown, numbers of women and men are represented by the height of the segments in a bar (ranging from light grey to black). Hover over them to see what is what. The lower bar segment corresponds to the “Height shows” field and here represents the total number of persons which changes between time step 13 and 14.



Time step End

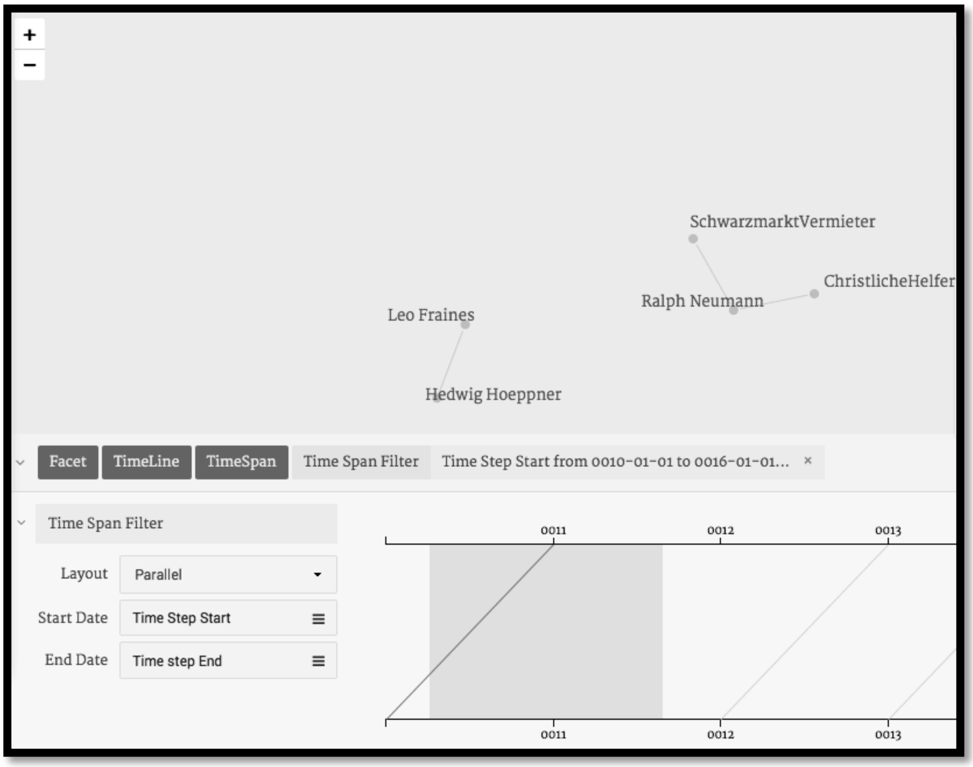
Dates: Time step End

Group by: Sex

Height shows: Number of Persons

Gender distribution in the network over time

14. Time Span. Even more interesting is the Time Span view which updates the network visualization dynamically. Click on “Time Span”. Figure 17 illustrates what you should see now. Use the mouse to highlight a section between the time steps which will then be highlighted in grey. You can now drag the highlighted section across the timeline and see how the graph changes from time step to time step.



Time Span Filter: Time Step Start from 0010-01-01 to 0016-01-01...

Time Span Filter

Layout: Parallel

Start Date: Time Step Start

End Date: Time step End

Timeline. Visualization of Time Steps

15. Node size. Palladio lets you size your nodes based on actor attributes. Note that this does not make sense for the sample data given that numerical values represent categories. Node sizes can however be useful if you were to represent the sum of a person's acts of help, which in this case would correspond to his or her *Out-Degree,*[394] the number of outgoing relations for a node.

16. Export your visualizations. Palladio lets you export your network as .svg files, a vector-based image format. Use your browser of choice to open them.

17. Lists, Maps and Galleries. You will have noticed that Palladio has a variety of additional visualization formats: Lists, Maps, and Galleries. All of which are as intuitive and well-designed as the Graph section. Galleries let you specify certain attributes of your actors and present them in a card-view. By adding latitude/longitude values to your actor attributes you will get an instant sense of where your network happens. Take a look at their own sample files to explore this.

The added value of network visualizations

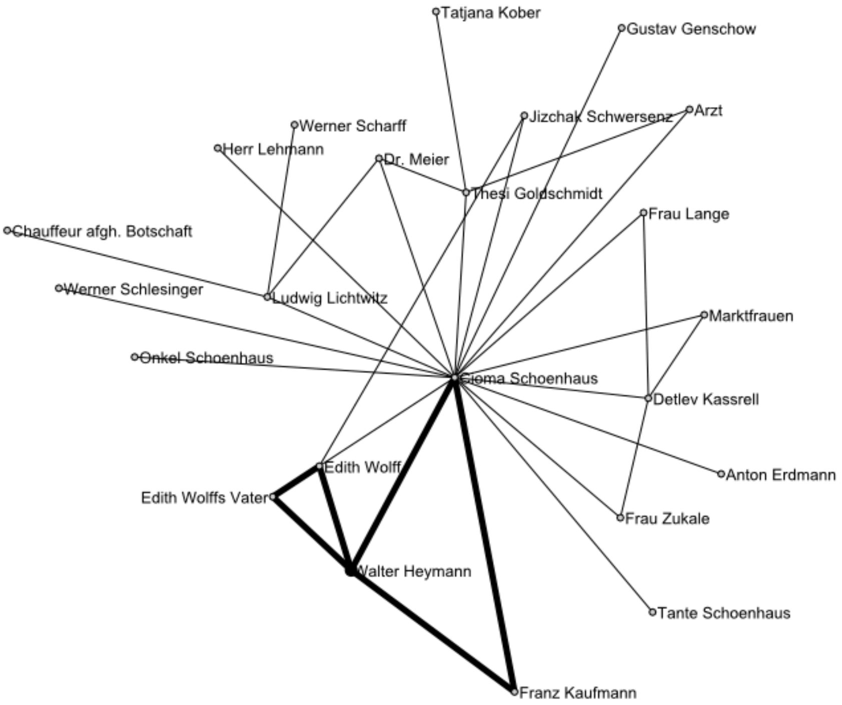
Careful extraction of network data from text is time consuming and exhausting since it requires full concentration at every step along the way. I regularly asked myself whether it was worth it--and in the end whether or not I could have made the same observations without the support of network visualizations. The answer is yes, I might have come to the same main conclusions without coding all this data and yes, it was worth it. Entering the relational data soon becomes fast and painless in the process of close reading.

In my experience, question-driven close reading and interpretation on one side and data coding and visualization on the other are not at all separate processes but intertwined and they can complement each other very effectively. Play is not generally considered to be something very academic, but especially with this type of data it is a valuable investment of your time: You don't just play with your data, you rearrange and thereby constantly rethink what you know about your topic and what you *can* know about your topic.

Each tie I coded represents a story of how somebody helped somebody else. Network visualizations helped me understand how these ca. 5,000 stories and 1,400 individuals relate to each other. They often confirmed what I knew but regularly also surprised me and raised interesting questions. For example, it led me to identify Walter Heymann as the person whose contact brokerage started off two major support networks and subsequently enabled them to save hundreds of people. Descriptions of his contacts to leading actors in both networks were scattered across different documents

[394] 'Indegree and outdegree', *Wikipedia*: https://en.wikipedia.org/wiki/Directed_graph#Indegree_and_outdegree

which I had worked on during different phases of the project. The visualization aggregated all these relations and revealed these connections. Further investigation then showed that it was in fact him who brought all of them together.



Walter Heymann brokered contacts which led to the emergence of two major support networks

On other occasions, visualizations revealed the existence of long reaching contact chains across different social classes which helped refugees create trusted ties with strangers, they also showed unexpected gaps between actors I expected to be connected, led me to identify clusters in overlapping lists of names, observe phases of activity and inactivity, helped me spot people who bridged different groups and overall led me to emphasize the contact brokerage of Jewish victims of persecution as a major, hitherto overlooked factor in the emergence of covert networks.

Visualisations are of course not “proof” of anything but tools to help understand complex relations; their interpretation is based on a good understanding of the underlying data and how it was visualized. Selected network visualizations can also accompany text and help your readers better understand the complex relationships you discuss, much like the maps you sometimes find on the inside covers of old books.

A few practical points:

Collect and store data in one spreadsheet and use a copy for visualizations

Make sure you understand the basic rationale behind any centrality and layout algorithms you choose as they will affect your view on your data. Wikipedia is usually a good source for comprehensive information on them. Don't hesitate to revise and start over if you sense that your coding scheme does not work out as expected. It will definitely be worth it.

Finally, any of the visualizations you can create with the small sample dataset I provide for this tutorial requires context knowledge to be really meaningful. The only way for you to find out whether this method makes sense for your research is to start coding your own data and to use your own context knowledge to make sense of your visualizations.

Good luck!

Other network visualization tools to consider

Nodegoat[395] – similar to Palladio in that it makes data collection, mapping and graph visualizations easy. Allows easy setup of relational databases and lets users store data on their servers. *Tutorial available.*[396]

NodeXL[397] – capable to perform many tasks common in SNA, easy-to-use, open source but requires Windows and MS Office 2007 or newer. *Tutorial 1,*[398] *Tutorial 2.*[399]

Gephi – open source, platform independent.[400] The best known and most versatile visualization tool available but expect a steep learning curve. The developers announce support for parallel edges in version 1.0. Tutorials: by *Clement Levallois*[401] and *Sebastien Heymann.*[402]

VennMaker – is platform-independent and can be tested for free.[403] VennMaker inverts the process of data collection: Users start with a customizable canvas and draw self-defined nodes and relations on it. The tool collects the corresponding data in the background.

The most commonly used tools for more mathematical analyses are *UCINET*[404](licensed, tutorials available on their website) and *Pajek*[405]

[395] 'Nodegoat': http://nodegoat.net/

[396] Yanan Sun, 'Geo-Layout for Dynamic Network in Nodegoat': http://nodegoat.net/cms/UPLOAD/AsmallguidebyYanan11082014.pdf

[397] 'NodeXL': http://nodexl.codeplex.com/

[398] 'NodeXL Tutorial (part 1 of 3)', *YouTube*: https://www.youtube.com/watch?v=pwsImFyc0lE

[399] 'Introduction to NodeXL – a', *YouTube:* https://www.youtube.com/watch?v=xKhYGRpbwOc

[400] 'Gephi': https://gephi.org/

[401] Clement Levallois, 'Teaching and Training Material': http://www.clementlevallois.net/training.html

[402] Sebastien Heymann, 'Sebastian Heymann Exploratory Network Analysis with Gephi Part 2': https://www.youtube.com/watch?v=L6hHv6y5GsQ

[403] 'Venn Maker': http://www.vennmaker.de/

[404] 'UCINET Software': https://sites.google.com/site/ucinetsoftware/home

[405] 'Pajek': http://mrvar.fdv.uni-lj.si/pajek/

(free) for which a great *handbook* exists.[406] Both were developed for Windows but run well elsewhere using Wine.

For Python users the very well documented package *Networkx*[407] is a great starting point; other packages exist for other programming languages.

About the Author

Marten Düring is a historian, works as researcher in the Digital Humanities Lab at CVCE Luxembourg, runs http://historicalnetworkresearch.org and regularly teaches workshops on network analysis.

[406] Woultor de Nooy, Andrej Mrvar, Vladimir Batagelj, *Exploratory Social Network Analysis with Pajek* (2nd edition), 2011.
[407] 'NetworkX': https://networkx.github.io/

35. Supervised Classification: The Naive Bayesian Returns to the Old Bailey

Vilja Hulden – 2014

Introduction

A few years back, William Turkel wrote a series of blog posts called A Naive Bayesian in the Old Bailey,[408] which showed how one could use machine learning to extract interesting documents out of a digital archive. This tutorial is a kind of an update on that blog essay, with roughly the same data but a slightly different version of the machine learner.

The idea is to show why machine learning methods are of interest to historians, as well as to present a step-by-step implementation of a supervised machine learner. This learner is then applied to the Old Bailey digital archive,[409] which contains several centuries' worth of transcripts of trials held at the Old Bailey in London. We will be using Python for the implementation.

One obvious use of machine learning for a historian is document selection. If we can get the computer to "learn" what kinds of documents we want to see, we can enlist its help in the always-time-consuming task of finding relevant documents in a digital archive (or any other digital collection of documents). We'll still be the ones reading and interpreting the documents; the computer is just acting as a fetch dog of sorts, running to the archive, nosing through documents, and bringing us those that it thinks we'll find interesting.

What we will do in this tutorial, then, is to apply a machine learner called Naive Bayesian to data from the Old Bailey digital archive. Our goals are to learn how a Naive Bayesian works and to evaluate how effectively it classifies documents into different categories - in this case, trials into offense categories (theft, assault, etc.). This will help us determine how useful a machine learner might be to us as historians: if it does well at this classification task, it might also do well at finding us documents that belong to a "class" we, given our particular research interests, want to see.

Step by step, we'll do the following:

learn what machine learners do, and look more closely at a popular learner called Naive Bayesian.
download a set of trial records from the Old Bailey archive.

[408] William J. Turkel, 'A Naïve Bayesian in the Old Bailey, Part 1', *Digital History Hacks (2005-08)*, (24 May 2008): http://digitalhistoryhacks.blogspot.co.uk/2008/05/naive-bayesian-in-old-bailey-part-1.html

[409] Tim Hitchcock, Robert Shoemaker, Clive Emsley, Sharon Howard and Jamie McLaughlin, *et al.*, *The Old Bailey Proceedings Online, 1674-1913* (www.oldbaileyonline.org, version 7.0, 24 March 2012).

write a script that saves the trials as text (removing the XML markup) and does a couple of other useful things.

write a couple of helper scripts to assist in testing the learners.

write a script that tests the performance of the learner.

Files you will need

```
save-trialtxts-by-category.py
tenfold-crossvalidation.py
count-offense-instances.py
pretestprocessing.py
test-nb-learner.py
naivebayes.py
english-stopwords.txt
```

A zip file of the scripts is available [http://programminghistorian.org/assets/baileycode.zip]. You can also download another zip file containing the scripts, the data that we are using and the files that result from the scripts [http://dx.doi.org/10.5281/zenodo.13284]. (The second option is probably easiest if you want to follow along with the lesson, since it gives you everything you need in the correct folder structure.) More information about where to put the files is in the "Preliminaries" section of the part where we actually begin to code.

Note: You will not need any Python modules that don't come with standard installations, except for *BeautifulSoup*[410] (used in the data creation step, not in the learner code itself).

The Old Bailey Digital Archive

The Old Bailey digital archive contains 197,745 criminal trials held at the Old Bailey, aka the Central Criminal Court in London. The trials were held between 1674 and 1913, and since the archive provides the full transcript of each trial, many of which include testimony by defendants, victims, and witnesses, it's a great resource for all kinds of historians interested in the lives of ordinary people in London.

What makes the collection particularly useful for our purposes is that the text of each trial is richly annotated with such information as what type of an offense was involved (pocketpicking, assault, robbery, conspiracy...), the name and gender of each witness, the verdict, etc. What's more, this information has been added to the document in XML markup, which allows us to extract it easily and reliably. That, in turn, lets us train a machine learner to recognize the things we are interested in, and then test the learner's performance.

[410] 'Beautiful Soup': http://www.crummy.com/software/BeautifulSoup/

Of course, in the case of the Old Bailey archive, we might not need this computer-assisted sorting all that badly, since the archive's curators, making use of the XML markup, offer us a ready-made search interface[411] that lets us look for documents by offense type, verdict, punishment, etc. But that's exactly what makes the Old Bailey such a good resource for testing a machine learner: we can check how well the learner performs by checking its judgments against the human-annotated information in the Old Bailey documents. That, in turn, helps us decide how (or whether) a learner could help us explore other digital document collections, most of which are not as richly annotated.

Machine learning

Machine learning can mean a lot of different things, but the most common tasks are classification and clustering.[412]

Classification is performed by supervised learners — "supervised" meaning that a human assistant helps them learn, and only then sends them out to classify by themselves. The basic training procedure is to give the learner labeled data: that is, we give it a stack of things (documents, for example) where each of those things is labeled as belonging to a group. This is called training data. The learner then looks at each item in the training data, looks at its label, and learns what distinguishes the groups from each other. To see how well the learner learned, we then test it by giving it data that is similar to the training data but that the learner hasn't seen before and that is not labeled. This is called (you guessed it!) test data. How well the learner performs on classifying this previously-unseen data is a measure of how well it has learned.

The classic case of a supervised classifier is a program that separates junk email (spam) from regular email (ham). Such a program is "trained" by giving it a lot of spam and ham to look at, along with the information of which is which. It then builds a statistical model of what a spam message looks like versus what a regular email message looks like. So it learns that a message is more likely to be spam if it contains sexual terms, or words like "offer" and "deal", or, as things turn out, "ff0000," the HTML code for red.[413] It can then apply that statistical model to incoming messages and discard the ones it identifies as spam.

Clustering is usually a task for unsupervised learners. An unsupervised learner doesn't get any tips on how the data "ought" to be sorted, but rather is expected to discover patterns in the data automatically, grouping the data by the patterns it has discovered. Unlike in supervised classification, in unsupervised clustering we don't tell the learner what the "right" groups are, or give it any hints on what items in the data set should go together.

[411] 'Search Home', *The Old Bailey Online*: http://www.oldbaileyonline.org/forms/formMain.jsp

[412] 'Statistical Classification', *Wikipedia*: https://en.wikipedia.org/wiki/Statistical_classification; 'Clustering Algorithms': http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/

[413] 'A Plan for Spam' (August 2002): http://www.paulgraham.com/spam.html

Rather, we give it data with a bunch of features, and (often, but not always) we tell it how many groups we want it to create. The features could be anything: in document clustering, they are normally words. But clustering isn't limited to grouping documents: it could also be used in, say, trying to improve diagnoses by clustering patient records. In that task, the features would be various attributes of the patient (age, weight, blood pressure, presence and quality of various symptoms etc.) and the clustering algorithm would attempt to create groups that share as many features as closely as possible.

A side note: Some of you may have come to think of an objection to this supervised/unsupervised distinction: namely, that the clustering method is not entirely "unsupervised" either. After all, we tell it what features it should look at, whether it is words (rather than sentences, or two-word sequences, or something else) in a document, or a list of numeric values in a patient record. The learner never encounters the data entirely unprepared. Quite true. But no matter - the distinction between unsupervised and supervised is useful nevertheless, in that in one we tell the learner what the right answer is, and in the other it comes to us with some pattern it has figured out without an answer key. Each is useful for different kind of tasks, or sometimes for different approaches to the same task.

In this tutorial, we are dealing with a supervised learner that we train to perform document classification. We give our learner a set of documents along with their correct classes, and then test it on a set of documents they haven't seen, with the hope that it will succeed in guessing the document's correct classification.

A Naive Bayesian learner

A Naive Bayesian is a supervised learner: we give it things marked with group labels, and its job is basically to learn the probability that a thing that looks a particular way belongs in a particular group.

But why "naive"? And what "Bayesian"?

"Naive" simply means that the learner makes the assumption that all the "features" that make up a document are independent of each other. In our case, the features are words, and so the learner assumes that the occurrence of a particular word is completely independent of the occurrence of another word. This, of course, is often not true, which is why we call it "naive." For example, when we put "new" and "york" together to form "New York," the result has a very different meaning than the "new" and "york" in "New clothes for the Prince of York." If we were to distinguish "New York" from "New" and "York" occurring separately, we might find that each tends to occur in very different types of documents, and thus not identifying the expression "New York" might throw our classifier off course.

Despite their simplistic assumption that the occurrence of any particular feature is independent of the occurrence of other features, Naive Bayesian classifiers do a good enough job to be very useful in many contexts (much of the real-world junk mail detection is performed by Naive Bayesian classifiers, for example). Meanwhile, the assumption of independence means that processing documents is much less computationally intensive, so a Naive Bayesian classifier can handle far more documents in a much shorter time than many other, more complex methods. That in itself is useful. For example, it wouldn't take too long retrain a Naive Bayesian learner if we accumulated more data. Or we could give it a bigger set of data to begin with; a pile of data that a Naive Bayesian could burrow through in a day might take a more complex method weeks or even months to process. Especially when it comes to classification, more data is often as significant as a better method — as Bob Mercer of IBM famously quipped in 1985, "there is no data like more data."

As for the "Bayesian" part, that refers to the 18th-century English minister, statistician, and philosopher Thomas Bayes. When you google for "Naive Bayesian," you will turn up a lot of references to "Bayes' theorem" or "Bayes' rule," which is a formula for applying conditional probabilities (the probability of some thing X, given some other thing Y).

Bayes' theorem is related to Naive Bayesian classifiers, in that we can formulate the classification question as "what is the probability of document X, given class Y?" However, unless you've done enough math and probability to be comfortable with that kind of thinking, it may not provide the easiest avenue to grasping how a Naive Bayesian classifier works. Instead, let's look at the classifier in a more procedural manner. (Meanwhile, if you prefer, you can check out an explanation of Bayes' rule and conditional probabilities[414] that does a very nice job and is also a good read.)

Understanding Naive Bayesian classification using a generative story

To understand Naive Bayesian classification, we will start by telling a story about how documents come into being. Telling such a story — called a "generative story" in the business — often simplifies the probabilistic analysis and helps us understand the assumptions we're making. Telling the story takes a while, so bear with me. There is a payoff at the end: the story directly informs us how to build a classifier under the assumptions that the particular generative story makes.

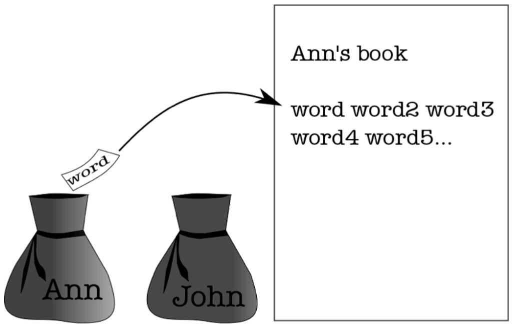
The fundamental assumption we will make in our generative story is that documents come into being not as a result of intellectual cogitation but as a result of a process whereby words are picked at random out of a bag and then put into a document (known as a bag-of-words model).

[414] Eliezer S. Yudkowsky, 'An Intuitive Explanation of Bayes' Theorem': http://www.yudkowsky.net/rational/bayes

So we pretend that historical works, for example, are written in something like the following manner. Each historian has his or her own bag of words with a vocabulary specific to that bag. So when Ann the Historian writes a book, what she does is this:

She goes to the bag that is her store of words.
She puts her hand in and pulls out a piece of paper.
She reads the word on the piece of paper, writes it down in her book, and puts the paper back in the bag.
Then she again puts her hand in the bag and pulls out a piece of paper.
She writes down that word in the book, and puts the piece of paper back in the bag.

Ann the Historian keeps going until she decides her book (or article, or blog post, or whatever) is finished. The next time she wants to write something, she goes back to her bag of words and does the same thing. If her friend John the Historian were to write a book, he would go to his own bag, which has a different set of words, and then he would follow the same procedure of taking out a word, writing it down, putting it back in. It's just one damn word after another.



Bags of Words

(If this procedure sounds familiar, that may be because it sounds a bit like the generative story told in explaining how *topic modeling* works.[415] However, the story in topic modeling is a bit different in that, for instance, each document contains words from more than one class. Also, you should note that topic modeling is unsupervised — you don't tell the modeler what the "right" topics are, it comes up with them all by itself.)

So let's say you are a curator of a library of historical works, and one day you discover a huge forgotten trunk in the basement of the library. It turns out that the trunk contains dozens and dozens of typed book manuscripts. After some digging, you find a document that explains that these are transcripts of unpublished book drafts by three historians: Edward Gibbon, Carl Becker, and Mercy Otis Warren.

[415] Shawn Graham, Scott Weingart, and Ian Milligan, 'Getting Started with Topic Modeling and MALLET', *The Programming Historian* (2012).

What a find! But unfortunately, as you begin sorting through the drafts, you realize that they are not marked with the author's name. What can you do? How can you classify them correctly?

Well, you do have other writings by these authors. And if historians write their documents in the manner described above — if each historian has his or her own bag of words with a particular vocabulary and a particular distribution of words — then we can figure out who wrote each document by looking at the words it contains and comparing the distribution of those words to the distribution of words in documents we *know* were written by Gibbon, Becker, and Warren, respectively.

So you go to your library stacks and get out all the books by Gibbon, Becker, and Warren. Then you start counting. You start with Edward Gibbon's *oeuvre*. For each word in a work by Gibbon, you add the word to a list marked "Gibbon." If the word is already in the list, you add to its count. Then you do the same with the works of Mercy Otis Warren and Carl Becker. Finally, for each author, you add up the total number of words you've seen. You also add up the total number of monographs you have examined so you'll have a metric for how much work each author has published.

So what you end up with is something like this:

Edward Gibbon (5)	Carl Becker (18)	Mercy Otis Warren (2)
empire, 985	everyman, 756	revolution, 989
rome, 897	revolution, 699	constitution, 920
fall, 887	philosopher, 613	principles, 899
...	...	...
(total), 352,003	(total), 745,532	(total), 300,487

What you have done, in essence, is to reconstruct each historian's "bag of words" — now you know (at least approximately) what words each historian uses and in what proportions. Armed with this representation of the word distributions in the works of Gibbons, Becker, and Warren, you're ready to tackle the task of figuring out who wrote which manuscripts.

You're going to work manuscript by manuscript and author by author, first pretending that the manuscript you're currently considering was written by Gibbons, then that it was written by Becker, and so on. For each author, you calculate how likely it is that the manuscript really was written by that author.

So with your first manuscript in hand, you start by assuming that the manuscript was written by Gibbons. First you figure out the overall probability of any monograph being written by Gibbons rather than either of the two others — that is, of the Gibbons bag rather than the Becker bag or the Warren bag being used to produce a monograph. You do this by taking the number of books written by Gibbons and dividing it by the total number of books written by all these authors. That comes out to 5/25, or 0.2 (20 percent).

Then, you start looking at the words in the manuscript. Let's say the first word is "fall." You check how often that word occurred in Gibbons' published *oeuvre*, and you find that the answer is 887. Then you check how many words, overall, there were in Gibbons' total works, and you note that the answer is 352,003. You divide 887 by 352,003 to get the proportional frequency (call it *p*) of "fall" in Gibbons' work (0.0025). For the next word, you do the same procedure, and then multiply the probabilities together (you multiply since each action — picking an author, or picking a word — represents an independent choice). In the end you end with a tally like this:

```
p_bag * p_word_1 * p_word_2 * ... * p_word_n
```

Note that including the probability of picking the bag (*p_bag*) is an important step: if you only go by the words in the manuscript and ignore how many manuscripts (or rather, published works) each author has written, you can easily go wrong. If Becker has written ten times the number of books that Warren has, it should reasonably require much firmer evidence in the form of an ample presence of "Warrenesque" words to assume that a manuscript was written by Warren than that it was written by Becker. "Extraordinary claims require extraordinary evidence," as Carl Sagan once said.

OK, so now you have a total probability of the manuscript having been written by Gibbons. Next, you repeat the whole procedure with the assumption that maybe it was instead written by Becker (that is, that it came out of the bag of words that Becker used when writing). That done, you move on to considering the probability that the author was Warren (and if you had more authors, you'd keep going until you had covered each of them).

When you're done, you have three total probabilities — one probability per author. Then you just pick out the largest one, and, as they say, Bob's your uncle! That's the author who most probably wrote this manuscript.

(Minor technical note: when calculating

```
p_bag * p_word1 * ... * p_word_n
```

in a software implementation we actually work with the logarithms[416] of the probabilities since the numbers easily become very small. When doing this, we actually calculate

```
log(p_bag) + log(p_word1) + ... + log(p_word_n)
```

[416] Kalid Azad, 'Using Logarithms in the Real World', *Better Explained*: http://betterexplained.com/articles/using-logs-in-the-real-world/

That is, our multiplications turn into additions in line with the rules of logarithms. But it all works out right: the class with the highest number at the end wins.)

But wait! What if a manuscript contains a word that we've never seen Gibbons use before, but also lots of words he used all the time? Won't that throw off our calculations?

Indeed. We shouldn't let outliers throw us off the scent. So we do something very "Bayesian": we put a "prior" on each word and each class — we pretend we've seen all imaginable words at least (say) once in each bag, and that each bag has produced at least (say) one document. Then we add those fake pretend counts — called priors,[417] or pseudocounts — to our real counts. Now, no word or bag gets a count of zero.

In fact, we can play around with the priors as much as we like: they're simply a way of modeling our "prior belief" in the probability of one thing over another. They could model our assumptions about a particular author being more likely than others, or a particular word being more likely to have come from the bag of a specific author, and so on. Such beliefs are "prior" in the sense that we hold the belief before we've seen the evidence we are considering in the actual calculation we are making. So above, for example, we could add a little bit to Mercy Otis Warren's *p_bag* number if we thought it likely that as a woman, she might well have had a harder time getting published, and so there might reasonably be more manuscript material from her than one might infer from a count of her published monographs.

In some cases, priors can make a Naive Bayesian classifier much more usable. Often when we're classifying, after all, we're not after some abstract "truth" — rather, we simply want a useful categorization. In some cases, it's much more desirable to be mistaken one way than another, and we can model that with proper class priors. The classic example is, again, sorting email into junk mail and regular mail piles. Obviously, you really don't want legitimate messages to be deleted as spam; that could do much more damage than letting a few junk messages slip through. So you set a big prior on the "legitimate" class that causes your classifier to only throw out a message as junk when faced with some hefty evidence. By the same token, if you're sorting the results of a medical test into "positive" and "negative" piles, you may want to weight the positive more heavily: you can always do a second test, but if you send the patient home telling them they're healthy when they're not, that might not turn out so well.

So there you have it, step by step. You have applied a Naive Bayesian to the unattributed manuscripts, and you now have three neat piles. Of course, you should keep in mind that Naive Bayesian classifiers are not

[417] 'Prior Distributions': http://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug_introbayes_sect004.htm

perfect, so you may want to do some further research before entering the newfound materials into the library catalog as works by Gibbons, Becker, and Warren, respectively.

OK, so let's code already!

So, our aim is to apply a Naive Bayesian learner to data from the Old Bailey. First we get the data; then we clean it up and write some routines to extract information from it; then we write the code that trains and tests the learner.

Before we get into the nitty-gritty of downloading the files and examining the training/testing script, let's just summarize what our aim is and what the basic procedure looks like.

We want to have our Naive Bayesian read in trial records from the Old Bailey and do with them the same thing as we did above in the examples about the works of Gibbons, Becker, and Warren. In that example, we used the published works of these authors to reconstruct each historian's bag of words, and then used that knowledge to decide which historian had written which unattributed manuscripts. In classifying the Old Bailey trials, we will give the learner a set of trials labeled with the offense for which the defendant was indicted so it can figure out the "bag of words" that is associated with that offense. Then the learner will use that knowledge to classify another set of trials where we have not given it any information about the offense involved. The goal is to see how well the learner can do this: how often does it label an unmarked trial with the right offense?

The procedure used in the scripts we employ to train the learner is no more complicated than the one in the historians-and-manuscripts example. Basically, each trial is represented as a list of words, like so:

```
michael, carney, was, indicted, for, stealing, on, the, 22nd, of, december,
26lb, weight, of, nails, value, 7s, 18, dozen, of, screws, ...
... , the, prisoners, came, to, my, shop, on, the, night, in, question, and,
 brought, in, some, ragged, pieces, of, beef, ...
..., i, had, left, my, door, open, and, when, i, returned, i, missed, all, t
his, property, i, found, it, at, the, pawnbroker, ...
```

When we train the learner, we give it a series of such word lists, along with their correct bag labels (correct offenses). The learner then creates word lists for each bag (offense), so that it ends up with a set of counts similar to the counts we created for Gibbons, Becker, and Warren, one count for each offense type (theft, deception, etc.)

When we test the learner, we feed it the same sort of word lists representing other trials. But this time we don't give it the information about what offense was involved. Instead, the learner does what we did above: when it gets a list of words, it compares that list to the word counts for each offense type, calculating which offense type has a bag of words most similar to this list of words. It works offense by offense, just like we worked author by author. So first it assumes that the trial involved, say, the offense "theft". It looks at the first word in the trial's word list, checks how often that word occurred in the "theft" bag, performs its probability calculations, moves on to the next word, and so on. Then it checks the trial's word list against the next category, and the next, until it has gone through each offense. Finally it tallies up the probabilities and labels the trial with the offense category that has the highest probability.

Finally, the testing script evaluates the performance of the learner and lets us know how good it was at guessing the offense associated with each trial.

Preliminaries

Many of the tools we are using to deal with the preliminaries have been discussed at Programming Historian before. You may find it helpful to check out (or revisit) the following tutorials:

Milligan & Baker, 'Introduction to the Bash Command Line'
Milligan, 'Automated Downloading with wget'
Knox, 'Understanding Regular Expressions'
Wieringa, 'Intro to Beautiful Soup'

A few words about the file structure the scripts assume/create:

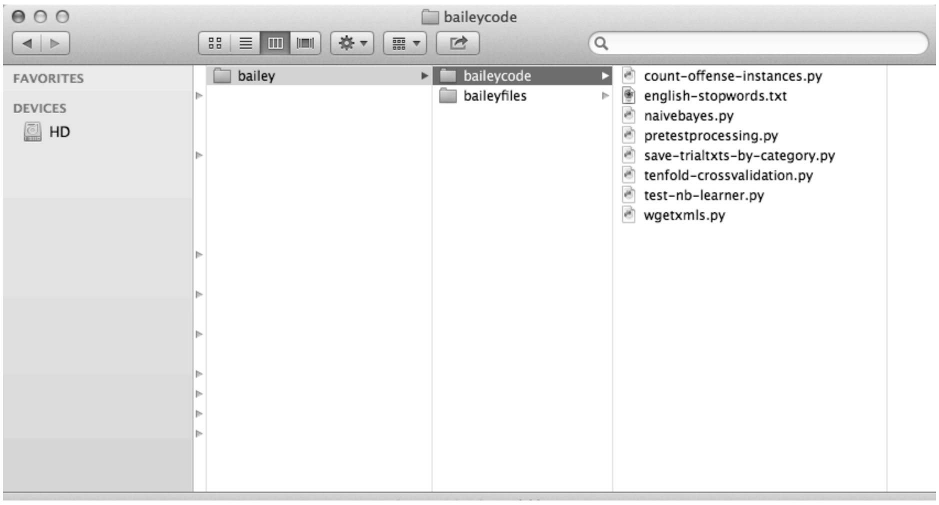
I have a "top-level" directory, which I'm calling *bailey* (you could call it something else, it's not referenced in the code). Under that I have two directories: *baileycode* and *baileyfiles*. The first contains all the scripts; the second contains the files that are either downloaded or created by the scripts. That in turn has subdirectories; all except one (for the downloaded XML files — see below) are created by the scripts.

If you downloaded the complete zip package with all the files and scripts, you automatically get the right structure; just unpack it in its own directory. The only files that are omitted from that are the zip files of trials downloaded below (if you got the complete package, you already have the unpacked contents of those files, and the zips would just take up unnecessary space).

If you only downloaded the scripts, you should do the following:

Create a directory and name it something sensible (say, *bailey*).
In that directory, create another directory called *baileycode* and unpack the contents of the script zip file into that directory (make sure you don't end up with two *baileycode* directories inside one another).
In the same directory (*bailey*), create another directory called *baileyfiles*.

On my Mac, the structure looks like this:



Bailey Folders

Downloading trials

The Old Bailey lets you download trials in zip files of 10 trials each, so that's what we'll do. This is how we do it: we first look at how the Old Bailey system requests files, and then we write a script that creates a file with a bunch of those requests. Then we feed that file to wget, so we don't have to sit by our computers all day downloading each set of 10 trials that we want.

As explained on the Old Bailey 'documentation for developers' page,[418] this is what the http request for a set of trials looks like:

```
http://www.oldbaileyonline.org/obapi/ob?term0=fromdate_18300114&term1=todate
_18391216&count=10&start=211&return=zip
```

As you see, we can request all trials that took place between two specified dates (*fromdate* and *todate*). The *count* specifies how many trials we want, and the *start* variable says where in the results to start (in the above, we start with result number 211 and get the ten following trials). Ten seems to be the highest number allowed for *count*, so we need to work around that.

We get around the restriction for how many trials can be in a zip file with a little script that builds as many of the above type of requests as we need to get all trials from the 1830s. We can find out how many trials that is by going to the Old Bailey 'search page'[419] and entering January 1830 as the start date, December 1839 as the end date, and choosing "Old Bailey Proceedings > trial accounts" in the "Search In" field. Turns out there were 22,711 trials in the 1830s.

Here's the whole script (`wgetxmls.py`) that creates the list of http requests we need:

[418] 'Documentation for Developers', *The Old Bailey Online*: http://www.oldbaileyonline.org/static/DocAPI.jsp

[419] 'Search Home', *The Old Bailey Online*: http://www.oldbaileyonline.org/forms/formMain.jsp

```
    mainoutdirname = '../baileyfiles/'
    wgets = ''
    for x in range(0,22711,10):
        getline = 'http://www.oldbaileyonline.org/obapi/ob?term0=fromdate_18
300114&term1=todate_18391216&count=10&start=' + str(x+1) + '&return=zip\n'
        wgets += getline
    filename = mainoutdirname + 'wget1830s.txt'
    with open(filename,'w') as f:
        f.write(wgets)
```

As you see, we accept the limitation of 10 trials at a time, but manipulate the start point until we have covered all the trials from the 1830s.

Assuming you're in the *baileycode* directory, you can run the script from the command line like this:

```
python wgetxmls.py
```

What that gets us is a file that looks like this:

```
http://www.oldbaileyonline.org/obapi/ob?term0=fromdate_18300114&term1=todate
_18391216&count=10&start=1&return=zip
http://www.oldbaileyonline.org/obapi/ob?term0=fromdate_18300114&term1=todate
_18391216&count=10&start=11&return=zip
http://www.oldbaileyonline.org/obapi/ob?term0=fromdate_18300114&term1=todate
_18391216&count=10&start=21&return=zip
...
```

This file is saved in the *baileyfiles* directory; it is called `wget1830s.txt`.

To download the trials, create a new directory under *baileyfiles*; call it *trialzips*. Then go into that directory and call *wget* with the file we just created. So, assuming you are still in the *baileycode* directory, you would write the following commands on the command line:

```
cd ../baileyfiles
mkdir trialzips
cd trialzips
wget -w 2 -i ../wget1830s.txt
```

The "-w 2" is just to be polite and not overload the server; it tells *wget* to wait 2 seconds between each request. The "-i" flag tells *wget* that it should request the URLs found in `wget1830s.txt`.

What *wget* returns is a lot of zip files that have unwieldy names and no extension. You should rename these so that the extension is ".zip". Then, in the directory *baileyfiles*, create a subdirectory called *1830s-trialxmls* and then unpack the zips into that so that it contains 22,170 XML files that each look like `t18391216-388.xml`. Assuming you are still in the *trialzips* directory, you would write:

```
for f in * ; do mv $f $f.zip; done;
mkdir ../1830s-trialxmls
unzip "*.zip" -d ../1830s-trialxmls/
```

If you open one of the trial XMLs in a browser, you can see that it contains all kinds of useful information: name and gender of defendant, name and gender of witnesses, type of offense, and so on. Here's a snippet from one trial:

```
<persname id="t18300114-2-defend110" type="defendantName">
THOMAS TAYLOR
    <interp inst="t18300114-2-defend110" type="surname" value="TAYLOR">
    <interp inst="t18300114-2-defend110" type="given" value="THOMAS">
    <interp inst="t18300114-2-defend110" type="gender" value="male">
    <interp inst="t18300114-2-defend110" type="age" value="25">
</interp></interp></interp></interp></persname>
was indicted for
    <rs id="t18300114-2-off7" type="offenceDescription">
        <interp inst="t18300114-2-off7" type="offenceCategory" value="violen
tTheft">
        <interp inst="t18300114-2-off7" type="offenceSubcategory" value="rob
bery">
            feloniously assaulting
        <persname id="t18300114-2-victim112" type="victimName">
            David Grant
                  <interp inst="t18300114-2-victim112" type="surname" value=
"Grant">
            <interp inst="t18300114-2-victim112" type="given" value="David">
            <interp inst="t18300114-2-victim112" type="gender" value="male">
            <join result="offenceVictim" targorder="Y" targets="t18300114-2-
off7 t18300114-2-victim112">
        </join></interp></interp></interp></persname>
</interp></interp></rs>
```

The structured information in the XML lets us reliably extract the "classes" we want to sort our documents into. We are going to classify the trials by offense category (and subcategory), so that's the information we're going to extract before converting the XML into a text file that we can then feed to our learner.

Saving the trials into text files

Now that we have the XML files, we can start extracting information and plain text from them to feed to our learner. We want to sort the trials into text files, so that each text file contains all the trials in a particular offense category (theft-simplelarceny, breakingpeace-riot, etc.).

We also want to create a text file that contains all the trial IDs (marked in the XML), so we can use that to easily create cross-validation samples. The reasons for doing this are discussed below in the section "Creating the cross-validation samples".

The script that does these things is called `save-txttrials-by-category.py` and it's pretty extensively commented, so I'll just note a few things here.

We strip the trial text of all punctuation, including quote marks and parentheses, and we equalize all spaces (newlines, tabs, multiple spaces) into a single space. This helps us simplify the coding of the training process (and, incidentally, keep the code that trains the learner general enough that as long as you have text files saved in the same format as we use here, you should be able to apply it more or less directly to your data).

That of course makes the text hard to read for a human. Therefore, we also save the text of each trial into a file named after the trial id, so that we can easily examine a particular trial if we want to (which we will).

The script creates the following directories and files under *baileyfiles*:

Directory *1830s-trialtxts*: this will contain the text file versions of the trials after they have been stripped of all XML formatting. Each file is named after the trial's ID.

Directory *1830s-trialsbycategory*: this will contain the text files that represent all the text in all the trials belonging to a particular category. These are named after the category, e.g., `theft-simplelarceny.txt`. Each category file contains all the trials in that category, with one trial per line.

File `trialids.txt`. This contains the sorted list of trial IDs, one ID per line; we will use it later in creating cross-validation samples for training the learner (this is the next step).

Files `offensedict.json` and `trialdict.json`. These json files will come into use in training the learner.

So if you're still in the *trialxmls* directory, you would write the following commands to run this script:

```
cd ../../baileycode/
python save-trialtxts-by-category.py
```

This will take a while. After it's done, you should have the directories and files described above.

Creating the cross-validation samples

Now that we have all our trials saved where we want them, all we need to do is to create the cross-validation samples and we're ready to test our learners.

Cross-validation simply means repeatedly splitting our data into chunks, some of which we use for training and others for testing. Since the idea is to get a learner to extract information from one set of documents that it can then use to determine the class of documents it has never seen, we obviously have to reserve a set of documents that are unknown to the learner if we want to test its performance. Otherwise it's a bit like letting your students first read an exam *and its answers* and then have them take that same exam. That would only tell you how closely they read the actual exam, not whether they've learned something more general.

So what you want to do is to test the learner on data it hasn't seen before, so that you can tell whether it has learned some general principles from the training data. You could just split your data into two sets, using, say, 80 percent for training and 20 percent for testing. But a common practice is to split your data repeatedly into different test and train sets, so that you can ensure that your test results aren't the consequence of some oddball quirk in the portion of data you left for testing.

Two scripts are involved in creating the cross-validation sets. The script `tenfold-crossvalidation.py` creates the samples. It reads in the list of trial ids we created in the previous step, shuffles that list to make it random, and divides it into 10 chunks of roughly equal length (that is, a roughly equal number of trial ids). Then it writes those 10 chunks each into its own text file, so we can read them into our learner code later. Next, to be meticulous, we can run the `count-offense-instances.py` to confirm that if we are interested in a particular trial category, that category is reasonably evenly distributed across the samples.

Before you run the `count-offense-instances.py` script, you should edit it to set the category to the one you're interested in and let the script know whether we're looking at a broad or a specific category. This is what the relevant part of the code looks like:

```
indirname = '../baileyfiles/'
offensedict_fn = indirname + 'offensedict.json'
offensecat = 'breakingpeace' #change to target category
broadcat = True #set true if category is e.g. "theft" instead of "theft-simp
lelarceny"
```

And here are the commands to run the cross-validation scripts (assuming you are still in the *baileycode* directory).

```
python tenfold-crossvalidation.py
python count-offense-instances.py
```

Alternatively, you can run them using pypy,[420] which is quite a bit faster.

```
pypy tenfold-crossvalidation.py
pypy count-offense-instances.py
```

The output of the `count-offense-instances.py` script looks like this:

```
Offense category checked for: breakingpeace
sample0.txt: 31
sample1.txt: 25
sample2.txt: 32
sample3.txt: 25
sample4.txt: 36
sample5.txt: 33
sample6.txt: 29
sample7.txt: 35
sample8.txt: 27
sample9.txt: 31
```

[420] ‘pypy’: http://pypy.org/

From the output, we can conclude that the distribution of instances of "breakingpeace" is more or less even. If it isn't, we can re-run the `tenfold-crossvalidation.py` script, and then check the distribution again.

Testing the learner

All right, we are ready train and test our Naive Bayesian! The script that does this is called `test-nb-learner.py`. It starts by defining a few variables:

```
categoriesdir = '../baileyfiles/1830s-trialsbycategory/'
sampledirname = '../baileyfiles/Samples_1830s/' #Location of 10-fold cross-v
alidation
stopwordfilename = '../baileyfiles/english-stopwords.txt'
# the ones below should be set to None if not using
cattocheck = 'breakingpeace' #if evaluating recognition one category against
 rest
pattern = '[^-]+' #regex pattern to use if category is not complete filename
```

Most of these are pretty self-explanatory, but note the two last ones. The variable "cattocheck" determines whether we are looking to identify a specific category or to sort each trial into its proper category (the latter is done if the variable is set to None). The variable "pattern" tells us whether we are using the whole file name as the category designation, or only a part of it, and if the latter, how to identify the part. In the example above, we are focusing on the broad category "breakingpeace", and so we are not using the whole file name, which would be e.g. "breakingpeace-riot" but only the part before the dash. Before you run the code, you should set these variables to what you want them to be.

Note that "cattocheck" here should match the "offensecat" that you checked for with the `count-offense-instances.py` script. No error is produced if it does not match, and it's fairly unlikely that it will have any real impact, but if the categories don't match, then of course you have no assurance that the category you're actually interested in is more or less evenly distributed across the ten cross-validation samples.

Note also that you can of course set "cattocheck" to "None" and leave the pattern as it is, in which case you will be sorting into the broader categories.

So, with the basic switches set and knobs turned, we begin by reading in all the trials that we have saved. We do this with the function called *process_data* that can be found in the `pretestprocessing.py` file. (That file contains functions that are called from the scripts you will run, so it isn't something you'll run directly at any point.)

```
print 'Reading in the data...'
trialdata = ptp.process_data(categoriesdir,stopwordfilename,cattocheck,patte
rn)
```

The *process_data* function reads in all the files in the directory that contains our trial category files, and processes them so that we get a list containing all the categories and the trials belonging to them, with the trial text lowercased and tokenized (split into a list of words), minus stopwords (common words like a, the, me, which, etc.) Each trial begins with its id number, so that's one of our words (though we ignore it in training and testing). Like this:

```
[
 [breakingpeace,
   ['trialid','victim','peace','disturbed','man','tree',...]
   ['trialid','dress','blood','head','incited',...]
  ...]
 [theft,
   ['trialid','apples','orchard','basket','screamed','guilty',....]
   ['trialid','rotten','fish']
  ...]
]
```

Next, making use of the results of the ten-fold cross-validation routine we created, we loop through the files that define the samples, each time making one sample the test set and the rest the train set. Then we split 'trialdata', the list of trials-by-category that we just created, into train and test sets accordingly. The functions that do these two steps are *create_sets* and *splittraintest*, both in the `pretestprocessing.py` file.

Now we train our Naive Bayesian classifier on the train set. The classifier we are using (which is included in the scripts zip file) is one written by Mans Hulden, and it does pretty much exactly what the "identify the author of the manuscript" example above describes.

```
# split train and test
print 'Creating train and test sets, run {0}'.format(run)
trainsetids, testsetids = ptp.create_sets(sampledirname,run)
traindata, testdata = ptp.splittraintest(trainsetids,testsetids,trialdata)

# train learner
print 'Training learner, run {0}...'.format(run)
mynb = nb.naivebayes()
mynb.train(traindata)
```

After the learner is trained, we are ready to test how well the it performs. Here's the code:

```
print 'Testing learner, run {0}...'.format(run)

for trialset in testdata:
    correctclass = trialset[0]
    for trial in trialset[1:]:
        result = mynb.classify(trial)
        guessedclass =  max(result, key=result.get)
        # then record correctness of classification result
        # note that first version does a more complex evaluation
        # ... for two-way (one class against rest) classification
        if cattocheck:
            if correctclass == cattocheck:
                catinsample += 1
            if guessedclass == cattocheck:
                 guesses += 1
                 if guessedclass == correctclass:
                     hits += 1
        if guessedclass == correctclass:
            correctguesses += 1

        total +=1
```

So we loop through the categories in the "testdata" list (which is of the same format as the "trialdata" list). For each category, we loop through the trials in that category, classifying each trial with our Naive Bayesian classifier, and comparing the result to the correct class (saved in the first element of each category list within the testdata list.) Then we add to various counts to be able to evaluate the results of the whole classification exercise.

To run the code that trains and tests the learner, first make sure you have edited it to set the "cattocheck" and "pattern" switches, and then call it on the command line (assuming you're still in the directory *baileycode*):

```
     python test-nb-learner.py
```

Again, for greater speed, you can also use pypy:

```
     pypy test-nb-learner.py
```

The code will print out some accuracy measures for the classification task you have chosen. The output should look something like this:

```
Reading in the data...
Creating train and test sets, run 0
Training learner, run 0...
Testing learner, run 0...
Creating train and test sets, run 1
Training learner, run 1...
...
Training learner, run 9...
Testing learner, run 9...
Saving correctly classified trials and close matches...
Calculating accuracy of classification...
Two-way classification, target category breakingpeace.
And the results are:
Accuracy 99.00%
Precision: 61.59%
Recall: 66.45%
Average number of target category trials in test sample per run: 30.4
Average number of trials in test sample per run: 2271.0
Obtained in 162.74 seconds
```

Next, let's take a look at what these measures of accuracy mean.

Measures of classification

The basic measure of classification prowess is *accuracy*: how often did classifier guess the class of a document correctly? This is calculated by simply dividing the number of correct guesses by the total number of documents considered.

If we're interested in a specific category, we can extract a bit more data. So if we set, for example, cattocheck = 'breakingpeace', like above, we can then examine how well the classifier did with respect to the "breakingpeace" category in particular.

So, in the testlearner code, if we're doing multiway classification, we only record how many trials we've seen ("total") and how many of our guesses were correct ("correctguesses"). But if we're considering a single category, say "breakingpeace," we record a few more numbers. First, we keep track of how many trials belonging to the category "breakingpeace" there are in our test sample (this tally is in "catinsample"). We also keep track of how many times we've guessed that a trial belongs to the "breakingpeace" category ("guesses"). And finally we record how many times we have guessed *correctly* that a trial belongs to "breakingpeace" ("hits").

Now that we have this information, we can use it to calculate a couple of standard measures of classification efficiency: *precision* and *recall*. Precision tells us how often we correctly guessed that a trial was in the "breakingpeace" category. Recall lets us know what proportion of the "breakingpeace" trials we caught.

Let's take another example to clarify precision and recall. Imagine you want all the books on a particular topic — World War I, say — from your university library. You send out one of your many minions (all historians possess droves of them, as you know) to get the books. The minion dutifully returns with a big pile.

Now, suppose you were in possession of a list that contained of every single book in the library on WWI and no books that weren't related to the WWI. You could then check the precision and recall of your minion with regard to the category of "books on WWI."

Recall is the term for the proportion of books on WWI in the library that your minion managed to grab. That is, the more books on WWI remaining in the library after your minion's visit, the lower your minion's recall.

Precision, in turn, is the term for the proportion of books in the pile brought by your minion that actually had to do with WWI. The more irrelevant (off-topic) books in the pile, the lower the precision.

So, say the library has 1,000 books on WWI, and your minion lugs you a pile containing 400 books, of which 300 have nothing to do with WWI. The minion's recall would be (400-300)/1,000, or 10 percent. The minion's precision, in turn, would be (400-300)/400, or 25 percent.

(Should have gone yourself, eh?)

A side note: the minion's overall accuracy — correct guesses divided by actual number of examples — would be:

```
(the number of books on WWI in your pile - the number of books *not* on
WWI in your pile + the number of books in the library *not* on WWI)
-----------------------------------------------------------------------
the total number of books in the library
```

So if the library held 100,000 volumes, this would be (100 - 300 + 99,000) / 100,000 — or 98.8 percent. That seems like a great number, but since it merely means that your minion was smart enough to leave most of the library books in the library, it's not very helpful in this case (except inasmuch as it is nice not to be buried under 100,000 volumes.)

How well does our Naive Bayesian do?

Our tests on the Naive Bayesian use the data set consisting of all the trials from the 1830s. It contains 17,549 different trials in 50 different offense categories (which can be grouped into 9 broad categories).

If we run the Naive Bayesian so that it attempts to sort all trials into their correct broad categories, its accuracy is pretty good: 94.3 percent. So 94 percent of the time, when it considers how it should sort trials in the test sample into "breakingpeace," "deception," "theft," and so on, it chooses correctly.

For the more specific categories ("theft-simplelarceny," "breakingpeace-riot," and so on) the same exercise is much less accurate: then, the classifier gets it right only 72 percent of the time. That's no wonder, really, given that some of the categories are so small that we barely have any examples. We might do a bit better with more data (say, all the trials from the whole 19th century, instead of only all the trials from the 1830s).

The first, overall category results are pretty impressive. They give us quite a bit of confidence that if what we needed to do was to sort documents into piles that weren't all too fine-grained, and we had a nice bunch of training data, a Naive Bayesian could do the job for us.

But the problem for a historian is often rather different. A historian using a Naive Bayesian learner is more likely to want to separate documents that are "interesting" from documents that are "not interesting" — usually meaning documents dealing with a particular issue or not dealing with it. So the question is really more one where we have a mass of "uncategorized" or "other" documents and a much smaller set of "interesting" documents, and we try to find more of the latter among the former.

In our current exercise, that situation is fairly well represented by trying to identify documents from a single category in the mass of the rest of the documents, set to category "other." So how well are we able to do that? In other words, if we set cattocheck = 'breakingpeace' (or another category) so that all trials get marked as either that category or as "other," and then run the classifier, what kinds of results do we get?

Well, our overall accuracy is still high: over 95 percent in all cases for the broad categories, and usually about that for the detailed ones as well. But just like the minion going off to the library to get books on WWI had a pretty high accuracy because he/she didn't bring back half the library, in this case, too, our accuracy is mostly just due to the fact that we manage to not misidentify *too* many "other" trials as being in the category we're interested in. Because there are so many "other" trials, those correct assessments far outweigh the minus points we may have gotten from missing interesting trials.

Precision and recall, therefore, are more in this case more interesting measures than overall accuracy. Here's a table showing precision and recall for each of the "broad" categories in our trial sample, and for a few sample detailed categories. The last column shows how many target category trials there were in the test set on average (remember, we did ten runs with different train/test splits, so all our results are averages of that).

Naive Bayesian classifier, two-way classification, 10-fold cross-validation

Broad categories

Category	Precision (%)	Recall (%)	Avg # trials in cat in TeS
breakingpeace	63.52	64.05	24.2
damage	0.00	0.00	1.2
deception	53.47	61.43	47.7
kill	62.5	89.39	17.9
miscellaneous	47.83	4.44	24.8
royaloffenses	85.56	91.02	42.3
sexual	93.65	49.17	24.0
theft	96.26	98.75	1551.8
violenttheft	68.32	33.01	20.9

Sample detailed categories

Category	Precision (%)	Recall (%)	Avg # trials in cat in TeS
theft-simplelarceny	64.37	89.03	805.9
theft-receiving	92.21	61.53	198.1
deception-forgery	74.29	11.87	21.9
violenttheft-robbery	68.42	31.86	20.4
theft-extortion	0.00	0.00	1.3

There are a few generalizations we can make from these numbers.

First, it's obvious that if the category is too small, we are out of luck. So for "damage," a small enough broad category that our test samples only held a little over one instance of it on average, we get no results. Similarly, in the detailed categories, when the occurrence of cases per test sample drops into the single digits, we fail miserably. This is no wonder: if the test sample contains about one case on average, there can't be much more than ten cases total in the whole data set. That's not much to go on.

Second, size isn't everything. Although we do best for the biggest category, theft (which in fact accounts for over half our sample), there are some smaller categories we do very well for. We have very high recall and precision for "royaloffenses," a mid-sized category, and very high recall plus decent precision for "kill," our smallest reasonable-sized category. A reasonable guess would be that the language that occurs in the trials is distinctive and, in the case of "royaloffenses," doesn't occur much anywhere else. Meanwhile, unsurprisingly, we get low scores for the "miscellaneous" category. We also have high precision for the "sexual" category, indicating that it has some language that doesn't tend to appear anywhere else — though we miss about half the instances of it, which would lead us to suspect that many of the trials in that category omit some of the language that most distinguishes it from others.

Third, in this sample at least, there seems to be no clear pattern regarding whether the learner has better recall or better precision. Sometimes it casts a wide net that drags in both a good portion of the category and some driftwood, and sometimes it handpicks the trials for good precision but misses a lot that don't look right enough for its taste. So in half the cases here, our learner has better precision than recall, and in half better recall than precision. The differences between precision and recall are, however, bigger for the cases where precision is better than recall. That isn't necessarily a good thing for us, since as historians we might be happier to see more of the "interesting" documents and do the additional culling ourselves than to have our learner miss a lot of good documents. We'll return to the question of the meaning of classification errors below.

Extracting the most indicative features

The `naivebayes.py` script has a feature that allows you to extract the most (and least) indicative features of your classification exercise. This allows you to see what weighs a lot in the learner's mind — what it has, in effect, learned.

The command to issue is: *mynb.topn_print(10)* (for the 10 most indicative; you can put in any number you like). Here are the results for a multi-way classification of the broad categories in our data:

```
deception ['norrington', 'election', 'flaum', 'polish', 'caton', 'spicer', '
saltzmaun', 'newcastle', 'stamps', 'rotherham']
royaloffences ['mould', 'coster', 'coin', 'caleb', 'counterfeit', 'obverse',
 'mint', 'moulds', 'plaster-of-paris', 'metal']
violenttheft ['turfrey', 'stannard', 'millward', 'falcon', 'crawfurd', 'weat
herly', 'keith', 'farr', 'ventom', 'shurety']
damage ['cow-house', 'ewins', 'filtering-room', 'fisk', 'calf', 'skirting',
'girder', 'clipping', 'saturated', 'firemen']
breakingpeace ['calthorpe-street', 'grievous', 'disable', 'mellish', 'flag',
 'bodily', 'banner', 'aforethought', 'fursey', 'emerson']
miscellaneous ['trevett', 'teuten', 'reitterhoffer', 'quantock', 'feaks', 'b
oone', 'bray', 'downshire', 'fagnoit', 'ely']
kill ['vault', 'external', 'appearances', 'slaying', 'deceased', 'marchell',
 'disease', 'pedley', 'healthy', 'killing']
theft ['sheep', 'embezzlement', 'stealing', 'table-cloth', 'fowls', 'dwellin
g-house', 'missed', 'pairs', 'breaking', 'blankets']
sexual ['bigamy', 'marriage', 'violate', 'ravish', 'marriages', 'busher', 'r
egister', 'spinster', 'bachelor', 'married']
```

Some of these make sense instantly. In "breakingpeace" (which includes assaults, riots and woundings) you can see the makings of phrases like "grievous bodily harm" and "malice aforethought," along with other indications of wreaking havoc like "disable" and "harm." In royaloffenses, the presence of "mint," "mould" and "plaster-of-paris" make sense since the largest subcategory is coining offenses. In "theft," one might infer that sheep, fowl, and table-cloths seem to have been popular objects for stealing (though table-cloth may of course have been a wrapping for stolen objects; one would have to examine the trials to know).

Others are more puzzling. Why is violenttheft almost exclusively composed of what seem to be person or place names? Why is "election" indicative of deception? Is there a lot of election fraud going on, or abuse of elected office? Looking at the documents, one finds that 9 of the words indicative of violent theft are person names, and one is a pub; why person and pub names should be more indicative here than for other categories is mildly intriguing and might bear further analysis (or might just be a quirk of our data set — remember that "violenttheft" is a fairly small category). As for "election," it's hard to distinguish a clear pattern, though it seems to be linked to fraud attempts on and by various officials at different levels of government.

The indicative features, then, may be intriguing in themselves (though obviously, one should not draw any conclusions about them without closely examining the data first). They are also useful in that they can help us determine whether something is skewing our results in a way we don't wish, something we may be able to correct for with different weighting or different selection of features (see the section on Tuning below).

The meanings of misclassification

Again, it's good to keep in mind that in classifying documents we are not always after an abstract "true" classification, but simply a useful or interesting one. Thus, it is a good idea to look a bit more closely at the "errors" in classification.

We'll focus on two-way classification, and look at the cases where the Naive Bayesian incorrectly includes a trial in a category (false positives) as well as take a look at trials it narrowly excludes from the category (let's call them close relatives).

In the script for testing the learner (`test-nb-learner.py`), we saved the trial ids for false positives and close relatives so we could examine them later.

Here's the relevant code bit:

```
result = mynb.classify(trial)
guessedclass =  max(result, key=result.get)
if cattocheck:
    diff = abs(result[cattocheck] - result['other'])
    if diff < 10 and guessedclass != cattocheck:
        closetrials.append(trial[0])
        difflist.append(diff)
    if correctclass == cattocheck:
        catinsample += 1
    if guessedclass == cattocheck:
         guesses += 1
         if guessedclass == correctclass:
             hits += 1
         else:
             falsepositives.append(trial[0])
if guessedclass == correctclass:
    correctguesses += 1
```

False positives are easy to catch: we simply save the cases where we guessed that a trial belonged to the category but it really did not.

For close relatives, we first check how confident we were that the trial did not belong in our category. When we issue the call to classify the trial *mynb.classify(trial)*, it returns us a dictionary that looks like this:

```
{
    'other': -2358.522248351527,
    'violenttheft-robbery': -2326.2878233211086
}
```

So to find the close relatives, we compare these two values, and if the difference between them is small, we save the id of the trial we are currently classifying into a list of close relatives. (In the code chunk above, we have rather arbitrarily defined a "small" difference as being under 10).

At the end of the script, we write the results of these operations into two text files: `falsepositives.txt` and `closerelatives.txt`.

Let's look more closely at misclassifications for the category "violenttheft-robbery." Here are the first 10 rows of the close relatives file and the first 20 rows of the false positives file, sorted by offense:

Close relatives

```
breakingpeace-wounding, t18350105-458, 1.899530878
theft-pocketpicking, t18310407-90, 0.282424548
theft-pocketpicking, t18380514-1168, 0.784184742
theft-pocketpicking, t18301028-208, 0.797341405
theft-pocketpicking, t18341016-85, 1.296811989
violenttheft-robbery, t18370102-317, 1.075548985
violenttheft-robbery, t18350921-2011, 1.105672712
violenttheft-robbery, t18310407-204, 1.521788666
violenttheft-robbery, t18370102-425, 1.840718222
violenttheft-robbery, t18330214-13, 2.150018805
```

False positives

```
breakingpeace-assault, t18391021-2933
breakingpeace-wounding, t18350615-1577
breakingpeace-wounding, t18331017-159
breakingpeace-wounding, t18350615-1578
breakingpeace-wounding, t18330704-5
kill-manslaughter, t18350706-1682
kill-manslaughter, t18360919-2161
kill-manslaughter, t18380618-1461
kill-murder, t18330103-7
kill-murder, t18391021-2937
miscellaneous-pervertingjustice, t18340904-144
theft-pocketpicking, t18300114-128
theft-pocketpicking, t18310407-66
theft-pocketpicking, t18330905-92
theft-pocketpicking, t18370703-1639
theft-pocketpicking, t18301028-127
theft-pocketpicking, t18310106-87
theft-pocketpicking, t18331017-109
theft-pocketpicking, t18320216-108
theft-pocketpicking, t18331128-116
```

The first thing we notice is that many of the close relatives are in fact from our target category — they are cases that our classifier has narrowly missed. So saving these separately could compensate nicely for an otherwise low recall number.

The second thing we notice is that more of the false positives seem to have to do with violence, whereas more of the close relatives seem to have to do with stealing; it seems our classifier has pegged the violence aspect of robberies as more significant in distinguishing them than the filching aspect.

The third thing we notice is that theft-pocketpicking is a very common category among both the close relatives and the false positives. And indeed, if we look at a sample trial from violenttheft-robbery and another from among the close pocketpicking relatives, we notice that there are definitely close similarities.

For example, trial t18310407-90, the closest close relative, involved a certain Eliza Williams indicted for pocketpicking. Williams was accused of stealing a watch and some other items from a certain Thomas Turk; according to Turk and his friend, they had been pub-crawling, Eliza

Williams (whom they did not know from before) had tagged along with them, and at one point in the evening had pocketed Turk's watch (Turk, by this time, was quite tipsy). Williams was found guilty and sentenced to be confined for one year.

Meanwhile, in trial t18300708-14, correctly classed as violenttheft-robbery, a man called Edward Overton was accused of feloniously assaulting a fellow by the name of John Quinlan. Quinlan explained that he had been out with friends, and when he parted from them he realized it was too late to get into the hotel where he worked as a waiter and (apparently) also had lodgings. Having nowhere to go, he decided to visit a few more public-houses. Along the way, he met Overton, whom he did not know from before, and treated him to a few drinks. But then, according to Quinlan, Overton attacked him as they were walking from one pub to another, and stole his watch as well as other possessions of his. According to Overton, however, Quinlan had given him the watch as a guarantee that he would repay Overton if Overton paid for his lodging for the night. Both men, it seems, were thoroughly drunk by the end of the evening. Overton was found not guilty.

Both trials, then, are stories of groups out drinking and losing their possessions; what made the latter a trial for robbery rather than for pocketpicking was simply Quinlan's accusation that Overton had "struck him down." For a historian interested in either robberies or pocketpickings (or pub-crawling in 1830s London), both would probably be equally interesting.

In fact, the misclassification patterns of the learner indicate that even when data is richly annotated, such as in the case of the Old Bailey, using a machine learner to extract documents may be useful for a historian: in this case, it would help you extract trials from different offense categories that share features of interest to you, regardless of the offense label.

Tuning

The possibilities for tuning are practically endless.

For example, you might consider tweaking your data. For instance, instead of giving your classifier all the words in the document, you might present it with a reduced set.

One way of reducing the number of words is to collapse different words together through stemming. So the verb forms "killed," "kills," "killing" would all become "kill" (as would the plural noun "kills"). A popular stemmer is the Snowball Stemmer,[421] and you could add that to the tokenization step. (I ran a couple of tests with this, and while it made the process much slower, it didn't much improve the results. But that would probably depend a bit on the kind of data you have.)

[421] 'Snoball': http://snowball.tartarus.org/

Another way is to select the words you give to the classifier according to some principle. One common solution is to pick only the words with a high **TF-IDF** score. TF-IDF is short for "term frequency - inverse document frequency," and a high score means that the term occurs quite frequently in the document under consideration but rarely in documents in general. (You can also check out a more detailed explanation of TF-IDF,[422] along with some Python code for calculating it.)

Other options include simply playing with the size of the priors: now, the Naive Bayesian has a class prior as well as a feature prior of 0.5, meaning that it pretends to have seen all classes and all words at least one-half times. Doing test runs with different priors might get you different results.

In addition to simply changing the general prior sizes, you might consider having the classifier set a higher prior on the target category than on the "other" category, in effect requiring less evidence to include a trial in the target category. It might be worth a try particularly since we noted above when examining the close relatives (under Meanings of Misclassification) that many of them were in fact members of our target category. Setting a larger prior on the target class would probably catch those cases, boosting the recall. At the same time, it probably would also lower the precision. (To change the priors, you need to edit the `naivebayes.py` script.)

As you can see, there is quite a lot of fuzziness here: how you pick the features, how you pick the priors, and how you weight various priors all affect the results you get, and how to pick and weight is not governed by hard logic but is rather a process of trial and error. Still, like we noted noted in the section on the meaning of classification error above, if your goal is to get some interesting data to do historical analysis on, some fuzziness may not be such a big problem.

About the Author

Vilja Hulden is a history instructor and research associate in the Departments of History and Linguistics and the University of Colorado Boulder.

[422] Steven Loria 'Tutorial: Finding Important Words in Text Using TF-IDF' (1 September 2013): http://stevenloria.com/finding-important-words-in-a-document-using-tf-idf/

Part Five: Presenting History

The digital environment offers many opportunities for displaying and presenting historical materials in a number of ways. The lessons in this part explore some of those opportunities by sharing modes of putting history online. From basic web pages, to web exhibits, to digital maps. This is about putting the products of your efforts out there.

36. Understanding Web Pages and HTML

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Python Introduction and Installation'.[423]

"Hello World" in HTML

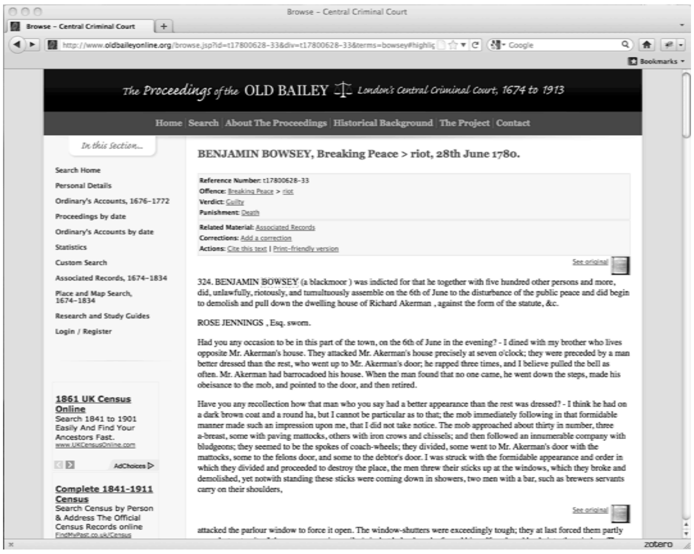
Viewing HTML files

When you are working with online sources, much of the time you will be using files that have been marked up with HTML (Hyper Text Markup Language). Your browser already knows how to interpret HTML, which is handy for human readers. Most browsers also let you see the HTML *source code* for any page that you visit. The two images below show a typical web page (from the *Old Bailey Online*)[424] and the HTML source used to generate that page, which you can see with the `Tools -> Web Developer -> Page Source` command in Firefox.

When you're working in the browser, you typically don't want or need to see the source for a web page. If you are writing a page of your own, however, it can be very useful to see how other people accomplished a particular effect. You will also want to study HTML source as you write programs to manipulate web pages or automatically extract information from them.

[423] William J. Turkel and Adam Crymble, 'Python Introduction and Installation', *The Programming Historian* (2012).

[424] Tim Hitchcock, Robert Shoemaker, Clive Emsley, Sharon Howard and Jamie McLaughlin, *et al.*, *The Old Bailey Proceedings Online, 1674-1913* (www.oldbaileyonline.org, version 7.0, 24 March 2012).


Browse – Central Criminal Court
http://www.oldbaileyonline.org/browse.jsp?id=t17800628-33&div=t17800628-33&terms=bowsey#highlig
Google
Bookmarks
The Proceedings of the OLD BAILEY London's Central Criminal Court, 1674 to 1913
Home | Search | About The Proceedings | Historical Background | The Project | Contact
In this Section...
Search Home
Personal Details
Ordinary's Accounts, 1676–1772
Proceedings by date
Ordinary's Accounts by date
Statistics
Custom Search
Associated Records, 1674–1834
Place and Map Search, 1674–1834
Research and Study Guides
Login / Register
1861 UK Census Online
Search 1841 to 1901 Easily And Find Your Ancestors Fast.
www.UKCensusOnline.com
AdChoices
Complete 1841-1911 Census
Search Census by Person & Address The Official Census Records online
FindMyPast.co.uk/Census
See original
zotero

BENJAMIN BOWSEY, Breaking Peace > riot, 28th June 1780.

Reference Number: t17800628-33
Offence: Breaking Peace > riot
Verdict: Guilty
Punishment: Death

Related Material: Associated Records
Corrections: Add a correction
Actions: Cite this text | Print-friendly version

324. BENJAMIN BOWSEY (a blackmoor) was indicted for that he together with five hundred other persons and more, did, unlawfully, riotously, and tumultuously assemble on the 6th of June to the disturbance of the public peace and did begin to demolish and pull down the dwelling house of Richard Akerman , against the form of the statute, &c.

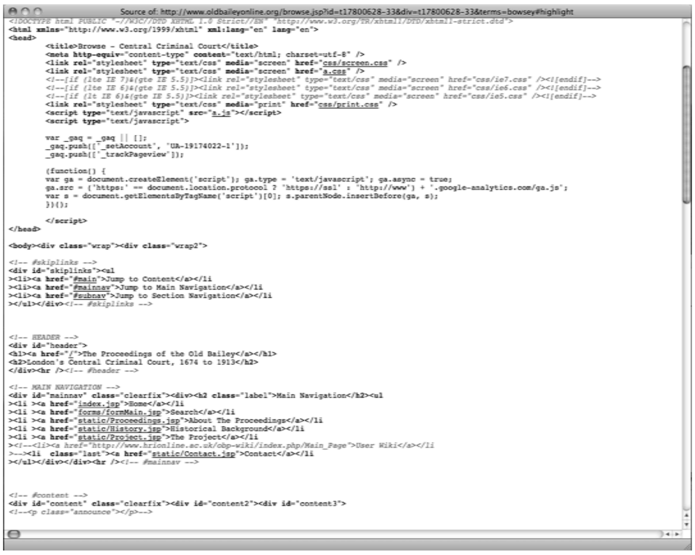
ROSE JENNINGS , Esq. sworn.

Had you any occasion to be in this part of the town, on the 6th of June in the evening? - I dined with my brother who lives opposite Mr. Akerman's house. They attacked Mr. Akerman's house precisely at seven o'clock; they were preceded by a man better dressed than the rest, who went up to Mr. Akerman's door; he rapped three times, and I believe pulled the bell as often. Mr. Akerman had barrocadoed his house. When the man found that no one came, he went down the steps, made his obeisance to the mob, and pointed to the door, and then retired.

Have you any recollection how that man who you say had a better appearance than the rest was dressed? - I think he had on a dark brown coat and a round ha, but I cannot be particular as to that; the mob immediately following in that formidable manner made such an impression upon me, that I did not take notice. The mob approached about thirty in number, three a-breast, some with paving mattocks, others with iron crows and chissels; and then followed an innumerable company with bludgeons; they seemed to be the spokes of coach-wheels; they divided, some went to Mr. Akerman's door with the mattocks, some to the felons door, and some to the debtor's door. I was struck with the formidable appearance and order in which they divided and proceeded to destroy the place, the men threw their sticks up at the windows, which they broke and demolished, yet notwith standing these sticks were coming down in showers, two men with a bar, such as brewers servants carry on their shoulders,

attacked the parlour window to force it open. The window-shutters were exceedingly tough; they at last forced them partly

Old Bailey Online Screenshot


Source of: http://www.oldbaileyonline.org/browse.jsp?id=t17800628-33&div=t17800628-33&terms=bowsey#highlight

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
        <title>Browse - Central Criminal Court</title>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <link rel="stylesheet" type="text/css" media="screen" href="css/screen.css" />
        <link rel="stylesheet" type="text/css" media="screen" href="a.css" />
        <!--[if (lte IE 7)&(gte IE 5.5)]><link rel="stylesheet" type="text/css" media="screen" href="css/ie7.css" /><![endif]-->
        <!--[if (lte IE 6)&(gte IE 5.5)]><link rel="stylesheet" type="text/css" media="screen" href="css/ie6.css" /><![endif]-->
        <!--[if (lt IE 6)&(gte IE 5.5)]><link rel="stylesheet" type="text/css" media="screen" href="css/ie5.css" /><![endif]-->
        <link rel="stylesheet" type="text/css" media="print" href="css/print.css" />
        <script type="text/javascript" src="a.js"></script>
        <script type="text/javascript">

        var _gaq = _gaq || [];
        _gaq.push(['_setAccount', 'UA-19174022-1']);
        _gaq.push(['_trackPageview']);

        (function() {
        var ga = document.createElement('script'); ga.type = 'text/javascript'; ga.async = true;
        ga.src = ('https:' == document.location.protocol ? 'https://ssl' : 'http://www') + '.google-analytics.com/ga.js';
        var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(ga, s);
        })();

        </script>
</head>

<body><div class="wrap"><div class="wrap2">

<!-- #skiplinks -->
<div id="skiplinks"><ul
><li><a href="#main">Jump to Content</a></li
><li><a href="#mainnav">Jump to Main Navigation</a></li
><li><a href="#subnav">Jump to Section Navigation</a></li
></ul></div><!-- #skiplinks -->



<!-- HEADER -->
<div id="header">
<h1><a href="/">The Proceedings of the Old Bailey</a></h1>
<h2>London's Central Criminal Court, 1674 to 1913</h2>
</div><hr /><!-- #header -->

<!-- MAIN NAVIGATION -->
<div id="mainnav" class="clearfix"><div><h2 class="label">Main Navigation</h2><ul
><li ><a href="index.jsp">Home</a></li
><li ><a href="forms/formMain.jsp">Search</a></li
><li ><a href="static/Proceedings.jsp">About The Proceedings</a></li
><li ><a href="static/History.jsp">Historical Background</a></li
><li ><a href="static/Project.jsp">The Project</a></li
><!--<li><a href="http://www.hrionline.ac.uk/obp-wiki/index.php/Main_Page">User Wiki</a></li
>--><li  class="last"><a href="static/Contact.jsp">Contact</a></li
></ul></div></div><hr /><!-- #mainnav -->



<!-- #content -->
<div id="content" class="clearfix"><div id="content2"><div id="content3">
<!--<p class="announce"></p>-->
```

HTML source for Old Bailey Online web page

(To learn more about HTML, you may find it useful at this point to work through the W3 Schools HTML tutorial.[425] Detailed knowledge of HTML isn't immediately necessary to continue reading, but any time that you spend learning HTML will be amply rewarded in your work as a digital historian or digital humanist.)

"Hello World" in HTML

HTML is what is known as a *markup* language. In other words, HTML is text that has been "marked up" with *tags* that provide information for the interpreter (which is often a web browser). Suppose you are formatting a bibliographic entry and you want to indicate the title of a work by italicizing it. In HTML you use `em` tags ("em" stands for emphasis). So part of your HTML file might look like this

[425] 'HTML (5) Tutorial', *W3 Schools*: http://www.w3schools.com/html/default.asp

```
... in Cohen and Rosenzweig's <em>Digital History</em>, for example ...
```

The simplest HTML file consists of tags which indicate the beginning and end of the whole document, and tags which identify a `head` and a `body` within that document. Information about the file usually goes into the head, whereas information that will be displayed on the screen usually goes into the body.

```
<html>
<head></head>
<body>Hello World!</body>
</html>
```

You can try creating some HTML code. Go to your text editor, and create a new file. Copy the code below into the editor. The first line tells the browser what kind of file it is. The `html` tag has the text direction set to `ltr` (left to right) and the `lang` (language) set to US English. The `title` tag in the head of the HTML document contains material that is usually displayed in the top bar of a window when the page is being viewed, and in Firefox tabs.

```
<!doctype html>
<html dir="ltr" lang="en-US">

<head>
    <title><!-- Insert your title here --></title>
</head>

<body>
    <!-- Insert your content here -->
</body>
</html>
```

Change both

```
<!-- Insert your title here -->
```

and

```
<!-- Insert your content here -->
```

to

```
Hello World!
```

Save the file to your `programming-historian` directory as `hello-world.html`. Now go to Firefox and choose `File -> New Tab` and then `File -> Open File`. Choose `hello-world.html`. Depending on your text editor you may have a 'view page in browser' or 'open in browser' option. Once you have opened the file, your message should appear in the browser. Note the difference between opening an HTML file with a browser like Firefox (which interprets it) and opening the same file with your text editor (which does not).

Suggested readings for learning HTML

W3 Schools HTML Tutorial[426]

If you are following along the ‘Intro to Python’ lessons in order, the next lesson in this sequence is ‘Working with Text Files in Python’.[427]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[426] ‘HTML (5) Tutorial’, *W3 Schools*: http://www.w3schools.com/html/default.asp

[427] William J. Turkel and Adam Crymble, ‘Working with Text Files in Python’, *The Programming Historian*, 2012.

37. Output Data as an HTML File with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Creating and Viewing HTML Files with Python'.[428]

Lesson Goals

This lesson takes the frequency pairs created in 'Counting Word Frequencies with Python'[429] and outputs them to an HTML file.

Here you will learn how to output data as an HTML file using Python. You will also learn about string formatting. The final result is an HTML file that shows the keywords found in the original source in order of descending frequency, along with the number of times that each keyword appears.

Files Needed For This Lesson

`obo.py`

If you do not have these files from the previous lesson, you can download programming-historian-3, a zip file from the previous lesson: http://programminghistorian.org/assets/programming-historian3.zip

Building an HTML wrapper

In the previous lesson, you learned how to embed the message "Hello World!" in HTML tags, write the result to a file and open it automatically in the browser. A program that puts formatting codes around something so that it can be used by another program is sometimes called a *wrapper*. What we're going to do now is develop an HTML wrapper for the output of our code that computes word frequencies. We're also going to add some helpful, dynamic *metadata* to supplement the frequency data collected in 'Counting Word Frequencies with Python'.

Metadata

The distinction between data and metadata is crucial to information science. Metadata are data about data. This concept should already be very familiar to you, even if you haven't heard the term before. Consider a traditional book. If we take the text of the book to be the data, there are a

[428] William J. Turkel and Adam Crymble, 'Creating and Viewing HTML Files with Python', *The Programming Historian* (2012).

[429] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

number of other characteristics which are associated with that text, but which may or may not be explicitly printed in the book. The title of the work, the author, the publisher, and the place and date of publication are metadata that are typically printed in the work. The place and date of writing, the name of the copy editor, Library of Congress cataloging data, and the name of the font used to typeset the book are sometimes printed in it. The person who purchased a particular copy may or may not write their name in the book. If the book belongs in the collection of a library, that library will keep additional metadata, only some of which will be physically attached to the book. The record of borrowing, for example, is usually kept in some kind of database and linked to the book by a unique identifier. Libraries, archives and museums all have elaborate systems to generate and keep track of metadata.

When you're working with digital data, it is a good idea to incorporate metadata into your own files whenever possible. We will now develop a few basic strategies for making our data files *self-documenting*. In our wrapper, we want to include dynamic information about the file, such as the time and date it was created, as well as an HTML title that is relevant to the file. In this case we could just give it a name ourselves, but when we start working with multiple files, automatically creating self-documenting files will save a lot of time, so we’ll practice now. And for that, we'll have to learn to take advantage of a few more powerful string formatting options.

Python string formatting

Python includes a special formatting operator that allows you to insert one string into another one. It is represented by a percent sign followed by an "s". Open a Python shell and try the following examples.

```
frame = 'This fruit is a %s'
print frame
-> This fruit is a %s

print frame % 'banana'
-> This fruit is a banana

print frame % 'pear'
-> This fruit is a pear
```

There is also a form which allows you to interpolate a list of strings into another one.

```
frame2 = 'These are %s, those are %s'
print frame2
-> These are %s, those are %s

print frame2 % ('bananas', 'pears')
-> These are bananas, those are pears
```

In these examples, a `%s` in one string indicates that another string is going to be embedded at that point. There are a range of other string formatting codes, most of which allow you to embed numbers in strings in various formats, like `%i` for integer (eg. 1, 2, 3), `%f` for floating-point decimal (eg. 3.023, 4.59, 1.0), and so on. Using this method we can input information that is unique to the file.

Self-documenting data file

Let's bundle some of the code that we've already written into functions. One of these will take a URL and return a string of lowercase text from the web page. Copy this code into the `obo.py` module.

```
# Given a URL, return string of lowercase text from page.

def webPageToText(url):
    import urllib2
    response = urllib2.urlopen(url)
    html = response.read()
    text = stripTags(html).lower()
    return text
```

We're also going to want a function that takes a string of any sort and makes it the body of an HTML file which is opened automatically in Firefox. This function should include some basic metadata, like the time and date that it was created and the name of the program that created it. Study the following code carefully, then copy it into the `obo.py` module.

Mac Instructions

If you are using a Mac, make sure you include the proper file path in the filename variable on the 2nd last line to reflect where you're saving your files.

```
# Given name of calling program, a url and a string to wrap,
# output string in html body with basic metadata and open in Firefox tab.
def wrapStringInHTML(program, url, body):
    import datetime
    from webbrowser import open_new_tab

    now = datetime.datetime.today().strftime("%Y%m%d-%H%M%S")
    filename = program + '.html'
    f = open(filename,'w')
    wrapper = """<html>
    <head>
    <title>%s output - %s</title>
    </head>
    <body><p>URL: <a href=\"%s\">%s</a></p><p>%s</p></body>
    </html>"""

    whole = wrapper % (program, now, url, url, body)
    f.write(whole)
    f.close()
    #Change the filepath variable below to match the location of your directory
    filename = 'file:///Users/username/Desktop/programming-historian/' + filename

    open_new_tab(filename)
```

Windows Instructions

```
# Given name of calling program, a url and a string to wrap,
# output string in html body with basic metadata
# and open in Firefox tab.

def wrapStringInHTML(program, url, body):
    import datetime
    from webbrowser import open_new_tab

    now = datetime.datetime.today().strftime("%Y%m%d-%H%M%S")

    filename = program + '.html'
    f = open(filename,'w')

    wrapper = """<html>
    <head>
    <title>%s output - %s</title>
    </head>
    <body><p>URL: <a href=\"%s\">%s</a></p><p>%s</p></body>
    </html>"""

    whole = wrapper % (program, now, url, url, body)
    f.write(whole)
    f.close()

    open_new_tab(filename)
```

Note that this function makes use of the string formatting operator about which we just learned. If you are still having trouble with this idea, take a look at the HTML file that opened in your new Firefox tab and you should see how this worked. If you're still stuck, take a look at the

```
URL: http://www.oldbaileyonline.org/print.jsp?div=t17800628-33
```

in the HTML file and trace back how the program knew to put the URL value there.

The function also calls the Python `datetime` library to determine the current time and date. Like the string formatting operator `%s`, this library uses the `%` as replacements for values. In this case, the `%Y %m %d %H %M %S` represents year, month, date, hour, minute and second respectively. Unlike the `%s`, the program will determine the value of these variables for you using your computer's clock. It is important to recognize this difference.

This date metadata, along with the name of the program that called the function, is stored in the HTML title tag. The HTML file that is created has the same name as the Python program that creates it, but with a `.html` extension rather than a `.py` one.

Putting it all together

Now we can create another version of our program to compute frequencies. Instead of sending its output to a text file or an output window, it sends the output to an HTML file which is opened in a new Firefox tab. From there, the program's output can be added easily as bibliographic entries to Zotero.

Type or copy the following code into your text editor, save it as `html-to-freq-3.py` and execute it, to confirm that it works as expected.

```
# html-to-freq-3.py
import obo

# create sorted dictionary of word-frequency pairs
url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'
text = obo.webPageToText(url)
fullwordlist = obo.stripNonAlphaNum(text)
wordlist = obo.removeStopwords(fullwordlist, obo.stopwords)
dictionary = obo.wordListToFreqDict(wordlist)
sorteddict = obo.sortFreqDict(dictionary)

# compile dictionary into string and wrap with HTML
outstring = ""
for s in sorteddict:
    outstring += str(s)
    outstring += "<br />"
obo.wrapStringInHTML("html-to-freq-3", url, outstring)
```

Note that we interspersed our word-frequency pairs with the HTML break tag `<br\>`, which acts as a *newline*. If all went well, you should see the same word frequencies that you computed in the last section, this time in your browser window.

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each chapter you can download the "programming-historian" zip file to make sure you have the correct code. If you are following along with the Mac / Linux version you may have to open the `obo.py` file and change "file:///Users/username/Desktop/programming-historian/" to the path to the directory on your own computer.

Mac/Linux: http://programminghistorian.org/assets/programming-historian-mac-linux.zip

Windows: http://programminghistorian.org/assets/programming-historian-windows.zip

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Keywords in Context (Using n-grams) with Python'.[430]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[430] William J. Turkel and Adam Crymble, 'Keywords in Context (Using n-grams) with Python)', *The Programming Historian*, 2012.

38. Creating and Viewing HTML Files with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Counting Word Frequencies with Python'.[431]

Lesson Goals

This lesson uses Python to create and view an HTML file. If you write programs that output HTML, you can use any browser to look at your results. This is especially convenient if your program is automatically creating hyperlinks or graphic entities like charts and diagrams.

Here you will learn how to create HTML files with Python scripts, and how to use Python to automatically open an HTML file in Firefox.

Files Needed For This Lesson

`obo.py`

If you do not have these files from the previous lesson, you can download programming-historian-3, a zip file from the previous lesson: http://programminghistorian.org/assets/programming-historian3.zip

Creating HTML with Python

At this point, we've started to learn how to use Python to download online sources and extract information from them automatically. Remember that our ultimate goal is to incorporate programming seamlessly into our research practice. In keeping with this goal, in this lesson and the next, we will learn how to output data back as HTML. This has a few advantages. First, by storing the information on our hard drive as an HTML file we can open it with Firefox and use Zotero[432] to index and annotate it later. Second, there are a wide range of visualization options for HTML which we can draw on later.

If you have not done the W3 Schools HTML tutorial yet,[433] take a few minutes to do it before continuing. We're going to be creating an HTML document using Python, so you will have to know what an HTML document is!

[431] William J. Turkel and Adam Crymble, 'Counting Word Frequencies with Python', *The Programming Historian* (2012).

[432] 'Zotero': http://zotero.org/

[433] 'HTML (5) Tutorial', *W3 Schools*: http://www.w3schools.com/html/default.asp

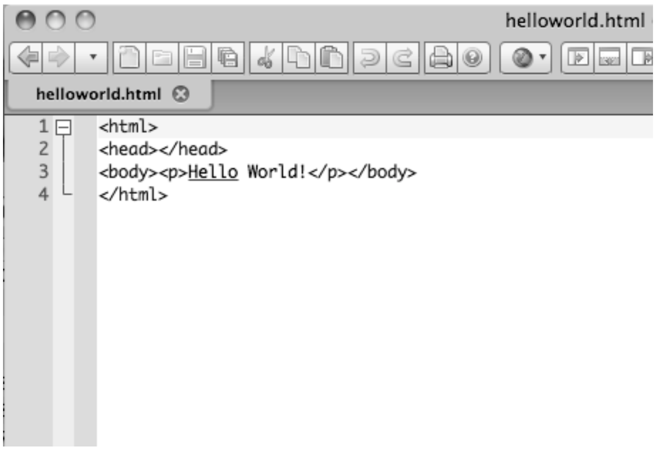
"Hello World" in HTML using Python

One of the more powerful ideas in computer science is that a file that seems to contain code from one perspective can be seen as data from another. It is possible, in other words, to write programs that manipulate other programs. What we're going to do next is create an HTML file that says "Hello World!" using Python. We will do this by storing HTML *tags* in a multiline Python *string* and saving the contents to a new file. This file will be saved with an `.html` extension rather than a `.txt` extension.

Typically an HTML file begins with a doctype declaration.[434] You saw this when you wrote an HTML "Hello World" program in an earlier lesson. To make reading our code easier, we will omit the doctype in this example. Recall a multi-line string is created by enclosing the text in three quotation marks (see below).

```
# write-html.py

f = open('helloworld.html','w')

message = """<html>
<head></head>
<body><p>Hello World!</p></body>
</html>"""

f.write(message)
f.close()
```

Save the above program as `write-html.py` and execute it. Use *File -> Open* in your chosen text editor to open `helloworld.html` to verify that your program actually created the file. The content should look like this:



```
<html>
<head></head>
<body><p>Hello World!</p></body>
</html>
```

HTML Source Generated by Python Program

Now go to your Firefox browser and choose *File -> New Tab*, go to the tab, and choose *File -> Open File*. Select `helloworld.html`. You should now be able to see your message in the browser. Take a moment to think about

[434] 'HTML <!DOCTYPE> Declaration': http://www.w3schools.com/tags/tag_doctype.asp

this: you now have the ability to write a program which can automatically create a webpage. There is no reason why you could not write a program to automatically create a whole website if you wanted to.

Using Python to Control Firefox

We automatically created an HTML file, but then we had to leave our editor and go to Firefox to open the file in a new tab. Wouldn't it be cool to have our Python program include that final step? Type or copy the code below and save it as `write-html-2.py`. When you execute it, it should create your HTML file and then automatically open it in a new tab in Firefox. Sweet!

Mac Instructions

Mac users will have to specify to the precise location of the `.html` file on their computer. To do this, locate the `programming-historian` folder you created to do these tutorials, right-click it and select "Get Info".

You can then cut and paste the file location listed after "Where:" and make sure you include a trailing slash (/) to let the computer know you want something inside the directory (rather than the directory itself).

```
# write-html-2.py
import webbrowser

f = open('helloworld.html','w')

message = """<html>
<head></head>
<body><p>Hello World!</p></body>
</html>"""

f.write(message)
f.close()

#Change path to reflect file location
filename = 'file:///Users/username/Desktop/programming-historian/'+'helloworld.html'
webbrowser.open_new_tab(filename)
```

If you're getting a "File not found" error you haven't changed the filename path correctly.

Windows Instructions

```
# write-html-2.py

import webbrowser

f = open('helloworld.html','w')

message = """<html>
<head></head>
<body><p>Hello World!</p></body>
</html>"""

f.write(message)
f.close()

webbrowser.open_new_tab('helloworld.html')
```

Not only have you written a Python program that can write simple HTML, but you've now controlled your Firefox browser using Python. In the next lesson, we turn to outputting the data that we have collected as an HTML file.

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each chapter you can download the "programming-historian" zip file to make sure you have the correct code. If you are following along with the Mac / Linux version you may have to open the `obo.py` file and change "file:///Users/username/Desktop/programming-historian/" to the path to the directory on your own computer.

programming-historian [Mac / Linux] (zip)
http://programminghistorian.org/assets/programming-historian-mac-linux.zip
programming-historian [Windows] (zip)
http://programminghistorian.org/assets/programming-historian-windows.zip

If you are following along the ‘Intro to Python’ lessons in order, the next lesson in this sequence is ‘Output Data as an HTML File with Python’.[435]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[435] William J. Turkel and Adam Crymble, ‘Output Data as an HTML File with Python’, *The Programming Historian*, 2012.

39. Output Keywords in Context in an HTML File with Python

William J. Turkel and Adam Crymble – 2012

Editor's Note: This lesson was originally written as part of a series of 'Intro to Python' lessons. You may find it easier to complete if you have already completed the previous lesson in this series: 'Keywords in Context (Using n-grams) with Python'.[436]

Lesson Goals

This lesson builds on 'Keywords in Context (Using N-grams)', where n-grams were extracted from a text. Here, you will learn how to output all of the n-grams of a given keyword in a document downloaded from the Internet, and display them clearly in your browser window.

Files Needed For This Lesson

`obo.py`

If you do not have these files from the previous lesson, you can download a zip file from the previous lesson: http://programminghistorian.org/assets/programming-historian3.zip

Making an N-Gram Dictionary

Our n-grams have an odd number of words in them for a reason. At this point, our n-grams don"t actually have a keyword; they're just a list of words. However, if we have an odd numbered n-gram the middle word will always have an equal number of words to the left and to the right. We can then use that middle word as our keyword. For instance, ["it", "was", "the", "best", "of", "times", "it"] is a 7-gram of the keyword "best".

Since we have a long text, we want to be able to output all n-grams for our keyword. To do this we will put each n-gram into a *dictionary*, using the middle word as the *key*. To figure out the keyword for each n-gram we can use the *index positions* of the list. If we are working with 5-grams, for example, the left context will consist of terms indexed by 0, 1, the keyword will be indexed by 2, and the right context terms indexed by 3, 4. Since Python indexes start at 0, a 5-gram's keyword will always be at index position 2.

[436] William J. Turkel and Adam Crymble, 'Keywords in Context (Using n-grams) with Python', *The Programming Historian* (2012).

That's fine for 5-grams, but to make the code a bit more robust, we want to make sure it will work for any length n-gram, assuming its length is an odd number. To do this we'll take the length of the n-gram, divide it by 2 and drop the remainder. We can achieve this using Python's `floor division` operator, represented by two slashes, which divides and then returns an answer to the nearest whole number, always rounding down – hence the term "floor".

```
print (7 // 2)
print (5 // 2)
print (3 // 2)
```

Let's build a function that can identify the index position of the keyword when given an n-gram with an odd number of words. Save the following to `obo.py`.

```
# Given a list of n-grams identify the index of the keyword.

def nGramsToKWICDict(ngrams):
    keyindex = len(ngrams[0]) // 2

    return keyindex
```

To determine the index of the keyword, we have used the `len` property to tell us how many items are in the first n-gram, then used floor division to isolate the middle index position. You can see if this worked by creating a new program, `get-keyword.py` and running it. If all goes well, since we are dealing with a 5-gram, you should get 2 as the index position of the keyword as we determined above.

```
import obo

test = 'this test sentence has eight words in it'
ngrams = obo.getNGrams(test.split(), 5)

print obo.nGramsToKWICDict(ngrams)
```

Now that we know the location of the keywords, let's add everything to a dictionary that can be used to output all KWIC n-grams of a particular keyword. Study this code and then replace your `nGramsToKWICDict` with the following in your `obo.py` module.

```
# Given a list of n-grams, return a dictionary of KWICs,
# indexed by keyword.

def nGramsToKWICDict(ngrams):
    keyindex = len(ngrams[0]) // 2

    kwicdict = {}

    for k in ngrams:
        if k[keyindex] not in kwicdict:
            kwicdict[k[keyindex]] = [k]
        else:
            kwicdict[k[keyindex]].append(k)
    return kwicdict
```

A `for` loop and `if` statement checks each n-gram to see if its keyword is already stored in the dictionary. If it isn't, it's added as a new entry. If it is, it's appended to the previous entry. We now have a dictionary named *kwicdict* that contains all the n-grams, sortable by keyword and we can turn to the task of outputting the information in a more useful format as we did in 'Output Data as HTML File'.[437]

Try rerunning the `get-keyword.py` program and you should now see what's in your KWIC dictionary.

Outputting to HTML

Pretty Printing a KWIC

"Pretty printing" is the process of formatting output so that it can be easily read by human beings. In the case of our keywords in context, we want to have the keywords lined up in a column, with the terms in the left-hand context right justified, and the terms in the right-hand context left justified. In other words, we want our KWIC display to look something like this:

```
                amongst them a black there was one
                 first saw the black i turned to
              had observed the black in the mob
                  say who that black was no seeing
                       i saw a black at first but
                  swear to any black yes there is
                    swear to a black than to a
                                 ...
```

This technique is not the best way to format text from a web designer's perspective. If you have some experience with HTML we encourage you to use another method that will create a standards compliant HTML file, but for new learners, we just can't resist the ease of the technique we're about to describe. After all, the point is to integrate programming principles quickly into your research.

To get this effect, we are going to need to do a number of list and string manipulations. Let's start by figuring out what our dictionary output will look like as it currently stands. Then we can work on refining it into what we want.

[437] William J. Turkel and Adam Crymble, 'Output Data as an HTML File with Python', *The Programming Historian* (2012).

```
# html-to-pretty-print.py
import obo

# create dictionary of n-grams
n = 7
url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

text = obo.webPageToText(url)
fullwordlist = obo.stripNonAlphaNum(text)
ngrams = obo.getNGrams(fullwordlist, n)
worddict = obo.nGramsToKWICDict(ngrams)

print worddict["black"]
```

As you can see when you run the above program, the output is not very readable yet. What we need to do is split the n-gram into three parts: before the keyword, the keyword, and after the keyword. We can then use the techniques learned in the previous chapters to wrap everything in HTML so that it is easy to read.

Using the same `slice` method as above, we will create our three parts. Open a Python shell and try the following examples. Pay close attention to what appears before and after the colon in each case. Knowing how to manipulate the slice method is a powerful skill for a new programming historian.

```
# calculate the length of the n-gram
kwic = 'amongst them a black there was one'.split()
n = len(kwic)
print n
-> 7

# calculate the index position of the keyword
keyindex = n // 2
print keyindex
-> 3

# display the items before the keyword
print kwic[:keyindex]
-> ['amongst', 'them', 'a']

# display the keyword only
print kwic[keyindex]
-> black

# display the items after the keyword
print kwic[(keyindex+1):]
-> ['there', 'was', 'one']
```

Now that we know how to find each of the three segments, we need to format each to one of three columns in our display.

The right-hand context is simply going to consist of a string of terms separated by blank spaces. We'll use the `join` method to turn the list entries into a string.

```
print ' '.join(kwic[(keyindex+1):])
-> there was one
```

We want the keywords to have a bit of *whitespace* padding around them. We can achieve this by using a string method called `center`, which will align the text to the middle of the screen. We can add padding by making the overall string be longer than the keyword itself. The expression below adds three blank spaces (6/2) to either side of the keyword. We've added hash marks at the beginning and end of the expression so you can see the leading and trailing blanks.

```
print '#' + str(kwic[keyindex]).center(len(kwic[keyindex])+6) + '#'
-> #   black   #
```

Finally, we want the left-hand context to be right justified. Depending on how large *n* is, we are going to need the overall length of this column to increase. We do this by defining a variable called *width* and then making the column length a multiple of this variable (we used a width of 10 characters, but you can make it larger or smaller as desired). The `rjust` method handles right justification. Once again, we've added hash marks so you can see the leading blanks.

```
width = 10
print '#' + ' '.join(kwic[:keyindex]).rjust(width*keyindex) + '#'
-> #                 amongst them a#
```

We can now combine these into a function that takes a KWIC and returns a pretty-printed string. Add this to the `obo.py` module. Study the code to make sure you understand it before moving on.

```
# Given a KWIC, return a string that is formatted for
# pretty printing.

def prettyPrintKWIC(kwic):
    n = len(kwic)
    keyindex = n // 2
    width = 10

    outstring = ' '.join(kwic[:keyindex]).rjust(width*keyindex)
    outstring += str(kwic[keyindex]).center(len(kwic[keyindex])+6)
    outstring += ' '.join(kwic[(keyindex+1):])

    return outstring
```

Putting it All Together

We can now create a program that, given a URL and a keyword, wraps a KWIC display in HTML and outputs it in Firefox. This program begins and ends in a similar fashion as the program that computed word frequencies. Type or copy the code into your text editor, save it as `html-to-kwic.py`, and execute it.

```
# html-to-kwic.py

import obo

# create dictionary of n-grams
n = 7
url = 'http://www.oldbaileyonline.org/print.jsp?div=t17800628-33'

text = obo.webPageToText(url)
fullwordlist = ('# ' * (n//2)).split()
fullwordlist += obo.stripNonAlphaNum(text)
fullwordlist += ('# ' * (n//2)).split()
ngrams = obo.getNGrams(fullwordlist, n)
worddict = obo.nGramsToKWICDict(ngrams)

# output KWIC and wrap with html
target = 'black'
outstr = '<pre>'
if worddict.has_key(target):
    for k in worddict[target]:
        outstr += obo.prettyPrintKWIC(k)
        outstr += '<br />'
else:
    outstr += 'Keyword not found in source'

outstr += '</pre>'
obo.wrapStringInHTML('html-to-kwic', url, outstr)
```

The first part is the same as above. In the second half of the program, we've wrapped everything in the HTML *pre* tag (pre-formatted), which tells the browser not to monkey with any of the spacing we've added.

Also, notice that we use the `has_key` dictionary method to make sure that the keyword actually occurs in our text. If it doesn't, we can print a message for the user before sending the output to Firefox. Try changing the target variable to a few other keywords. Try one you know isn't there to make sure your program doesn't output something when it shouldn't.

We have now created a program that looks for a keyword in a dictionary created from an HTML page on the web, and then outputs the n-grams of that keyword to a new HTML file for display on the web. All of the lessons up to this point have included parts of Python vocabulary and methods needed to create this final program. By referring to those lessons, you can now experiment with Python to create programs that accomplish specific tasks that will help in your research process.

Code Syncing

To follow along with future lessons it is important that you have the right files and programs in your "programming-historian" directory. At the end of each chapter you can download the "programming-historian" zip file to make sure you have the correct code. If you are following along with the Mac / Linux version you may have to open the `obo.py` file and change "file:///Users/username/Desktop/programming-historian/" to the path to the directory on your own computer.

programming-historian [Mac / Linux] (zip)

http://programminghistorian.org/assets/programming-historian-mac-linux.zip

programming-historian [Windows] (zip)

http://programminghistorian.org/assets/programming-historian-windows.zip

If you are following along the 'Intro to Python' lessons in order, the next lesson in this sequence is 'Downloading Multiple Records Using Query Strings'.[438]

About the Authors

William J. Turkel is a professor of history at Western University. Adam Crymble is a lecturer of digital history at the University of Hertfordshire.

[438] Adam Crymble, 'Downloading Multiple Records Using Query Strings', *The Programming Historian*, 2012.

40. Up and Running with Omeka.net

Miriam Posner – 2013

'Omeka'[439] is a free content management system[440] that makes it easy to create websites that show off collections of items. As you will learn below, there are actually two versions of Omeka: Omeka.net and Omeka.org. In this lesson you willl be using the former. If you would rather learn how to install Omeka yourself, read the Jonathan Reeve's lesson on, 'Installing Omeka'.[441]

Omeka is an ideal solution for historians who want to display collections of documents, archivists who want to organize artifacts into categories, and teachers who want students to learn about the choices involved in assembling historical collections. It is not difficult, but it is helpful to start off with some basic terms and concepts. In this lesson, you will sign up for an account at Omeka.net and start adding digital objects to your site.

When might Omeka.net be the right choice for your website?

You have a set of items you want to display on the web. Omeka is designed to display collections. The content of the collection can be anything from physical objects, to photographs, to people, or even ideas. To make the most of Omeka you should have lots of items that you want to show off.

You want to tell stories with those items. With Omeka, you can create exhibits: narrative walk-throughs of items.

You want to preserve complete information about each object. Omeka excels at metadata;[442] that is, information about the items in your collection. With Omeka you can fill out a form to describe the attributes of each item in your collection, helping you to keep track of this information in the future.

[439] 'Omeka': http://www.omeka.net/

[440] 'Content Management System', *Wikipedia*: https://en.wikipedia.org/wiki/Content_management_system

[441] Jonathan Reeve, 'Installing Omeka', *The Programming Historian* (2015).

[442] 'Metadata', *Wikipedia*: https://en.wikipedia.org/wiki/Metadata

When might Omeka.net not be the right choice for your website?

You want a simple website. If you just want a website with a few pages, some text, images, and other media, Omeka might be more tool than you need. Instead, consider WordPress[443] or some basic HTML.

You want a lot of control over the way things look. Omeka.net sites come with a number of built-in themes[444] which define how the website looks (the colours, fonts, layouts, etc), but you cannot control every element of your site's appearance. If you want to fine-tune the appearance of your site, consider using Omeka.org and customizing a theme. You will need some experience with CSS to do this effectively.[445]

You want sophisticated, dynamic queries of your database. A user can search your Omeka collection, but you cannot easily customize the home page so that it, say, always shows the most-viewed spoon in your spoon collection. That is, Omeka does not allow you to create custom queries. If this is important to you, consider Drupal.[446]

You want to create very complex paths through your collection. Omeka exhibits, which tell the story of your items, are pretty linear and straightforward. If you find this constraining, you might consider Scalar, which allows you to set up and visualize multiple paths through a database.[447]

An Omeka vocabulary lesson

Item: The basic unit of an Omeka site. An item can be anything: a photograph, a work of art, a person, an idea. You will describe each item, and you can upload files to represent it. You will build your Omeka site by assembling items.

Collection: A set of items that you have grouped together. Your Omeka site can have multiple collections, but an individual item can only belong to one collection at a time.

Exhibit: A thematic tour of your items. Each exhibit has sections and pages. You might think of these as akin to book chapters and book pages. A section is a group of pages, and a page is a group of items (along with descriptions). You can have multiple exhibits, and items can belong to multiple exhibits.

Dublin Core: Dublin Core is the name for a kind of metadata. Metadata is sort of what it sounds like; that is, information about information. You will use metadata to describe attributes of your items, like their sizes, dates of creation, etc. In order to keep these descriptions consistent,

[443] 'Wordpress': https://wordpress.com/

[444] 'Themes': http://omeka.org/add-ons/themes/

[445] 'CSS Tutorial', *W3 Schools*: http://www.w3schools.com/css/

[446] 'Drupal': https://www.drupal.org/

[447] 'Scalar': http://scalar.usc.edu/

information professionals have defined various metadata standards. Dublin Core[448] is the name of the standard that Omeka uses.

Item Type: An item, as we learned, can be many different things, like a photograph, a website, a book, or a person. An "item type" is just the kind of thing the item is. You can choose from a built-in list of item types, or you can create your own.

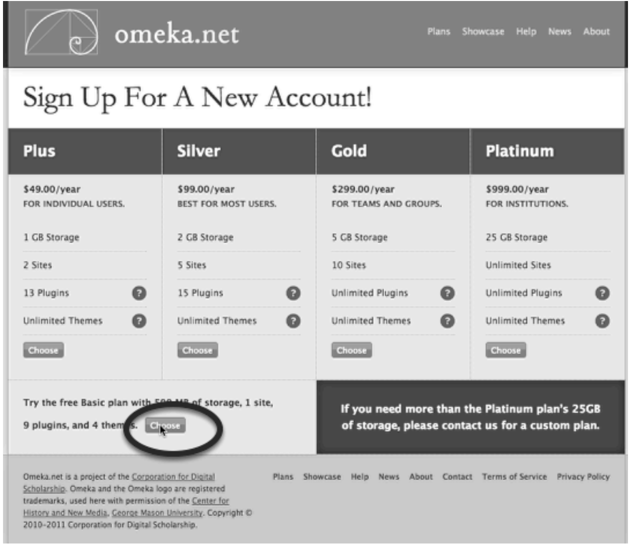
Simple Pages: A page on your Omeka site that is not part of an exhibit or item. For example, you can add an "About" page using Simple Pages.

Omeka.org versus Omeka.net: There are two kinds of Omeka sites. The kind you are using is hosted at Omeka.net, meaning that you do not have to install anything and you do not need to have a web server of your own. You just sign up for an account using a web form. If you would like to customize your Omeka site more heavily than Omeka.net allows, you might consider Omeka.org. With an Omeka.org site, you download a free software package and install it on your own server. This means that Omeka.org sites can be more customized, but you have to be comfortable installing Omeka on a server.

… And one more thing! You might think it's pronounced oh-mee-ka, but it's actually oh-meh-ka. Confusing, I know!

Now that we have got that out of the way, let's get started!

Sign up for an Omeka account



Sign Up For A New Account!

Plus	Silver	Gold	Platinum
$49.00/year FOR INDIVIDUAL USERS.	$99.00/year BEST FOR MOST USERS.	$299.00/year FOR TEAMS AND GROUPS.	$999.00/year FOR INSTITUTIONS.
1 GB Storage	2 GB Storage	5 GB Storage	25 GB Storage
2 Sites	5 Sites	10 Sites	Unlimited Sites
13 Plugins ?	15 Plugins ?	Unlimited Plugins ?	Unlimited Plugins ?
Unlimited Themes ?	Unlimited Themes ?	Unlimited Themes ?	Unlimited Themes ?
Choose	Choose	Choose	Choose

Try the free Basic plan with 500 MB of storage, 1 site, 9 plugins, and 4 themes. Choose

If you need more than the Platinum plan's 25GB of storage, please contact us for a custom plan.

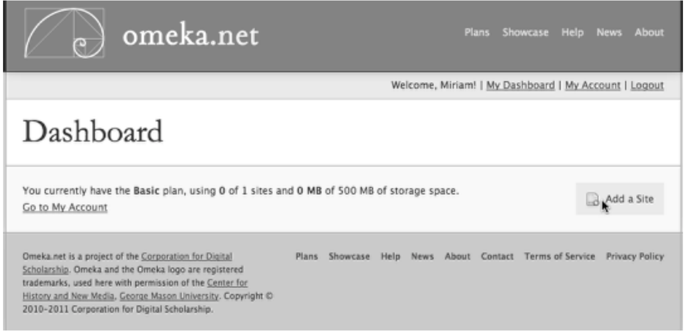
Omeka.net is a project of the Corporation for Digital Scholarship. Omeka and the Omeka logo are registered trademarks, used here with permission of the Center for History and New Media, George Mason University. Copyright © 2010-2011 Corporation for Digital Scholarship.

Sign up for a new account screen on Omeka.net

Go to www.omeka.net and click on **Sign Up**. Choose the Basic plan. Fill in the sign-up form. Check your email for the link to activate your account.

[448] 'Dublin Core': http://dublincore.org/documents/dcmi-terms/

Create your new Omeka site


omeka.net
Plans Showcase Help News About
Add a Site
Plans Showcase Help News About Contact Terms of Service Privacy Policy

Welcome, Miriam! | My Dashboard | My Account | Logout

Dashboard

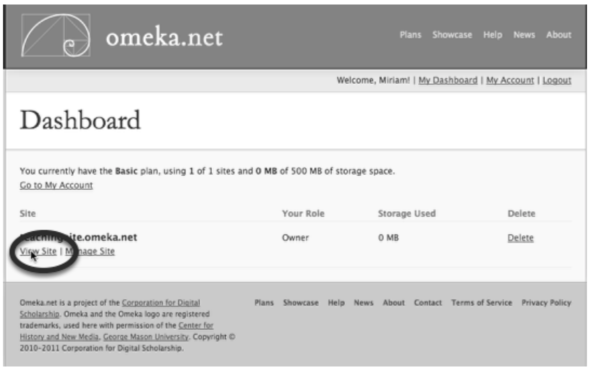
You currently have the **Basic** plan, using **0** of 1 sites and **0 MB** of 500 MB of storage space.
Go to My Account

Omeka.net is a project of the Corporation for Digital Scholarship. Omeka and the Omeka logo are registered trademarks, used here with permission of the Center for History and New Media, George Mason University. Copyright © 2010-2011 Corporation for Digital Scholarship.

The Omeka Dashboard, add your site

After you have clicked on the link in your email, click on **Add a Site**. Fill in information about your site's URL, the title you want to use, and a description if you would like. Click on **Add Your Site**.

You have a new Omeka site!


omeka.net
Plans Showcase Help News About
View Site | Manage Site
Plans Showcase Help News About Contact Terms of Service Privacy Policy

Welcome, Miriam! | My Dashboard | My Account | Logout

Dashboard

You currently have the **Basic** plan, using **1** of 1 sites and **0 MB** of 500 MB of storage space.
Go to My Account

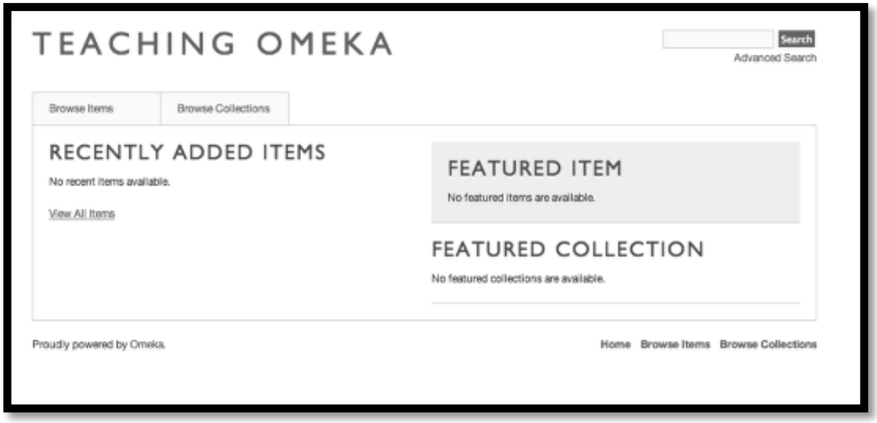
Site	Your Role	Storage Used	Delete
teachingsite.omeka.net	Owner	0 MB	Delete

Omeka.net is a project of the Corporation for Digital Scholarship. Omeka and the Omeka logo are registered trademarks, used here with permission of the Center for History and New Media, George Mason University. Copyright © 2010-2011 Corporation for Digital Scholarship.

The Omeka Dashboard, view your site

To see what the website looks like, click on **View Site**.

An empty Omeka site



RECENTLY ADDED ITEMS
No recent items available.

FEATURED ITEM
No featured items are available.

FEATURED COLLECTION
No featured collections are available.

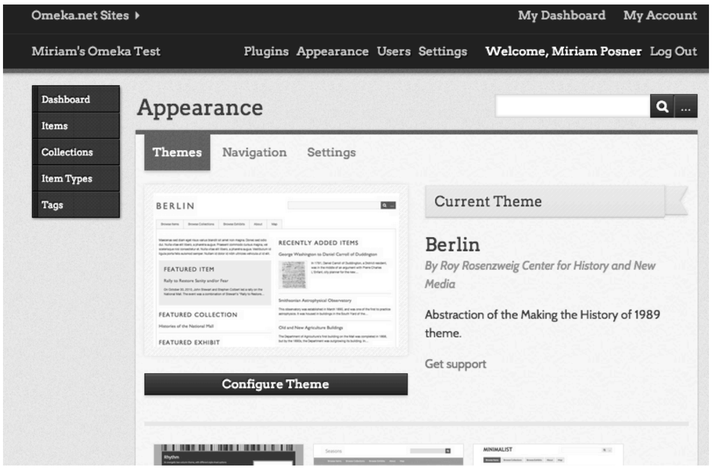
Proudly powered by Omeka.

The public view of the website

This is the public-facing element of your empty Omeka site. It is currently empty, waiting for you to fill it in. You will need to return to the dashboard to begin filling in the website. To get back to your dashboard, click the

Back button or enter **http://www.nameofyoursite.omeka.net/admin**. This time, click on **Manage Site**.

Switch themes



Current Theme

Berlin

By Roy Rosenzweig Center for History and New Media

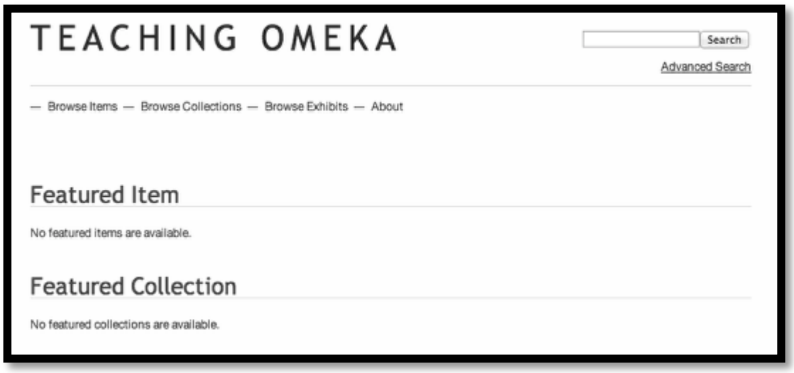
Abstraction of the Making the History of 1989 theme.

Get support

Switching Omeka Themes

Omeka allows you to change the look of your public-facing site by switching themes. To do this, click on **Settings** (at the top right of your dashboard), then select **Themes** on the left side of the page. Switch themes by selecting one of the options on the page. Press the green **Switch Theme** button to activate your new theme. Then visit your public site by clicking on **View Public Site** at the top right. If you do not immediately see the new theme, try doing a hard refresh[449] on your browser.

You have a new theme!


Search
Advanced Search
— Browse Items — Browse Collections — Browse Exhibits — About

TEACHING OMEKA

Featured Item

No featured items are available.

Featured Collection

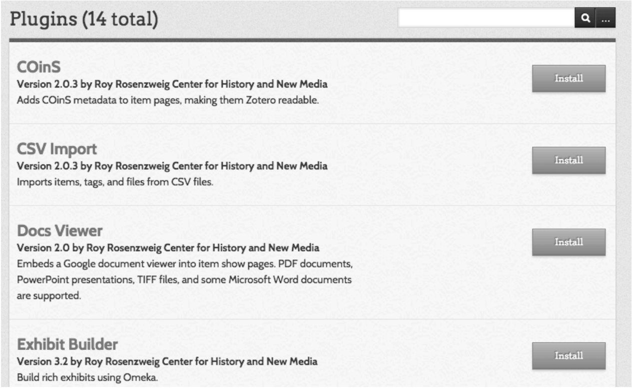
No featured collections are available.

Your site with a new Omeka theme

Once you have checked out your new theme, head back to your dashboard. You can switch back to your old theme, keep this one, or select one of the other options.

[449] 'Bypass your cache', *Wikipedia*: https://en.wikipedia.org/wiki/Wikipedia:Bypass_your_cache

Install plugins



Plugins (14 total)

COinS
Version 2.0.3 by Roy Rosenzweig Center for History and New Media
Adds COinS metadata to item pages, making them Zotero readable.

CSV Import
Version 2.0.3 by Roy Rosenzweig Center for History and New Media
Imports items, tags, and files from CSV files.

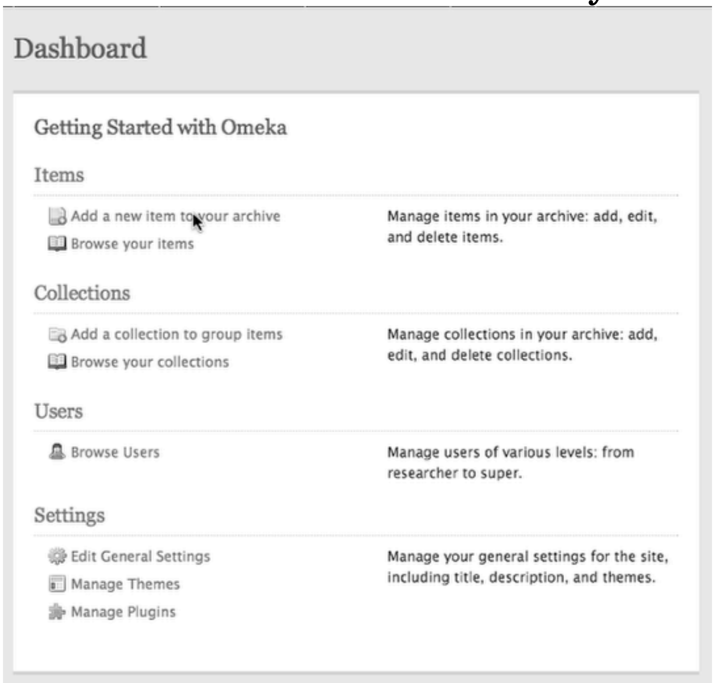
Docs Viewer
Version 2.0 by Roy Rosenzweig Center for History and New Media
Embeds a Google document viewer into item show pages. PDF documents, PowerPoint presentations, TIFF files, and some Microsoft Word documents are supported.

Exhibit Builder
Version 3.2 by Roy Rosenzweig Center for History and New Media
Build rich exhibits using Omeka.

Installing Omeka plugins

Your Omeka site comes with plugins, which are snippets of pre-written code that offer some extra functionality. These plugins are deactivated by default. If you want to use this extra functionality you need to enable the desired plugin. To do that, click on the red **Settings** button at the top right of the dashboard screen. On the following page, click the **Install** button next to **Exhibit Builder** and **Simple Pages**. On the following page you will be given additional options, but leave these as they are for now.

Add an item to your archive



Dashboard

Getting Started with Omeka

Items

Add a new item to your archive
Browse your items

Manage items in your archive: add, edit, and delete items.

Collections

Add a collection to group items
Browse your collections

Manage collections in your archive: add, edit, and delete collections.

Users

Browse Users

Manage users of various levels: from researcher to super.

Settings

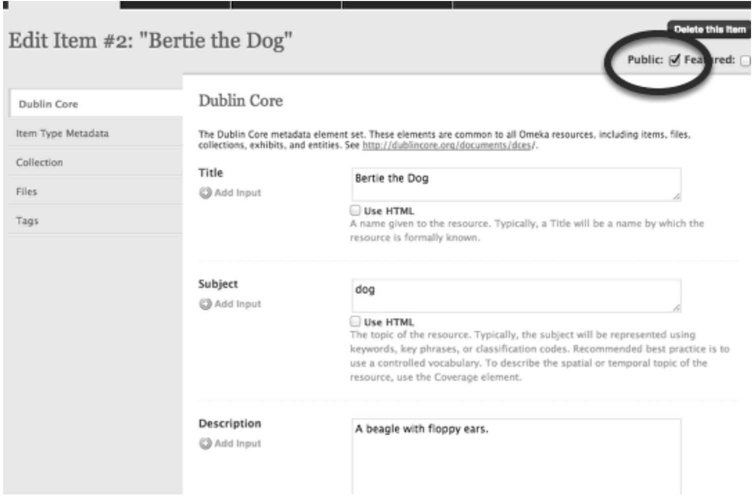
Edit General Settings
Manage Themes
Manage Plugins

Manage your general settings for the site, including title, description, and themes.

Add an item to your Omeka archive

Click on Add a new item to your archive.

Describe your new item



Edit Item #2: "Bertie the Dog"

Dublin Core

The Dublin Core metadata element set. These elements are common to all Omeka resources, including items, files, collections, exhibits, and entities. See http://dublincore.org/documents/dces/.

Title: Bertie the Dog

□ Use HTML

A name given to the resource. Typically, a Title will be a name by which the resource is formally known.

Subject: dog

□ Use HTML

The topic of the resource. Typically, the subject will be represented using keywords, key phrases, or classification codes. Recommended best practice is to use a controlled vocabulary. To describe the spatial or temporal topic of the resource, use the Coverage element.

Description: A beagle with floppy ears.

Describe an Omeka item

Remember, **Dublin Core** refers to the descriptive information you will enter about your item. All of this information is optional, and you cannot really do it wrong. But try to be consistent. (If you are interested in learning about each of the Dublin Core fields and how to use them consistently, read more about them in the Dublin Core documentation.)[450]

Be sure to click the **Public** checkbox so that your item is viewable by the general public. If you do not click that box, only people who are logged into your site will be able to see the item.

To add multiple fields — for example, if you want to add multiple subjects for your item — use the green **Add input** button to the left of the text boxes.

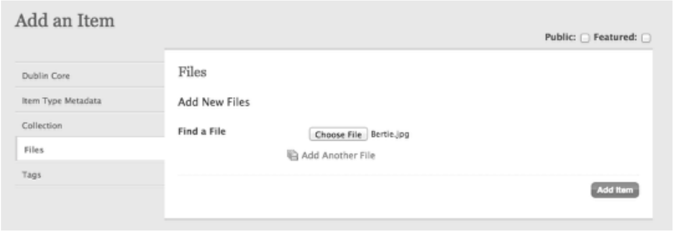
[450] 'DCMI Metadata Terms': http://dublincore.org/documents/dcmi-terms/

To what does the metadata really refer?

Is the metadata referring to Bertie, my dog, or this photograph of Bertie?

I am creating an item record for my dog, Bertie. But am I describing Bertie *himself* or a *photograph* of Bertie? If it is the former, the **Creator** would be — well, I guess that depends on your religious outlook. If it is the latter, the creator would be Brad Wallace, who took the photo. The decision about whether you are describing the object or the representation of the object is up to you. But once you have decided, be consistent.

Attach a file to your item record



Find a File: Bertie.jpg

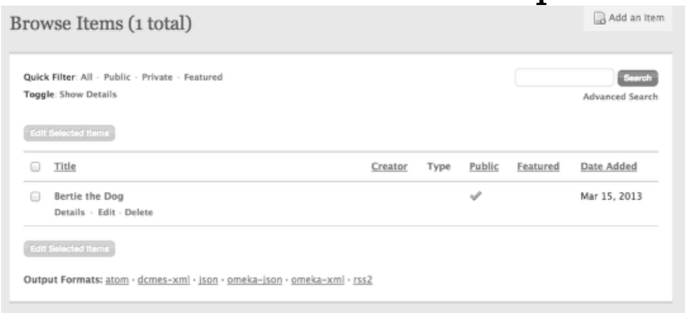
Attach a file to an Omeka item

Once you have finished adding Dublin Core metadata, you can attach a file to your item record by clicking **Files** to the left of the Dublin Core form. (You do not have to click **Add Item** before you do this; Omeka will automatically save your information.) You can add multiple files, but be aware that the Basic plan only comes with 500 MB of storage space.

Once you have added a file or files, you can add **Tags** by clicking on the button. You can also click on **Item Type Metadata** to choose the category — person, place, animal, vegetable, mineral — your item is. If you do not see the appropriate item type for your item, do not worry. You can add a new item type later.

When you are finished, click the green **Add Item** button.

Your completed item



Browse Items (1 total)

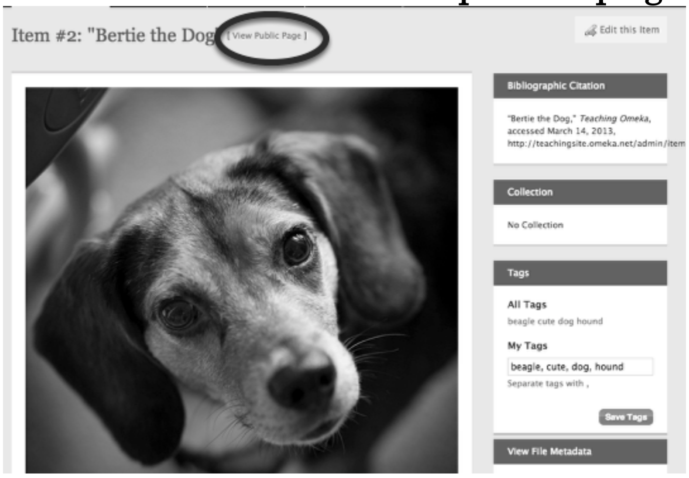
□	Title	Creator	Type	Public	Featured	Date Added
□	Bertie the Dog Details · Edit · Delete			✓		Mar 15, 2013

Output Formats: atom · dcmes-xml · json · omeka-json · omeka-xml · rss2

A completed Omeka item

This list contains all the items you have added, which so far numbers only one. Notice the green checkmark that appears in the **Public** column. To see what the page for your new item looks like, click on the name of the item.

This is not the public page for your item



Item #2: "Bertie the Dog"

"Bertie the Dog," *Teaching Omeka*, accessed March 14, 2013, http://teachingsite.omeka.net/admin/item

No Collection

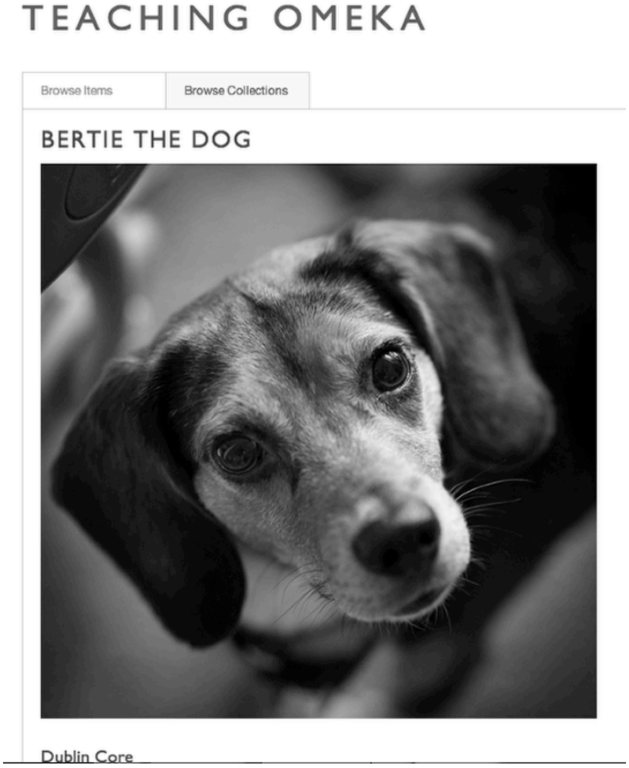
beagle cute dog hound

beagle, cute, dog, hound

The private view of your item page

It may look like it, but this page is not what a non-logged-in user will see when she navigates to the page for your item. To see what a user would see, click on **View Public Page**. (Or you can continue to edit the item by clicking on **Edit this item** at the top right.)

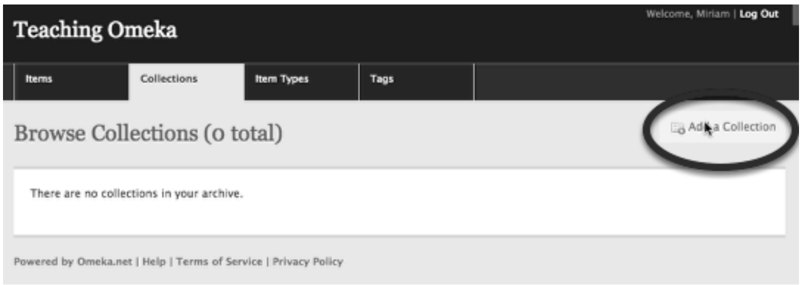
The public page for your item



The public page of an Omeka item

This is what a general user will see if she navigates to your page.

Create a collection

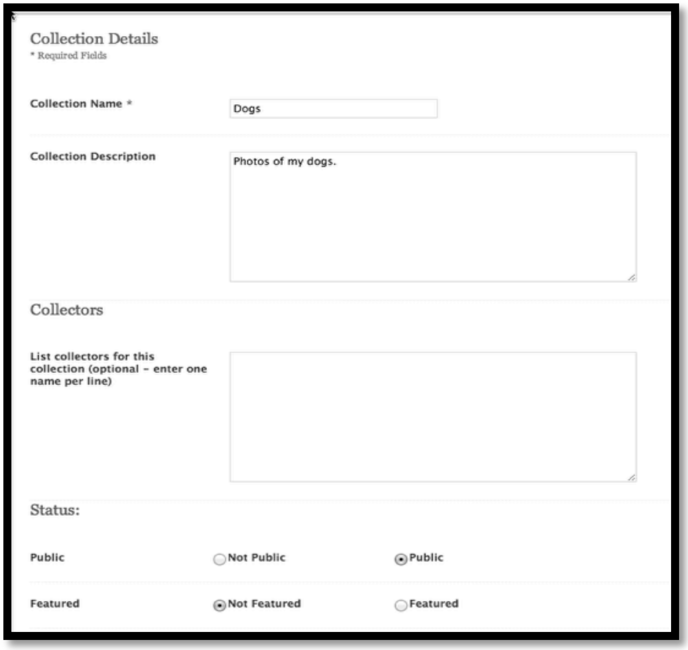

Teaching Omeka
Welcome, Miriam | Log Out
Items
Collections
Item Types
Tags
Browse Collections (0 total)
Add a Collection
Powered by Omeka.net | Help | Terms of Service | Privacy Policy

There are no collections in your archive.

Create an Omeka collection

Once you have several items, you can begin to bring order to those items by grouping them together into collections. To do this, return to your dashboard, click on the **Collections** tab, and click on **Add a Collection**.

Enter information about your collection



Collection Details
* Required Fields

Collection Name *: Dogs

Collection Description: Photos of my dogs.

Collectors

List collectors for this collection (optional – enter one name per line):

Status:

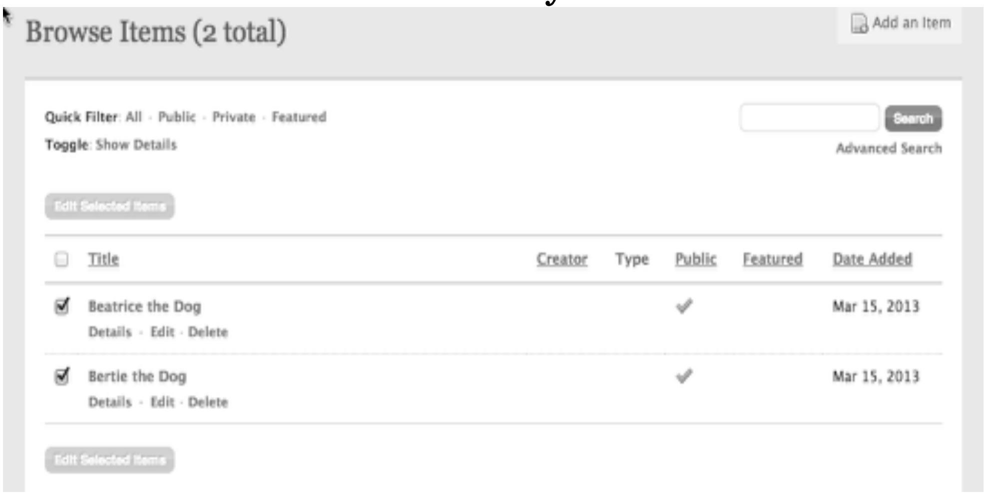
Public ○ Not Public ● Public

Featured ● Not Featured ○ Featured

Enter information about your Omeka collection

In Omeka, metadata is key. Enter some information about your new collection, and remember to click on the **Public** button near the bottom of the page. Then save your collection. You now have an empty collection.

Add items to your collection



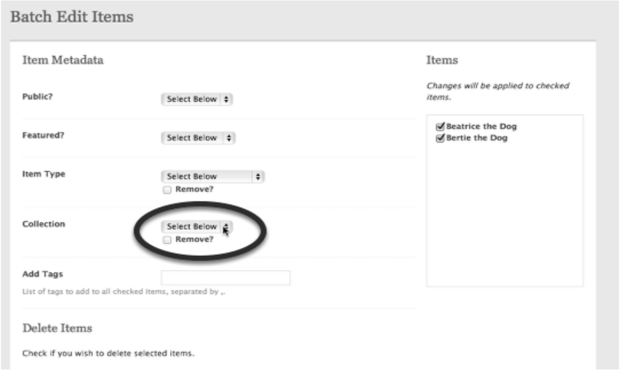
Browse Items (2 total)

	Title	Creator	Type	Public	Featured	Date Added
☑	Beatrice the Dog Details · Edit · Delete			✓		Mar 15, 2013
☑	Bertie the Dog Details · Edit · Delete			✓		Mar 15, 2013

Add items to an Omeka collection

To add items to the collection you have just created, click on the **Items** tab. From your **Browse Items** list, click the boxes of the items that belong in your new collection. Then click on the green **Edit Selected Items** button.

Choose the collection



Batch Edit Items

Item Metadata

Public? Select Below

Featured? Select Below

Item Type Select Below
□ Remove?

Collection Select Below
□ Remove?

Add Tags
List of tags to add to all checked items, separated by ,.

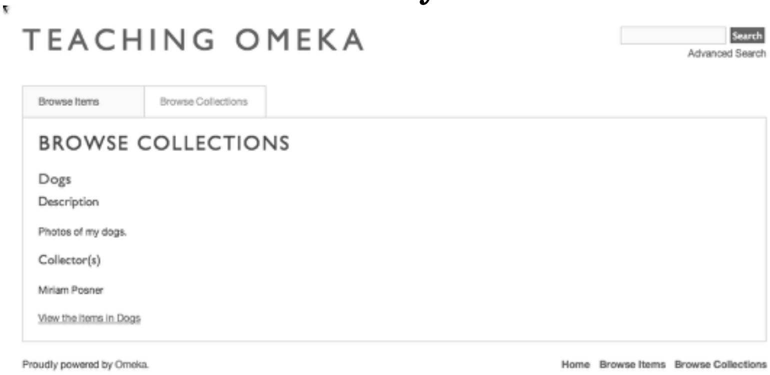
Delete Items
Check if you wish to delete selected items.

Items
Changes will be applied to checked items.
☑ Beatrice the Dog
☑ Bertie the Dog

Choose the Omeka collection to which you wish to add your item

On the **Batch Edit Items** page, select the Collection you would like to add your items to. (Also, take note of all the other options you have on this page.)

View your new collection



BROWSE COLLECTIONS

Dogs

Description

Photos of my dogs.

Collector(s)

Miriam Posner

View the Items in Dogs

View the Omeka collection

To view the new collection, return to the public site. If you click on the **Browse Collections** tab on the public-facing site, you should now have a new collection containing the items you identified.

Now that you have added some items and grouped them into a collection, take some time to play with your site. It is beginning to take shape now that you have both individual items and thematic units. But Omeka can do even more. We will talk about that in the next lesson.

Further Resources

The Omeka team has put together great resources on the software's help pages[451]

About the Author

Miriam Posner is the digital humanities program coordinator at the University of California, Los Angeles.

[451] 'Help for Omeka.net': http://info.omeka.net/

41. Creating an Omeka.net Exhibit

Miriam Posner – 2013

In the previous lesson, ‘Up and Running with Omeka.net’, you added items to your Omeka.net site and grouped them into collections. Now you are ready for the next step: taking your users on a guided tour through the items you have collected.

Before you begin: Plan your exhibit

It pays to do some thinking before you launch into creating an exhibit. You will be creating both sections and pages, and you will need to give some thought to the argument you want to make and how you intend to make it. In this lesson that follows, I use the silly example of my dogs. But what if I were discussing, say, silent film? My sections might be thematic (comedies, romances, dramas), chronological (early silent film, the transitional period, classical era), or stylistic (modernist, impressionist, narrative). It all depends on the message I want to convey to the site’s visitors. You might draw out a map of your exhibit, showing where you want to put each digital asset.

Add an exhibit

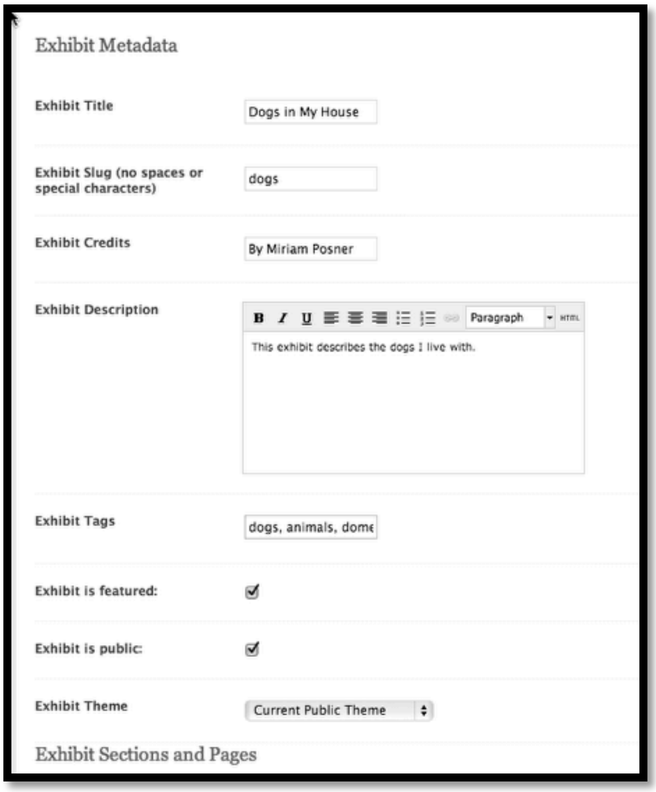


Exhibit Metadata

Exhibit Title: Dogs in My House

Exhibit Slug (no spaces or special characters): dogs

Exhibit Credits: By Miriam Posner

Exhibit Description: This exhibit describes the dogs I live with.

Exhibit Tags: dogs, animals, dome

Exhibit is featured: ☑

Exhibit is public: ☑

Exhibit Theme: Current Public Theme

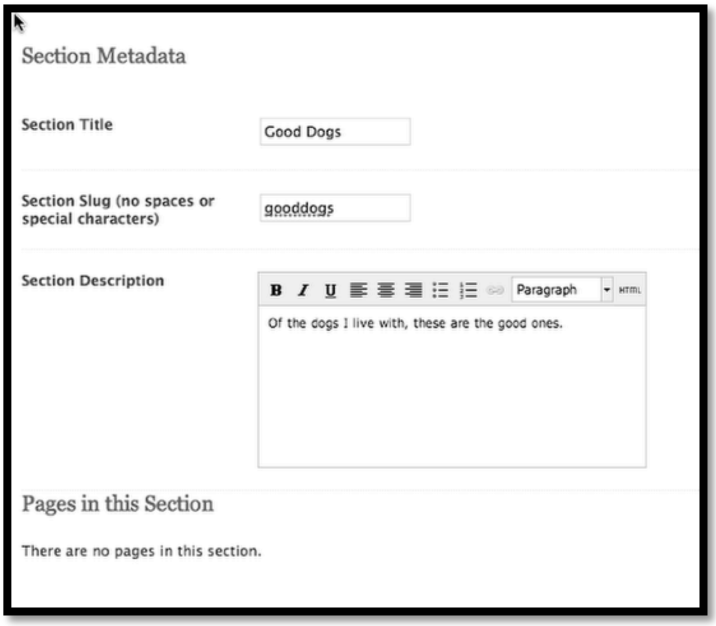
Exhibit Sections and Pages

Add an exhibit in Omeka

A collection is just a list of objects. An exhibit, on the other hand, is a guided tour through your items, complete with descriptive text and customized layouts. To create one, click on the **Exhibits** tab and then **Add**

an exhibit. Fill out the form on the top half of the page. A **slug** is a machine-readable name for your exhibit and will become part of your URL. The slug of this lesson is "creating-an-omeka-exhibit," which you can see in the URL at the top of your browser.

Add a section



Section Metadata

Section Title: Good Dogs

Section Slug (no spaces or special characters): gooddogs

Section Description: Of the dogs I live with, these are the good ones.

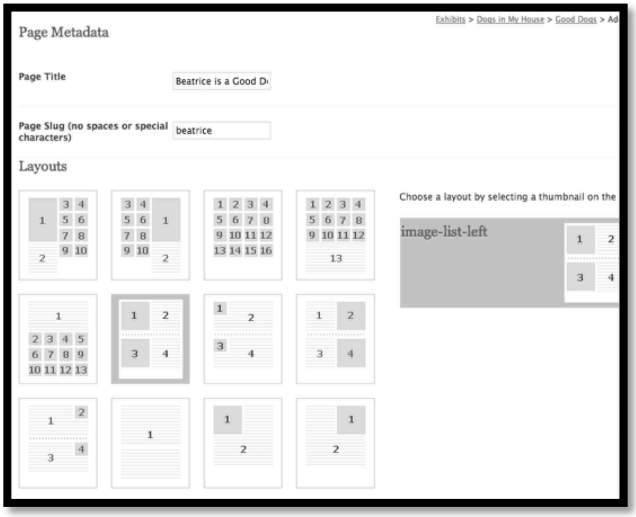
Pages in this Section

There are no pages in this section.

Add a section in Omeka

Every exhibit has sections and pages — like the chapters and individual pages in a book. Add a new section by clicking on the green **Add Section** button and then filling out the information on the following page.

Add a page



Exhibits > Dogs in My House > Good Dogs > Ad

Page Metadata

Page Title: Beatrice is a Good D

Page Slug (no spaces or special characters): beatrice

Layouts

Choose a layout by selecting a thumbnail on the

image-list-left

Add a page in Omeka

Pages are where you will stick the actual items in your exhibit. Click on the green **Add Page** button. On the following page, you will enter some information and pick a layout for your exhibit page. The small blue squares indicate item thumbnails, the large blue squares indicate full-sized images, and the lined areas indicate descriptive text. Pick a layout; you can change it later. Then click on **Save Changes**.

Add items to your page

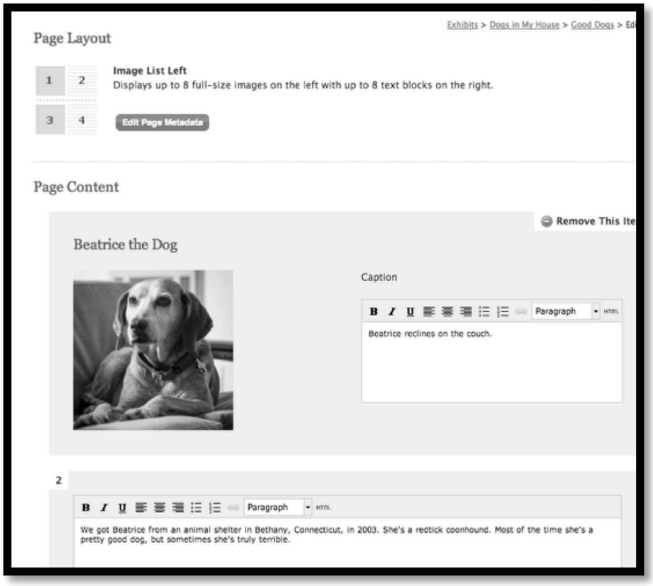


Image List Left
Displays up to 8 full-size images on the left with up to 8 text blocks on the right.

Beatrice the Dog

Beatrice reclines on the couch.

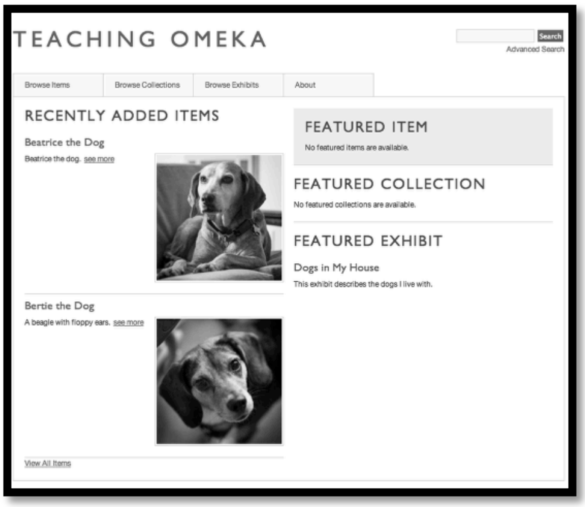
We got Beatrice from an animal shelter in Bethany, Connecticut, in 2003. She's a redtick coonhound. Most of the time she's a pretty good dog, but sometimes she's truly terrible.

Add items to your page

On the page that follows, you will see a numbered grid. You will fill in that grid by attaching items (in the places indicated by blue boxes) and typing in descriptive information about your item. Remember, an exhibit is a kind of guided tour through your items, so try to write descriptions that guide the reader from one item to the next. When you are finished adding items, you can add another page, or another section, or both.

When you are done, return to your public site to see how your Omeka site looks.

You have an Omeka site!



Beatrice the Dog
Beatrice the dog.

Bertie the Dog
A beagle with floppy ears.

No featured items are available.

No featured collections are available.

Dogs in My House
This exhibit describes the dogs I live with.

The completed Omeka Exhibit

Now your site has items, collections, and an exhibit — all the basic units of an Omeka site.

Further Resources

The Omeka team has put together great resources on the software's help pages.[452]

About the Author

Miriam Posner is the digital humanities program coordinator at the University of California, Los Angeles.

[452] 'Help for Omeka.net': http://info.omeka.net/

42. Intro to Google Maps and Google Earth

Jim Clifford, Joshua MacFadyen, Daniel Macfarlane – 2013

Google Maps

Google Maps and Google Earth provide an easy way to start creating digital maps. With a Google Account you can create and edit personal maps by clicking on My Places. In the new Google Maps interface, click on the gear menu [icon] at the upper right of the menu bar, and select My Places. The new (as of summer 2013) interface provides a new way of creating custom maps: ‘Google Maps Engine Lite’ allows users to import and add data onto the map to visualize trends.

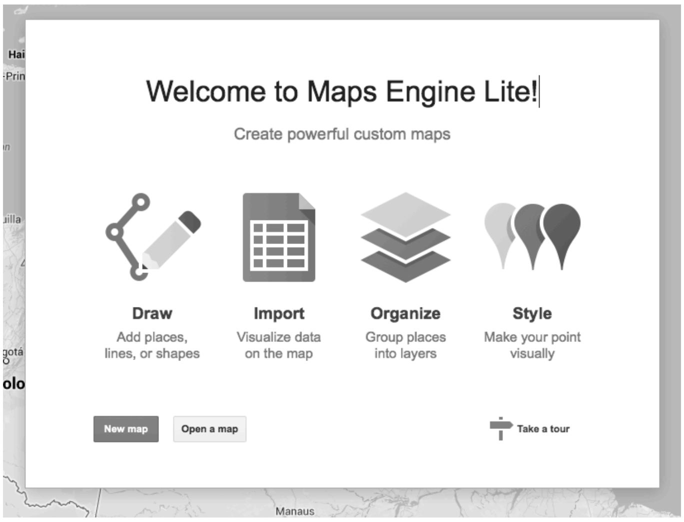
In Maps Engine Lite you can choose between several different base maps (including the standard satellite, terrain, or standard maps) and add points, lines and polygons. It is also possible to import data from a spreadsheet, if you have columns with geographical information (i.e. longitudes and latitudes or place names). This automates a formerly complex task known as geocoding. Not only is this one of the easiest ways to begin plotting your historical data on a map, but it also has the power of Google's search engine. As you read about unfamiliar places in historical documents, journal articles or books, you can search for them using Google Maps. It is then possible to mark numerous locations and explore how they relate to each other geographically. Your personal maps are saved by Google (in their cloud), meaning you can access them from any computer with an internet connection. You can keep them private or embed them in your website or blog. Finally, you can export your points, lines, and polygons as KML files and open them in Google Earth or Quantum GIS.

Getting Started

Open your favorite browser

Go to Google's Maps Engine Lite: https://mapsengine.google.com/map/?gmp=mpp

Log in to your Google Account if you aren't already logged in (follow the basic instructions to create an account if necessary)



Welcome to Maps Engine Lite!

Create powerful custom maps

Draw
Add places, lines, or shapes

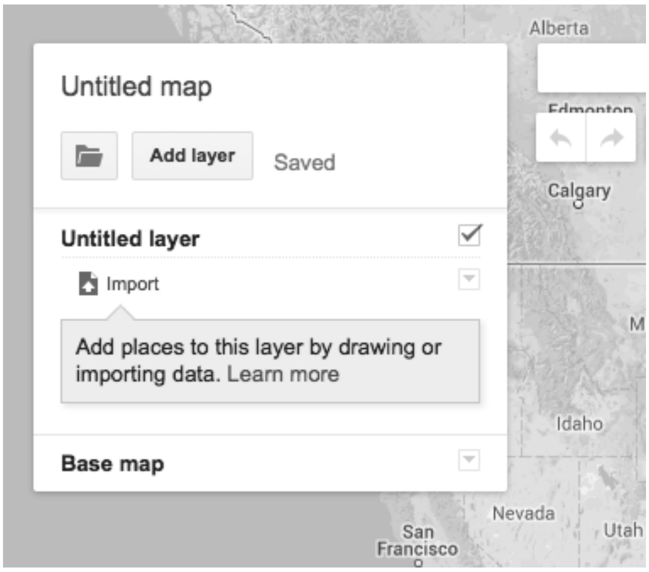
Import
Visualize data on the map

Organize
Group places into layers

Style
Make your point visually

Select Take a Tour at the bottom right for an introduction to how Maps Engine Lite works

The Tour will first prompt you to Click New Map



Untitled map

Add layer Saved

Untitled layer

Import

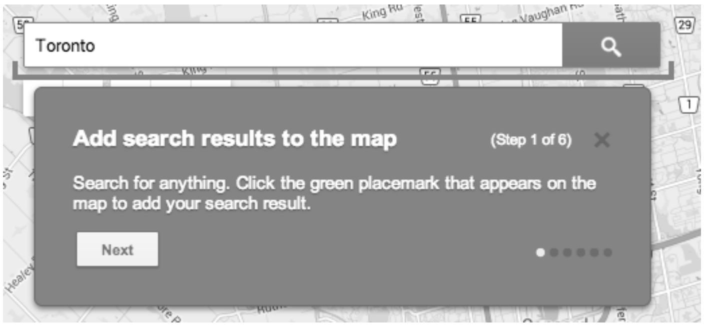
Add places to this layer by drawing or importing data. Learn more

Base map

At the upper left corner, a menu box appears, titled 'Untitled Map'. By clicking on the title you can rename as 'My test map' or a title of your choice.

The Tour next prompts you to search for a place in the search bar. Try searching the location of your current research project. You can then click on the location and add it to your map by clicking 'add to map'. This is the simplest method of adding points to your new map. Try searching for some historical place names that no longer exist (Ontario's Berlin or Constantinople). You will find mixed results, where Google often identifies the correct location, but also offers up incorrect alternatives. This is important to keep in mind when creating spreadsheet, as it is normally better to use the modern place names and avoid risking that Google with choose the wrong Constantinople.

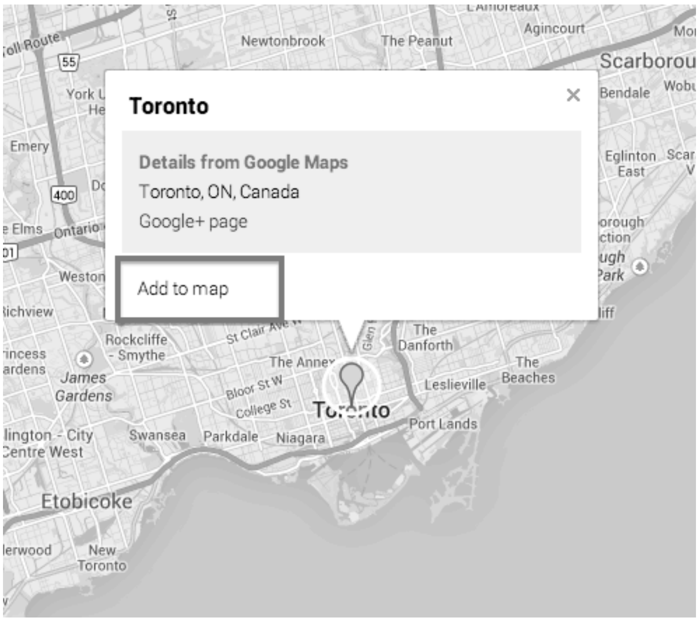
CLICK NEXT on the Google Maps Tour.



Toronto

Add search results to the map (Step 1 of 6)

Search for anything. Click the green placemark that appears on the map to add your search result.



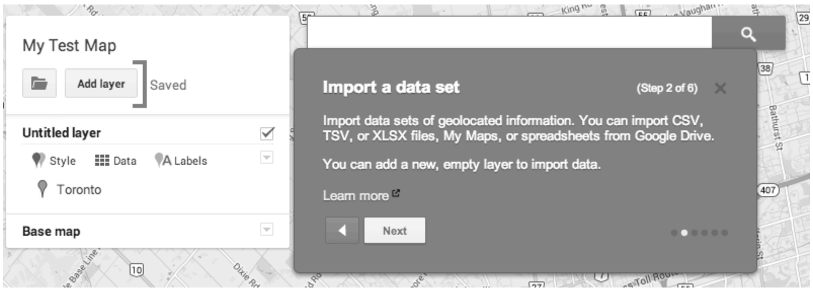
Toronto

Details from Google Maps

Toronto, ON, Canada

Google+ page

The Tour next prompts you to Import a Dataset. Click the ADD Layer button. Then click the Import.



My Test Map

Untitled layer

Toronto

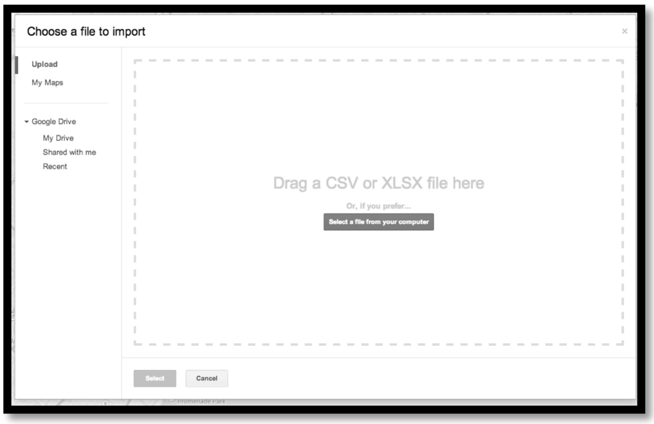
Base map

Import a data set (Step 2 of 6)

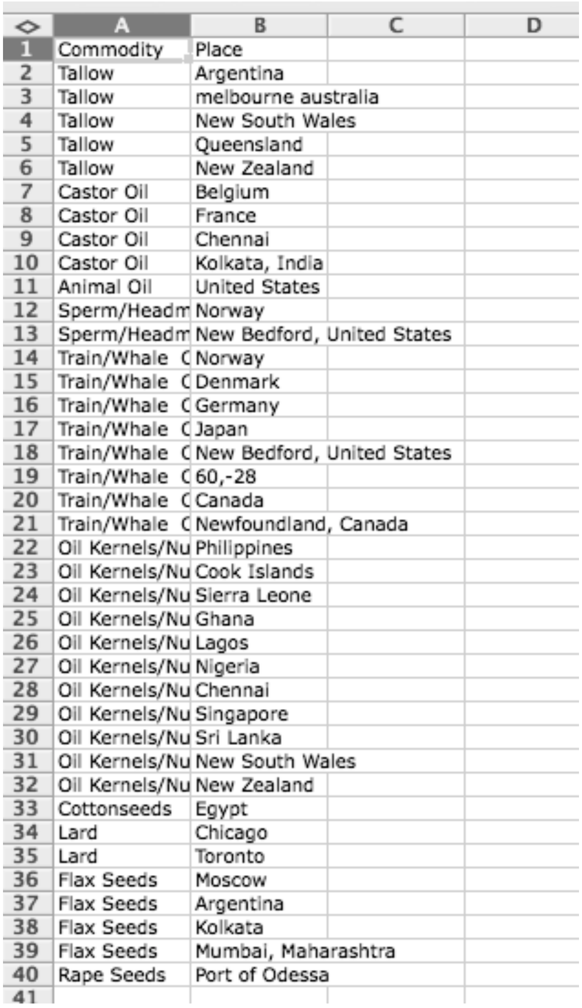
Import data sets of geolocated information. You can import CSV, TSV, or XLSX files, My Maps, or spreadsheets from Google Drive.

You can add a new, empty layer to import data.

A new window will pop up and give you the option of importing a CSV (comma separated value) or XLXS (Microsoft Excel) file. These are two common spreadsheet formats; CSV is simple and universal, XLXS is the MS Excel format. You can also work with a Google spreadsheet from your Drive account.



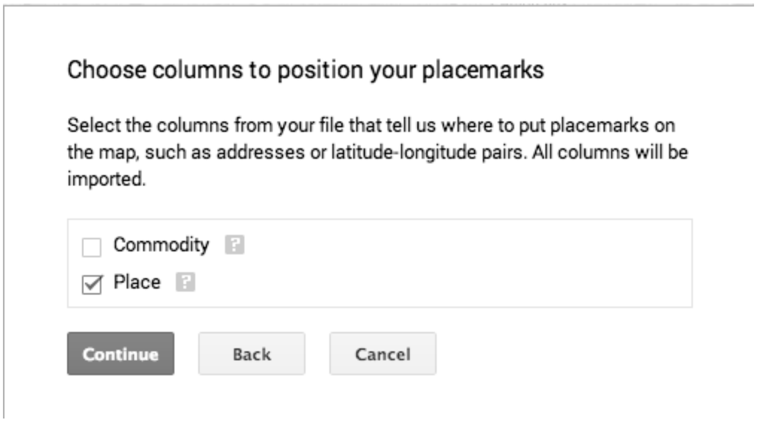
Download this sample data and located it on your computer: UK Global Fat Supply CSV file. If you open the file in Excel or another spreadsheet program, you'll find a simple two column dataset with a list of different kinds of fats and the associated list of places. This data was created using British import tables from 1896.



◇	A	B	C	D
1	Commodity	Place		
2	Tallow	Argentina		
3	Tallow	melbourne australia		
4	Tallow	New South Wales		
5	Tallow	Queensland		
6	Tallow	New Zealand		
7	Castor Oil	Belgium		
8	Castor Oil	France		
9	Castor Oil	Chennai		
10	Castor Oil	Kolkata, India		
11	Animal Oil	United States		
12	Sperm/Headm	Norway		
13	Sperm/Headm	New Bedford, United States		
14	Train/Whale C	Norway		
15	Train/Whale C	Denmark		
16	Train/Whale C	Germany		
17	Train/Whale C	Japan		
18	Train/Whale C	New Bedford, United States		
19	Train/Whale C	60,-28		
20	Train/Whale C	Canada		
21	Train/Whale C	Newfoundland, Canada		
22	Oil Kernels/Nu	Philippines		
23	Oil Kernels/Nu	Cook Islands		
24	Oil Kernels/Nu	Sierra Leone		
25	Oil Kernels/Nu	Ghana		
26	Oil Kernels/Nu	Lagos		
27	Oil Kernels/Nu	Nigeria		
28	Oil Kernels/Nu	Chennai		
29	Oil Kernels/Nu	Singapore		
30	Oil Kernels/Nu	Sri Lanka		
31	Oil Kernels/Nu	New South Wales		
32	Oil Kernels/Nu	New Zealand		
33	Cottonseeds	Egypt		
34	Lard	Chicago		
35	Lard	Toronto		
36	Flax Seeds	Moscow		
37	Flax Seeds	Argentina		
38	Flax Seeds	Kolkata		
39	Flax Seeds	Mumbai, Maharashtra		
40	Rape Seeds	Port of Odessa		
41				

Drag the file into the box provided by Google Maps.

You will then be promoted to choose which column Google should use to identify a the location. Choose Place.



Choose columns to position your placemarks

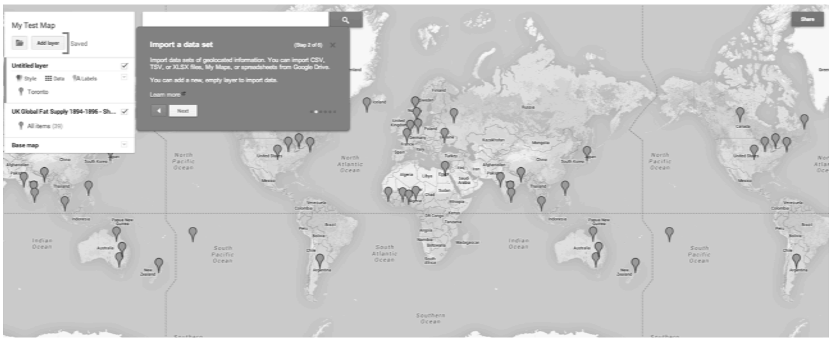
Select the columns from your file that tell us where to put placemarks on the map, such as addresses or latitude-longitude pairs. All columns will be imported.

□ Commodity

☑ Place

You will then be promoted again to choose which column should be used for the label. Choose 'Commodity'.

You should now have a global map of the major exporters of fat to Britain during the mid-1890s.



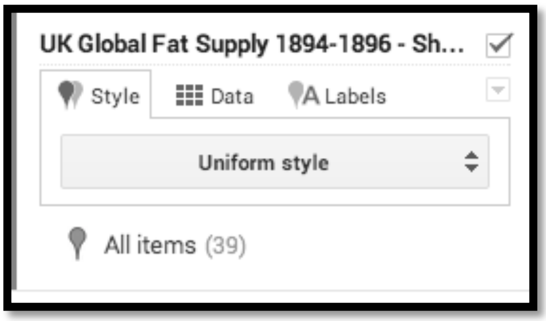
Import a data set (Step 2 of 6)

Import data sets of geolocated information. You can import CSV, TSV, or XLSX files, My Maps, or spreadsheets from Google Drive.

You can add a new, empty layer to import data.

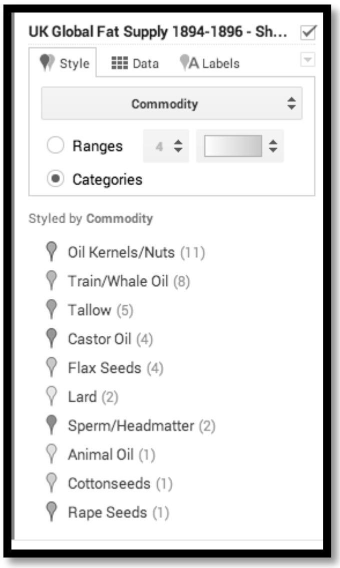
You can now explore the data in more detail and change the Style to distinguish between the different types of fats.

Click on the UK Global Fats Layer, then click on Style and finally click on Uniform Style and change it to Style by Data Column: Commodities. On the left hand side, the legend will show the amount of occurrences of each style in brackets, e.g. 'Flax Seeds (4)'.



UK Global Fat Supply 1894-1896 - Sh... ☑

All items (39)



UK Global Fat Supply 1894-1896 - Sh... ☑

Commodity

○ Ranges 4

● Categories

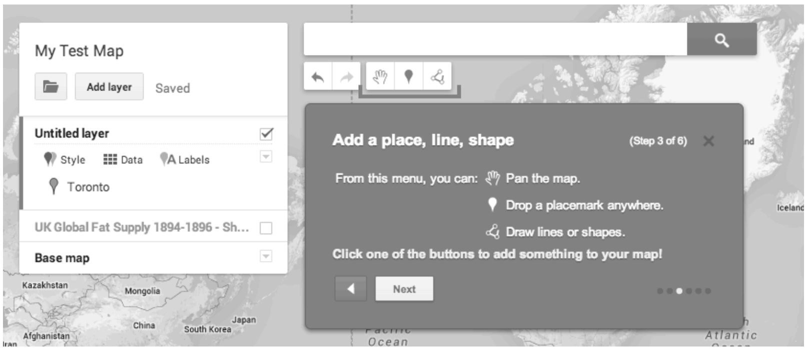
Styled by Commodity

- Oil Kernels/Nuts (11)
- Train/Whale Oil (8)
- Tallow (5)
- Castor Oil (4)
- Flax Seeds (4)
- Lard (2)
- Sperm/Headmatter (2)
- Animal Oil (1)
- Cottonseeds (1)
- Rape Seeds (1)

Continue to play with the options.

- This feature provides a powerful tool to display historical datasets. It does have limitations, however, as Google Maps will only import the first 100 rows of a spreadsheet. At this point it only allows you to include three datasets in a map, so a maximum of 300 features.

- When you are done exploring this feature **click Next** on the Google Tour. (Note: The Google tour crashed a few times while we created this lesson. It is possible to go back to the beginning to start a new Tour. Feel free to skip through the Import Data section if the Tour crashes at this point.)



My Test Map

Untitled layer ☑

UK Global Fat Supply 1894-1896 - Sh... □

Base map

Add a place, line, shape (Step 3 of 6)

From this menu, you can: Pan the map.

Drop a placemark anywhere.

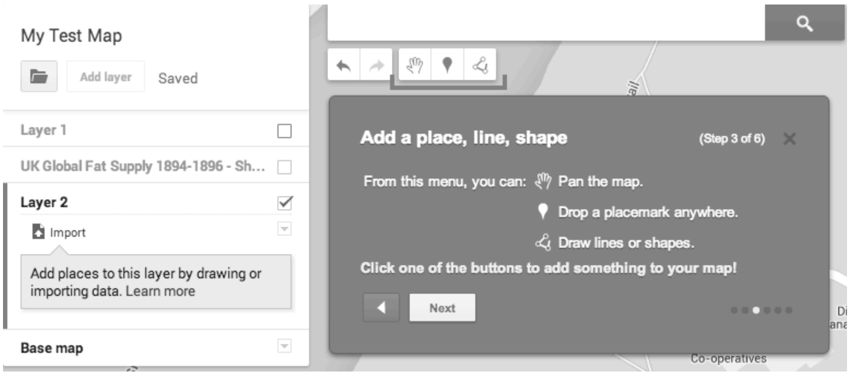
Draw lines or shapes.

Click one of the buttons to add something to your map!

Creating Vector Layers

At this stage of the tutorial, we create map layers (known more formally as vector layers). Vector layers are one of the main components of digital mapping (including GIS). They are simply points, lines, or polygons used to represent geographic features. Points can be used to identify and label key locations, lines are often used for streets or railroads, and polygons allow you to represent area (fields, buildings, city wards, etc). They work the same in Google Maps as they do in GIS. The big limitation is that you can only add limited information into the database tables associated with the points, lines, or polygons. This is a problem as you scale up your digital mapping research, but it is not a problem when you are starting out. In Google Maps you can add a label, a text description, and links to a website or photo. More information about creating historical vectors in a full GIS is available in 'Creating New Vector Layers in QGIS 2.0'.[453]

To add a layer, you can either click on the layer that has been created for you in the menu box, with the name 'Untitled Layer'. Click on 'Untitled Layer' and rename it 'Layer 1′. Or you can create another layer: click on the 'Add layer' button. This will add a new 'Untitled Layer' which you can name as 'Layer 2′. It should look like this:



Add places to this layer by drawing or importing data. Learn more

Add a place, line, shape (Step 3 of 6)

From this menu, you can: Pan the map.
Drop a placemark anywhere.
Draw lines or shapes.

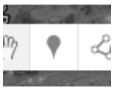
Click one of the buttons to add something to your map!

Note that to the right of Layer there is a checkbox – unchecking this box turns off (i.e. it doesn't appear on the map) a layer and its information. Uncheck the UK Global Fats layer and click on Layer

Before adding vector layers we should consider which base map to use. At the bottom of the menu window, there is a line that says 'base map'. A base map is a map** depicting background reference information such as roads, borders, landforms, etc. on top of which layers containing different types of spatial information can be placed. Google's Maps Engine allows you to choose from a variety of base maps, depending on the kind of map you want to create. Satellite imagery is becoming a standard form of base map, but it is information-rich and may detract from the other map features you are trying to highlight. Some simple alternatives include 'light landmass', or even 'light political' if you require political boundaries.

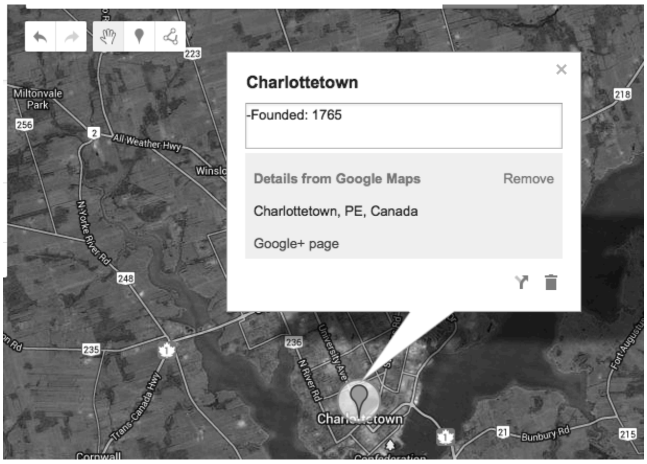
- Click on the arrow to the right of 'Base map' in the window; a submenu appears allowing you to choose different types of base maps. Choose 'Satellite'.

- Start by adding some Placemarks (the Google equivalent of a point). Click on the add Markers button underneath the search bar near the top of the window. Click on the spot on the map where you want the Placemark to appear.



[453] Jim Clifford, Josh MacFadyen, and Dan Macfarlane 'Creating New Vector Layers in QGIS 2.0', *The Programming Historian* (2013).

A box will pop up and give you the opportunity to label the Placemark and add a description into the text box. We added Charlottetown and included that it was founded in 1765 in the description box.



Charlottetown

-Founded: 1765

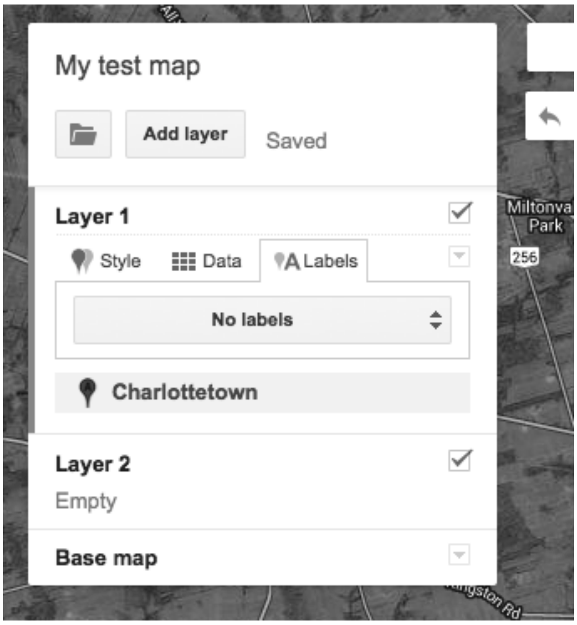
Details from Google Maps

Charlottetown, PE, Canada

Google+ page

Add a few more points, including labels and descriptions.

- You will notice that your Placemark now appears under Layer 1 on the left of the screen in your menu window. There is a place to change icon shape and icon colour if you click on the symbol just to the right of the Placemark name. Also, directly under the title Layer 1 there are menus titled Style, Data, and Labels. The Style menu controls different aspects of the Layer's appearance, while Data shows you the data you added in the description box for your Placemark. Labels menu allows you to control whether the name or description of your Placemark appears besides it on the actual map.



My test map

Saved

Layer 1 ☑

Labels: No labels

Charlottetown

Layer 2 ☑

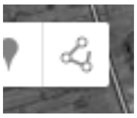
Empty

Base map

Now we will add some lines and shapes (called polygons in GIS software). Adding lines and polygons is a very similar process. We will draw some lines in a new layer (different types of points, lines, and shapes should be in separate layers).

Select Layer 2 in your menu box (you will know which layer you have selected because of the blue outline on the left of the box).

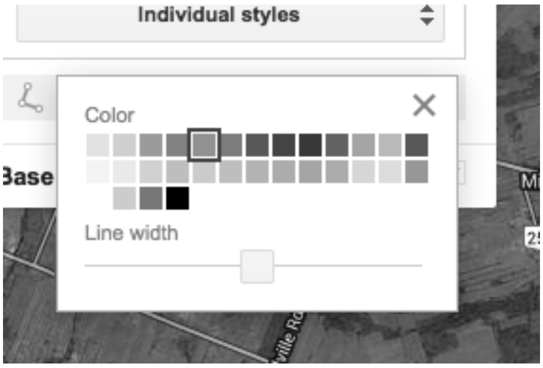
Click the 'add line or shape' icon box directly to the right of the Markers symbol:



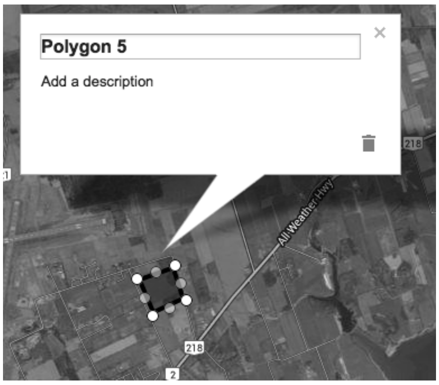
- Pick a road and click with your mouse along it, tracing the route for a while. Hit "enter" when you want to finish the line.

Again you can add a label (i.e. name a road) and description information.

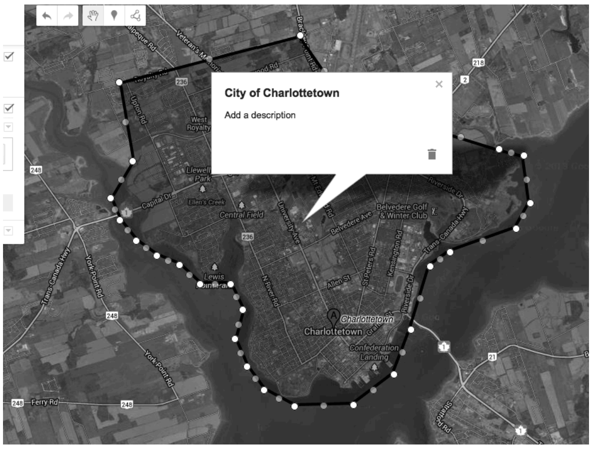
- You can also change the colour and width of the line. To do this, find the road you have drawn in Layer 2 in the menu box, and click to the right of the name of the road.



To create a polygon (a shape) you can connect the dots of the line to create an enclosed formation. To do this, start drawing and finish by clicking on the first point in your line. You can create simple shapes, such as a farmer's field, or much more complex shapes, such as the outline of a city (see examples below). Feel free to experiment with creating lines and polygons.


Polygon 5

Add a description



City of Charlottetown

Add a description

Like placemarks and lines, you can change the name and description of a polygon. You can also change the colour and line width by clicking on the icon to the right of your polygon name in the menu box. Here you can also change the transparency, which is discussed immediately below.

Note that the area bounded (i.e. inside) a polygon is shaded the same colour as the polygon outline. You can change the opaqueness of this shading by changing the 'transparency' which alters the extent to which you can clearly see the background image (your base map).

Now you can **click Next** on the Google Tour

The tour will again show you how to Change the colour and style of your points, lines and polygons

Click Next on the Google Tour.

The Tour will remind you how to click on layers to hid them or add them back on the map. Try adding the UK Global Fats layer again and zoom between your global and local data points.

Click Next on the Google Tour.

Share your custom map

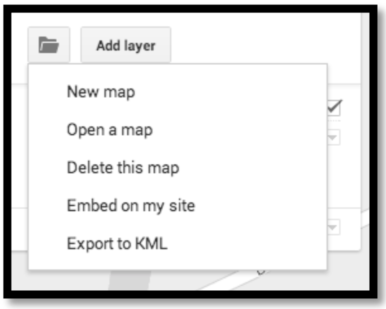
The best way to share the map online is by using the green **Share** button in the top right corner. This provides a link which can be share in an email or through social media like G+, Facebook, or Twitter.

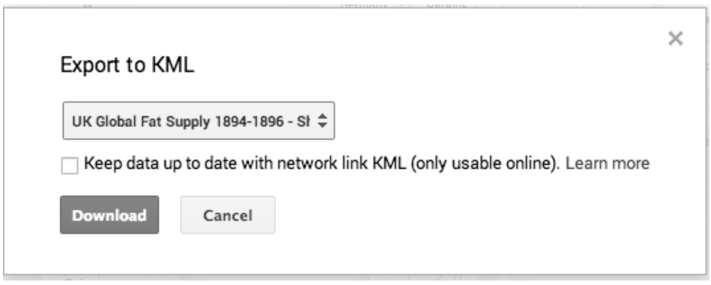
Another way to share a dynamic version of your map is to embed it in a blog or website using the "embed on my website" option under the **Save** menu. Selecting this option provides an inline frame or <iframe> tag that you can then insert into an HTML site. You can modify the height and width of the frame by changing the numbers in quotation marks.

Note: there is currently (as of summer 2013) no way to set the default scale or legend options of the embedded map, but if you need to eliminate the legend from the map that appears on your HTML site you can do so by reducing the width if the <iframe> to 580 or less.

You can also export the data as a KML file. It will give you the option to export the whole map or to select one layer in particular. Try exporting the UK Global Fats layer as a KML layer. You'll be able to import this data into other programs, including Google Earth and Quantum GIS. This is an important feature, as it means you can start working with digital maps using Google Map and still export your work into a GIS database in the future.

You can stop the lesson here if you think this free Google Map service provides all the tools you need for your research topic. Or you can keep going and learn about Google Earth and in lesson 2, Quantum GIS.




Export to KML

UK Global Fat Supply 1894-1896 - St

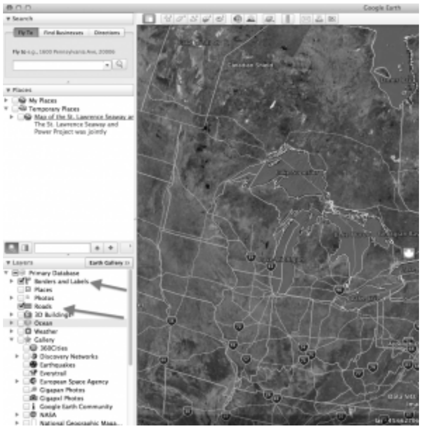
□ Keep data up to date with network link KML (only usable online). Learn more

Google Earth

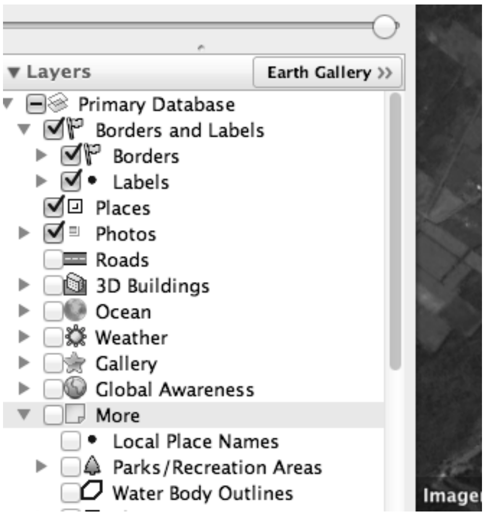
Google Earth works in much the same way as Google Maps Engine Lite, but has additional features. For example, it provides 3-D maps and access to data from numerous third party sources, including collections of historical maps. Google Maps doesn't require you to install software and your maps are saved in the cloud. Google Earth requires software installation and is not cloud-based, though maps you create can be exported.

Install Google Earth: http://www.google.com/earth/index.html

Open the program and familiarize yourself with the digital globe. Use the menu to add and remove layers of information. This is very similar to how more advanced GIS programs work. You can add and remove different kinds of geographical information including Political Boundaries (polygons), Roads (lines), and Places (points). See the red arrows in the following image for the location of these layers.

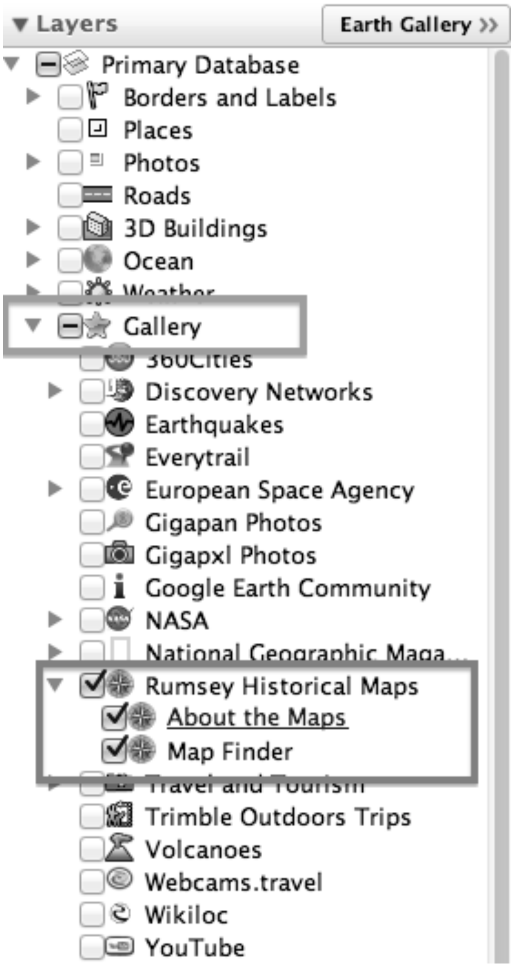


Note that under the 'Layer' heading on the lower left side of the window margin, Google provides a number of ready-to-go layers that can be turned on by selecting the corresponding checkbox.



- ▼ Layers — Earth Gallery >>
- ▼ ⊟ Primary Database
  - ▼ ☑ Borders and Labels
    - ▶ ☑ Borders
    - ▶ ☑ Labels
  - ☑ Places
  - ▶ ☑ Photos
  - □ Roads
  - ▶ □ 3D Buildings
  - ▶ □ Ocean
  - ▶ □ Weather
  - ▶ □ Gallery
  - ▶ □ Global Awareness
  - ▼ □ More
    - □ Local Place Names
    - ▶ □ Parks/Recreation Areas
    - □ Water Body Outlines

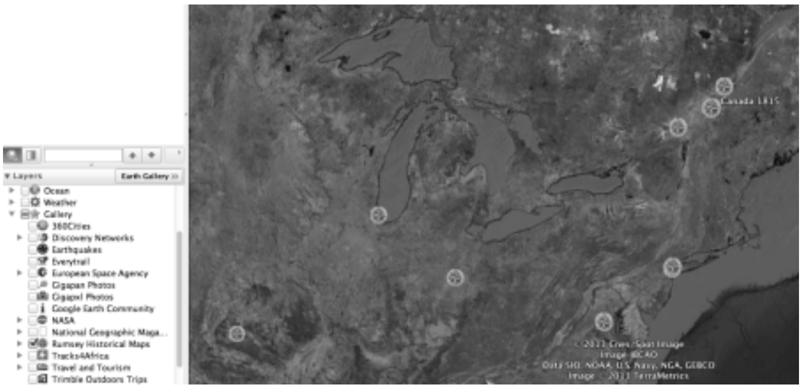
Google Earth also contains some scanned historical maps and aerial photographs (in GIS these types of maps, which are made up of pixels, are known as raster data). Under Gallery you can find and click Rumsey Historical Maps. This will add icons all over the globe (with a concentration in the United States) of scanned maps that have been georeferenced (stretched and pinned to match a location) onto the digital globe. This previews a key methodology in historical GIS. (You can also find historical map layers and other HGIS layers in the Earth Gallery). Take some time to explore a number of historical maps. See if there are any maps included in the Rumsey Collection that might be useful for your research or teaching. (You can find many more digitized, but not georeferenced maps at www.davidrumsey.com.)



▼ Layers — Earth Gallery »

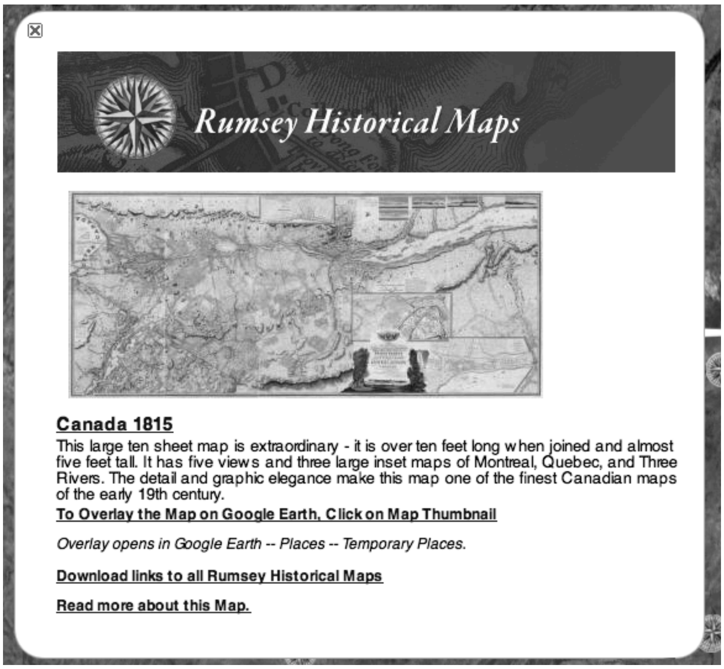
- ▼ ⊟ Primary Database
  - ▶ □ Borders and Labels
  - □ Places
  - ▶ □ Photos
  - □ Roads
  - ▶ □ 3D Buildings
  - ▶ □ Ocean
  - ▶ □ Weather
  - ▼ ⊟ Gallery
    - □ 360Cities
    - ▶ □ Discovery Networks
    - □ Earthquakes
    - □ Everytrail
    - ▶ □ European Space Agency
    - □ Gigapan Photos
    - □ Gigapxl Photos
    - □ Google Earth Community
    - ▶ □ NASA
    - ▶ □ National Geographic Maga...
    - ▼ ☑ Rumsey Historical Maps
      - ☑ About the Maps
      - ☑ Map Finder
    - ▶ □ Travel and Tourism
    - □ Trimble Outdoors Trips
    - □ Volcanoes
    - □ Webcams.travel
    - □ Wikiloc
    - □ YouTube

You might need to zoom in to see all of the Map icons. Can you find the World Globe from 1812?



Once you click on an icon an information panel pops up. Click on the map thumbnail to see the map tacked onto the digital globe. We will learn to properly georeference maps in 'Georeferencing in QGIS 2.0'.[454]

[454] Jim Clifford, Josh MacFadyen, and Dan Macfarlane 'Georeferencing in QGIS 2.0', *The Programming Historian* (2013).



Canada 1815

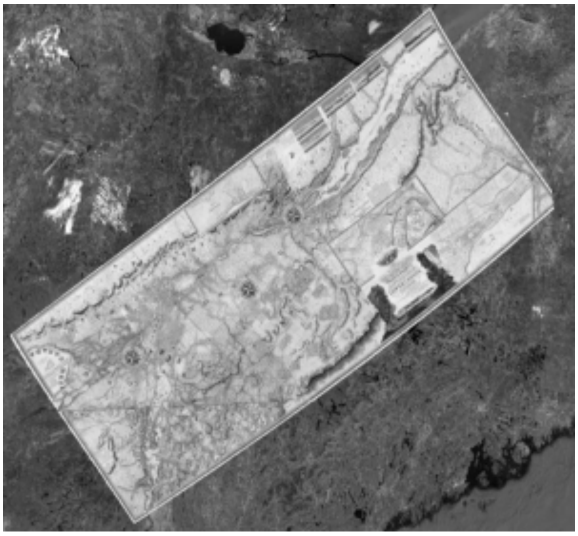
This large ten sheet map is extraordinary - it is over ten feet long when joined and almost five feet tall. It has five views and three large inset maps of Montreal, Quebec, and Three Rivers. The detail and graphic elegance make this map one of the finest Canadian maps of the early 19th century.

To Overlay the Map on Google Earth, Click on Map Thumbnail

Overlay opens in Google Earth -- Places -- Temporary Places.

Download links to all Rumsey Historical Maps

Read more about this Map.



KML: Keyhole Markup Language files

Google developed a file format to save and share map data: KML. This stands for Keyhole Markup Language, and it is a highly portable type of file (i.e. a KML can be used with different types of GIS software) that can store many different types of GIS data, including vector data.

Maps and images you create in Google Maps and Google Earth can be saved as KML files This means you can save work done in Google Maps or Google Earth. With KML files you can transfer data between these two platforms and bring your map data into Quantum GIS or ArcGIS.

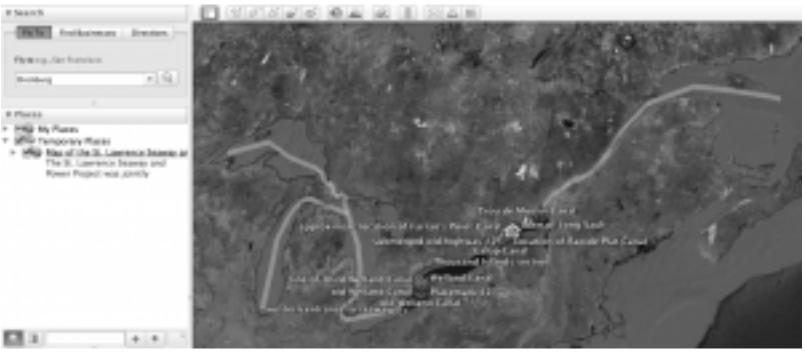
For example, you can import the data you created in Google Maps Engine Lite. If you created a map in the exercise above, it can be found by clicking "Open Map" on the 'Maps Engine Lite' home page.[455] Click on the folder icon on the left hand side of the legend beneath the map title and click

[455] 'Welcome to My Maps': https://www.google.com/maps/d/

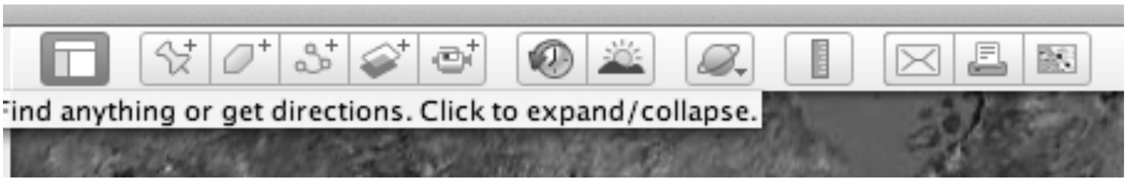
"export to KML". (You can also download and explore Dan Macfarlane's ‘Seaway map’[456] for this part of the exercise).

Bringing your KML file into Google Earth

Download the KML file from Google Maps Engine Lite (as described above).
Double click on the KML file in your Download folder.
Find the data in the Temporary Folder in Google Earth.



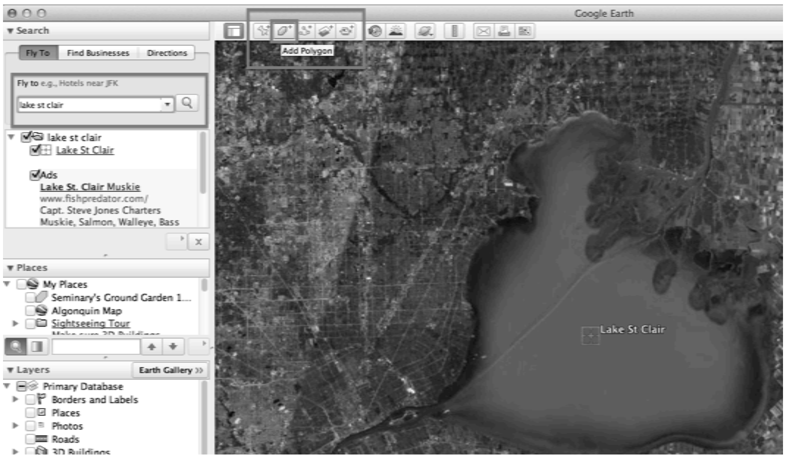
You can now explore these map features in 3D, or you can add new lines, points and polygons using the various icons along the top left of your Google Earth window (see image below). This works in essentially the same way as it did for Google Maps, although there is more functionality and options. In Dan's Seaway map, the old canals and current Seaway route were traced in different line colours and widths using the line feature (this was made possible by overlaying historical maps, which is described below), while various features were marked off with appropriate Placemarks. For those so inclined, there is also the option of recording a tour that could be useful for presentations or teaching purposes (when the "record a tour" icon is selected, recording options will show up on the bottom left of the window).



ind anything or get directions. Click to expand/collapse.

Try adding a new feature to Dan's Seaway data. We've created a polygon (in GIS terminology a polygon is a closed shape of any type – a circle, hexagon, and square are all examples) of Lake St. Clair in the next image. Find Lake St. Clair (east of Detroit) and try adding a polygon.

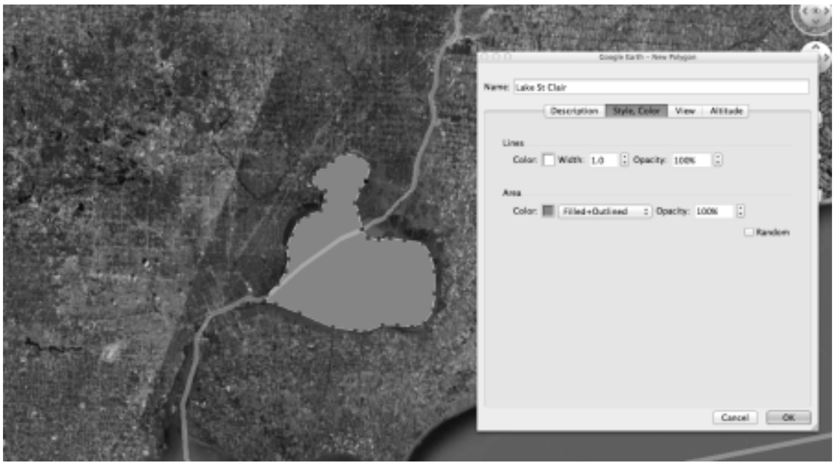
[456] ‘Seaway Map’: https://www.google.com/maps/d/kml?mid=zLGDOilNtMgo.kTGA5fuMJTds&ie=UTF8&hl=en&t=h&source=embed&authuser=0&msa=0&output=kml



Fly to e.g., Hotels near JFK: lake st clair

lake st clair
Lake St Clair

Ads
Lake St. Clair Muskie
www.fishpredator.com/
Capt. Steve Jones Charters
Muskie, Salmon, Walleye, Bass

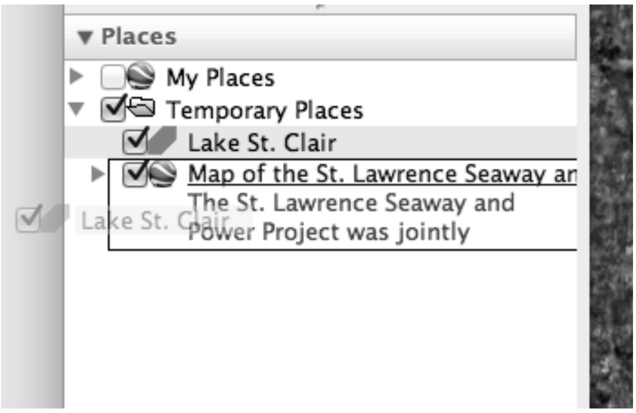


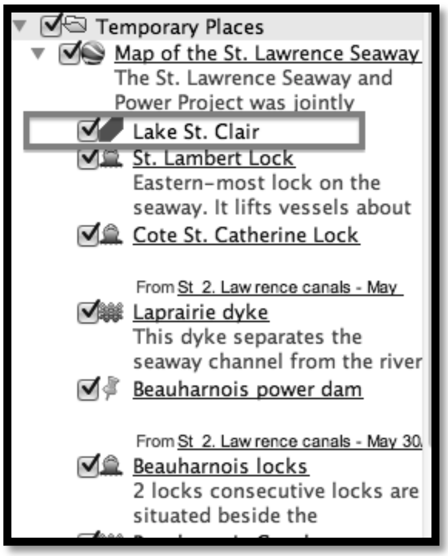
Name: Lake St Clair

Lines
Color: □ Width: 1.0 Opacity: 100%

Area
Color: ■ Filled+Outlined Opacity: 100%

Label the new feature Lake St. Clair. You can then drag the new feature onto Dan's Seaway data and add it to the collection. You can then save this expanded version of the Seaway map as a KML to share via email, upload into Google Maps, or to export this data into QGIS. Find the save option by right-clicking on the Seaway collection and choose Save Place As or Email.


Places
My Places
Temporary Places
Lake St. Clair
Map of the St. Lawrence Seaway an
The St. Lawrence Seaway and
Power Project was jointly
Lake St. Clair



☑ Temporary Places

☑ Map of the St. Lawrence Seaway
The St. Lawrence Seaway and Power Project was jointly

☑ Lake St. Clair

☑ St. Lambert Lock
Eastern-most lock on the seaway. It lifts vessels about

☑ Cote St. Catherine Lock

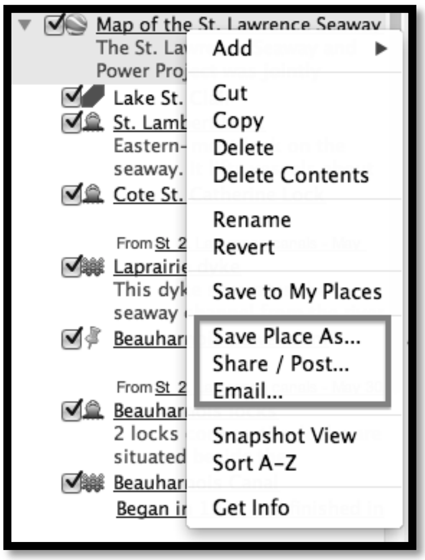
From St 2. Law rence canals - May

☑ Laprairie dyke
This dyke separates the seaway channel from the river

☑ Beauharnois power dam

From St 2. Law rence canals - May 30,

☑ Beauharnois locks
2 locks consecutive locks are situated beside the

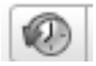


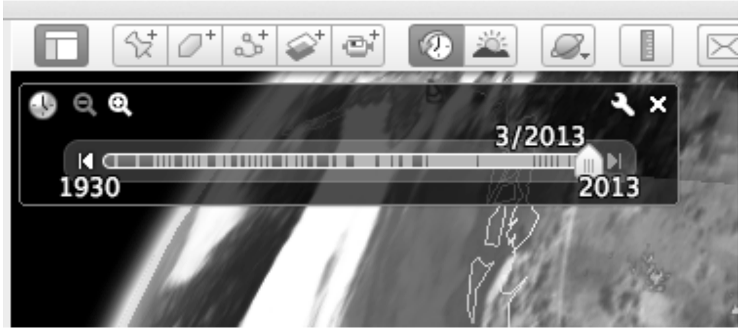
Adding Scanned Historical Maps

Within Google Earth, you can upload a digital copy of a historical map. This could be a map that has been scanned, or an image obtained that is already in a digital format (for tips on finding historical maps online see: Mobile Mapping and Historical GIS in the Field). The main purpose for uploading a digital map, from a historical perspective, is to place it over top of a Google Earth image in the browser. This is known as an overlay. Performing an overlay allows for useful comparisons of change over time.

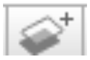
Start by identifying the images you want to use: the image within Google Earth, and the map you want to overlay with. For the map you want to overlay, the file can be in JPEG or TIFF format, but not PDF.

Within Google Earth, identify the area of the map you want to overlay. Note that you can go back in time (i.e. look at older satellite photos) by clicking on the 'Show historical imagery' icon on the top toolbar. and then adjusting the time-scale slider that will appear.

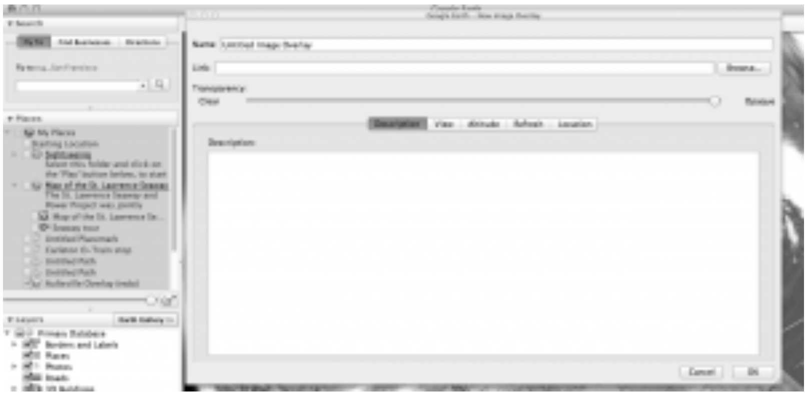




Once you have identified the images you plan to use, click on the 'Add Image Overlay' icon on the top toolbar.



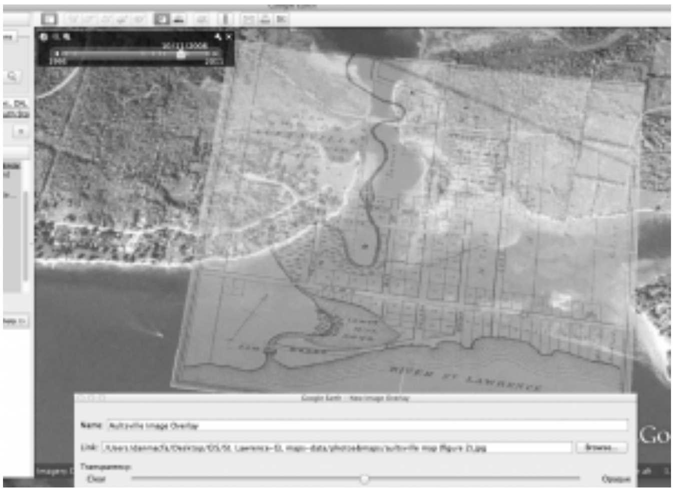
A new window will appear. Begin by giving it a different title if you wish (the default is 'Untitled Image Overlay').



To the right of the Link field, click the Browse button to select from your files the map you wish to be the overlaying image.

Move the New Image Overlay window out of the way (don't close it or click "Cancel" or "OK") so that you can see the Google Earth browser. The map you uploaded will now appear over top of the Google Earth satellite image in the Google Earth browser.

There are fluorescent green markers in the middle and at the edges of the uploaded map. These can be used to stretch, shrink, and move the map so that it aligns properly with the satellite image. This is a simple form of georeferencing (see 'Georeferencing in QGIS 2.0' lesson). The image below shows the above steps using an old map of the town of Aultsville overlaid on top of Google satellite imagery from 2008 in which the remains of the town's roads and building foundations in the St. Lawrence River are visible (Aultsville was one of the Lost Villages flooded out by the St. Lawrence Seaway and Power Project).


Google Earth - New Image Overlay

Name: Aultsville Image Overlay

Link: /Users/danmacfa/Desktop/DS/St. Lawrence-SL maps-data/photos&maps/aultsville map figure 2l.jpg

Back in the New Image Overlay window, note that there are a range of options (Description, View, Altitude, Refresh, Location) that you can select. At this point, you likely don't need to worry about these, although you may wish to add information under the Description tab.

Once you are satisfied with your overlay, in the New Image Overlay window click on OK in the bottom right corner.

You will want to save your work. Under File on your computer's menu bar, you have two options. You can save a copy of the image (File -> Save -> Save Image…) you have created to your computer in jpg format, and you can also save the overlay within Google Earth so that it can be accessed in the future (File -> Save -> Save My Places). The latter is saved as a KML file.

To share KML files simply locate the file you saved to your computer and upload it to your website, social media site, or send it as an email attachment.

You have learned how to use Google Maps and Earth. Make sure you save your work!

About the Authors

Jim Clifford is an assistant professor in the Department of History at the University of Saskatchewan. Josh MacFadyen is a Project Coordinator at the Network in Canadian History & Environment. Daniel Macfarlane is a Visiting Scholar in the School of Canadian Studies at Carleton University.

43. Installing QGIS 2.0 and Adding Layers

Jim Clifford, Joshua MacFadyen, Daniel MacFarlane – 2013

Lesson Goals

In this lesson you will install QGIS software, download geospatial files like shapefiles and GeoTIFFs, and create a map out of a number of vector and raster layers. Quantum or QGIS is an open source alternative to the industry leader, ArcGIS from ESRI. QGIS is multiplatform, which means it runs on Windows, Macs, and Linux and it has many of the functions most commonly used by historians. ArcGIS is prohibitively expensive and only runs on Windows (though software can be purchased to allow it to run on Mac). However, many universities have site licenses, meaning students and employees have access to free copies of the software (try contacting your map librarian, computer services, or the geography department). QGIS is ideal for those without access to a free copy of Arc and it is also a good option for learning basic GIS skills and deciding if you want to install a copy of ArcGIS on your machine. Moreover, any work you do in QGIS can be exported to ArcGIS at a later date if you decide to upgrade. The authors tend to use both and are happy to run QGIS on Mac and Linux computers for basic tasks, but still return to ArcGIS for more advanced work. In many cases it is not lack of functions, but stability issues that bring us back to ArcGIS. For those who are learning Python with the Programming Historian, you will be glad to know that both QGIS and ArcGIS use Python as their main scripting language.

Installing QGIS

Navigate to the QGIS Download page.[457] The procedure is a little different depending on your operating system. Click on the appropriate Operating System. Follow the instructions below.

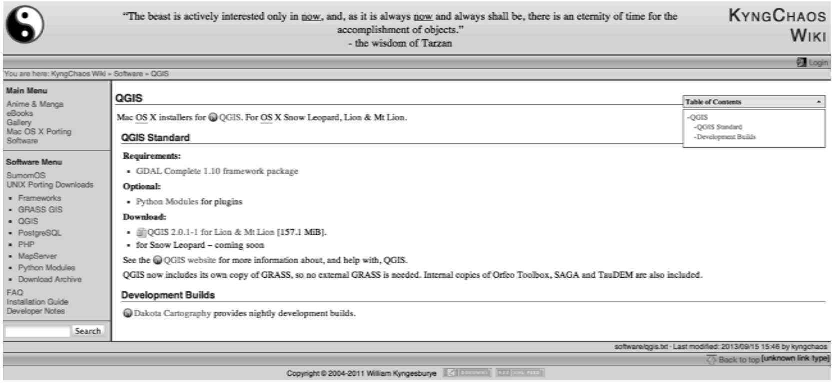
Mac Instructions

For most people it will be best to choose Master release (the one that has a single installer package). You will still need to install other software packages before installing QGIS. Under 4.2, click on the link (KyngChaos Qgis download page)[458] and download the following two files (see screen shot below): 1) GDAL complete 1.10 framework package (under Requirements) and 2) QGIS 2.0.1 (under Download) for your respective Mac OS (this works with Lion, Mountain Lion, and Snow Leopard – no

[457] 'QGIS': http://hub.qgis.org/projects/quantum-gis/wiki/Download

[458] 'QGIS', *KychChaos Wiki*: http://www.kyngchaos.com/software/qgis

word yet on using it with the forthcoming Mavericks). Install these like any other Mac programs.



"The beast is actively interested only in now, and, as it is always now and always shall be, there is an eternity of time for the accomplishment of objects."
- the wisdom of Tarzan

You are here: KyngChaos Wiki » Software » QGIS

QGIS

Mac OS X installers for QGIS. For OS X Snow Leopard, Lion & Mt Lion.

QGIS Standard

Requirements:

- GDAL Complete 1.10 framework package

Optional:

- Python Modules for plugins

Download:

- QGIS 2.0.1-1 for Lion & Mt Lion [157.1 MiB].
- for Snow Leopard – coming soon

See the QGIS website for more information about, and help with, QGIS.

QGIS now includes its own copy of GRASS, so no external GRASS is needed. Internal copies of Orfeo Toolbox, SAGA and TauDEM are also included.

Development Builds

Dakota Cartography provides nightly development builds.

software/qgis.txt · Last modified: 2013/09/15 15:46 by kyngchaos

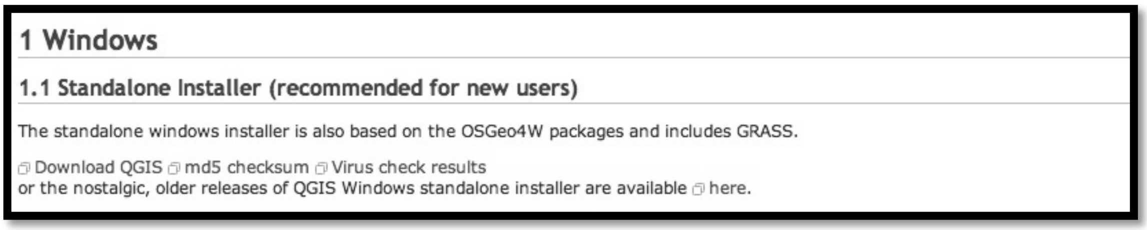
Copyright © 2004-2011 William Kyngesburye

once the frameworks are installed, download and install QGIS.

as with any other Mac application you are using for the first time, you will have to go find the QGIS application in Applications

Windows Instructions

under Standalone Installer, click on the link to Download QGIS



1 Windows

1.1 Standalone Installer (recommended for new users)

The standalone windows installer is also based on the OSGeo4W packages and includes GRASS.

Download QGIS md5 checksum Virus check results
or the nostalgic, older releases of QGIS Windows standalone installer are available here.

double-click on the `.exe` file to execute

QGIS is very simple to install in most versions of Linux. Follow the instructions on the download page.

Prince Edward Island Data

We will be using some government data from the Canadian province of Prince Edward Island. PEI is a great example because there is a lot of data for free online and because it is Canada's smallest province, making the downloads quick! Download PEI shapefiles:

Navigate to the links below in your web browser, read/accept the license agreement, and then download the following (they will ask for your name and email with each download). We created the final two shapefiles, so they should download directly:

http://www.gov.pe.ca/gis/license_agreement.php3?name=coastline&file_format=SHP

http://www.gov.pe.ca/gis/license_agreement.php3?name=lot_town&file_format=SHP

http://www.gov.pe.ca/gis/license_agreement.php3?name=hydronetwork&file_format=SHP

http://www.gov.pe.ca/gis/license_agreement.php3?name=forest_35&file_format=SHP

http://www.gov.pe.ca/gis/license_agreement.php3?name=nat_parks&file_format=SHP

PEI Highways: http://programminghistorian.org/assets/PEI_highway.zip

PEI Places: http://programminghistorian.org/assets/PEI_placenames.zip

After downloading all seven files, move them into a folder and unzip the zipped files. Take a look at the contents of the folders. You will notice four files with the same name, but different file types. When you navigate to these folders from GIS software, you will find that you only need to click on the .shp and that the other three formats support this file in the background. It is important when moving files on your computer to always move and keep all four files together. This is one reason Shapefiles are normally shared using zip compression. Remember the folder in which you save uncompressed shapefile folders, as you will need to find them from within QGIS in a few minutes.

Starting Your GIS Project

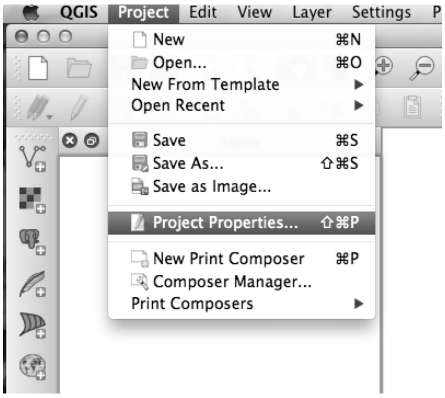
Open QGIS. The first thing we need to do is set up the Coordinate Reference System (CRS) correctly.[459] The CRS is the map projection and projections are the various ways to represent real world places on two-dimensional maps. The default is WGS84 (it is increasingly common to use WGS 84 which is compatible with Google Earth type software), but since most of our data and examples are created by Canadian governments we recommend using NAD 83 (North American Datum, 1983). For more on NAD 83 and the Federal Government's datum, see NRCan's website.[460] PEI has its own NAD 83 coordinate reference system which uses a Double Stereographic projection.[461] Managing the CRS of different layers of information and making sure they are working correctly is one of the most complicated aspects of GIS for beginners. Nonetheless, if the software is setup correctly, it should convert the CRS and allow you to work with data imported from different sources. *Select Project Properties*

Mac: Project–>Project Properties

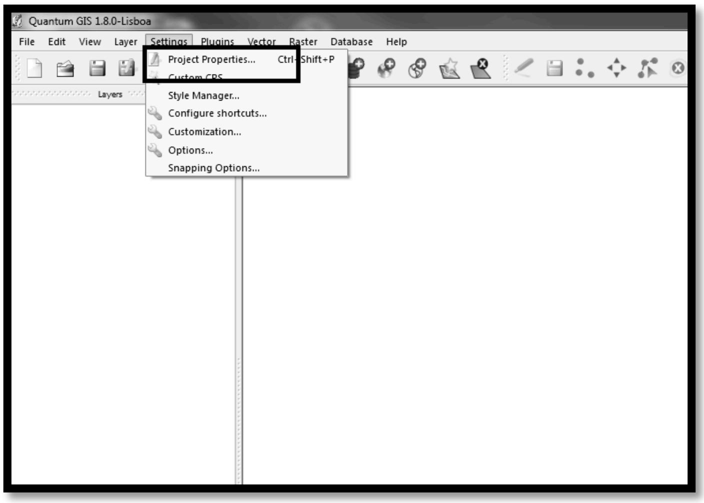
[459] 'Spatial reference system', *Wikipedia*: https://en.wikipedia.org/wiki/Spatial_reference_system

[460] 'Map Datum': http://www.nrcan.gc.ca/earth-sciences/geography/topographic-information/maps/9791

[461] 'PEI Coordinate System': http://www.gov.pe.ca/gis/index.php3?number=77865&lang=E



Windows: Settings-> Project Properties

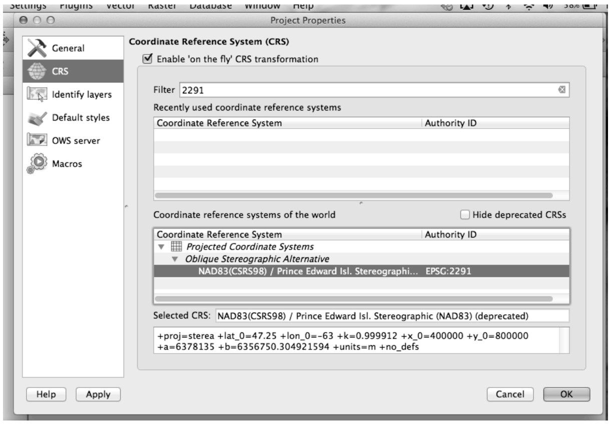


In the left window pane select CRS (second from the top)

- click Enable 'on the fly' CRS transformation button on top left

in the Filter box enter '2291' – this quickly navigates to the best Coordinate reference system for Prince Edward Island.

- under the box titled Coordinate reference systems of the world, select 'NAD83(CSRS98) / Prince Edward Isl. (Stereographic)' and hit OK


Project Properties

Coordinate Reference System (CRS)

☑ Enable 'on the fly' CRS transformation

Filter 2291

Recently used coordinate reference systems

Coordinate Reference System	Authority ID

Coordinate reference systems of the world ☐ Hide deprecated CRSs

Coordinate Reference System	Authority ID
▼ *Projected Coordinate Systems*	
▼ *Oblique Stereographic Alternative*	
NAD83(CSRS98) / Prince Edward Isl. Stereographi...	EPSG:2291

Selected CRS: NAD83(CSRS98) / Prince Edward Isl. Stereographic (NAD83) (deprecated)

```
+proj=sterea +lat_0=47.25 +lon_0=-63 +k=0.999912 +x_0=400000 +y_0=800000
+a=6378135 +b=6356750.304921594 +units=m +no_defs
```

notice the projection has changed in the bottom right corner of the QGIS window. Next to that you will see the geographic location of your mouse pointer in metres under Project menu, select Save Project (you should save your project after each step)

You are now set up to work on the tutorial project, but might have a few questions about what CRS to use for your own project. WGS83 might work in the short term, particularly if you are working on a fairly large scale, but it will be difficult to accurately work on local maps. One hint is to learn what CRS or Projections is used for paper maps of the region. If you are scanning a good quality paper map to use as the base layer it might be a good idea to use the same projection. You can also try searching the internet for the more common CRS for a particular region. For those of you working on North American projects identifying the correct NAD83 for your region will often be the best CRS. Here are a few links to other resources that will help you choose a CRS for your own project: Tutorial: Working with Projections in QGIS.[462]

Building a Base Map

Now that your computer is driving with the right directions, it's time to add some information that makes sense to humans. Your project should start with a base map, or a selection of geospatial information that lets your readers recognize real world features on the map. For most users this will be comprised of several 'layers' of vector and raster data, which can be rearranged, coloured, and labeled in such a way that they make sense to your readers and your project's objectives. A relatively new feature on many GIS programs is the availability of pre-fab base maps, but since this technology is under development for open source platforms like QGIS we will walk through the process of creating our own base map by adding vector and raster layers in this module. For those who would like to add pre-fab base maps to QGIS, you can try installing the 'OpenLayers' Plugin under Plugins->Manage and Install Plugins. Select "Get More" on the left. Click OpenLayers and then click Install plugin. Click OK and then click close. Once installed, you'll find OpenLayers in the Plugins Menu. Try installing some of the different Google and OpenStreetMaps layers. At the time of writing this module, the OpenLayers plugin (v. 1.1.1) installs but fails to work properly on any Mac using OSX. It appears to work more consistently on QGIS running on Windows 7. Give it a try, as we expect it will only get better in the months ahead. Note, however, that the projection for some of these global maps do not correct on the fly, so the satellite images might not alway sync up with data projected in a different CRS.

Opening Vectors

Vectors defined:[463] GIS uses points, lines, and polygons, also known as **vector** data. Its first order of work is to arrange these points, lines, and

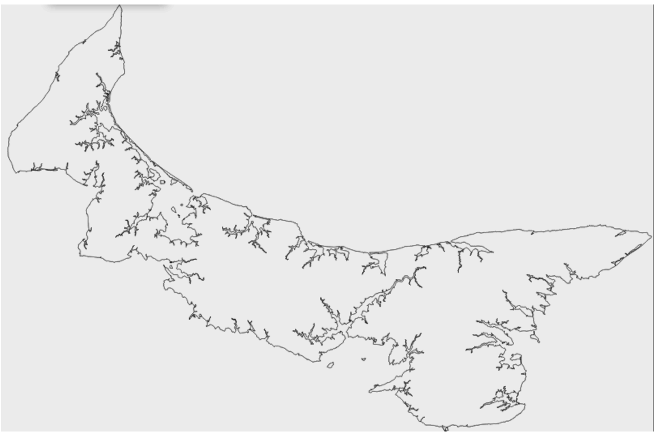
[462] 'Tutorial: Working with Projections in QGIS': http://qgis.spatialthoughts.com/2012/04/tutorial-working-with-projections-in.html

[463] 'GIS Data Explored – Vector and Raster Data': https://www.gislounge.com/geodatabases-explored-vector-and-raster-data/

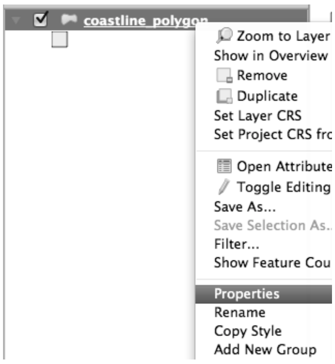
polygons and project them accurately on maps. Points may be towns or telephone poles; lines could represent rivers, roads, or railroads; and polygons could encompass a farmer's lot or larger political boundaries. However, it is also possible to attach historical data to these geographical places and study how people interacted with and changed their physical environments. The population of towns changed, rivers moved their courses, lots were subdivided, and land was planted with various crops.

under Layer on toolbar, choose Add Vector Layer (alternatively the same icon you see next to 'Add Vector Layer' can also be selected from the tool bar on the upper left side)

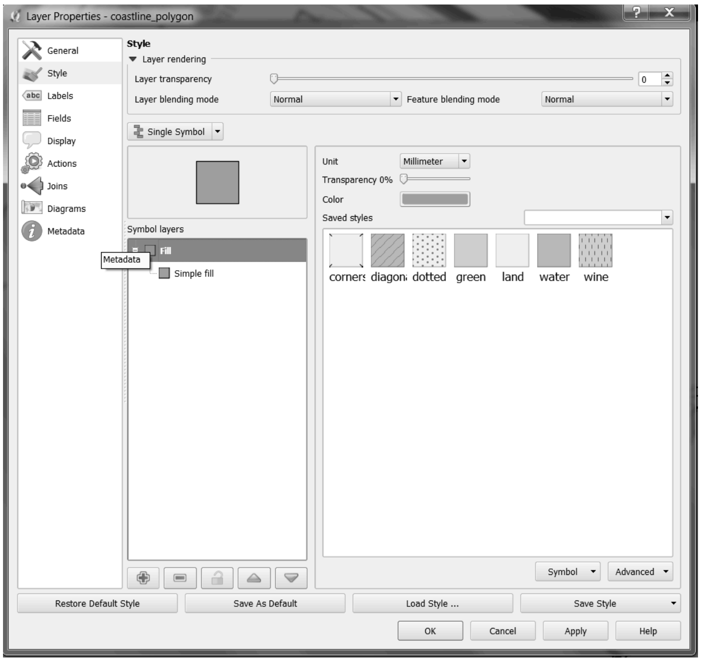
click Browse, find your downloaded Prince Edward Island shapefiles in the folder
open the coastline_polygon folder



select coastline_polygon.shp, then select 'OK', and you should see the island's coastline on your screen. Sometimes QGIS adds a coloured background (see the image above). If you have a coloured background, follow the steps below. If not, skip down the page to the ***.
right click the layer (coastline_polygon) in the Layers menu and choose Properties.



- In the ensuing window, click Style in the left pane
- There are a range of options, but we want to get rid of the background all together. Click **Simple fill**.



Layer Properties - coastline_polygon

Style

Layer rendering

Layer transparency 0

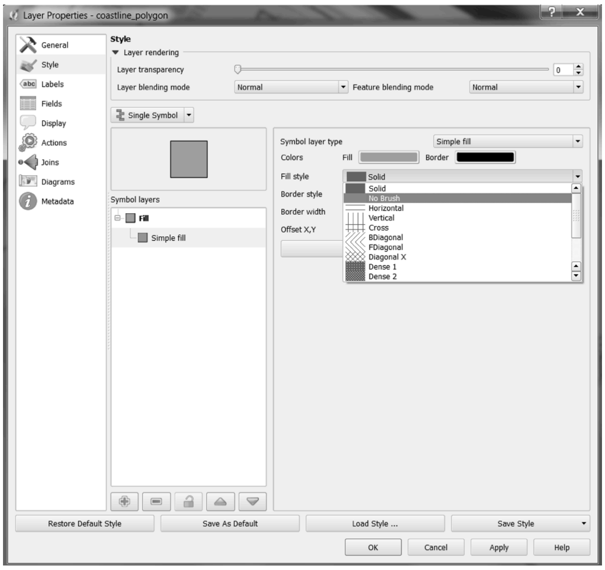
Layer blending mode Normal

Feature blending mode Normal

Unit Millimeter

Transparency 0%

Then choose '**No Brush**' in the **Fill style** drop down menu. **Click OK.**



Layer Properties - coastline_polygon

Style

Layer rendering

Layer transparency 0

Layer blending mode Normal

Feature blending mode Normal

Symbol layer type Simple fill

Fill style Solid

Choose Add Vector Layer again.

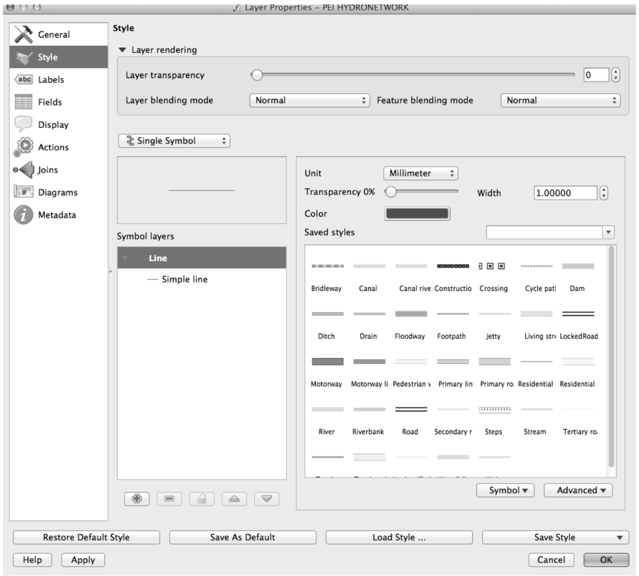
click Browse, find your downloaded Prince Edward Island shapefiles in the folder

select 'PEI_HYDRONETWORK'

click on 'PEI_HYDRONETWORK.shp' and then hit 'Open'

right click the layer in the Layers menu and choose Properties.

select Style tab, and choose an appropriate blue to color the hydronetwork and select 'OK' at the bottom right of the window



Layer Properties – PEI HYDRONETWORK

Style

Layer rendering

Layer transparency: 0

Layer blending mode: Normal

Feature blending mode: Normal

Single Symbol

Unit: Millimeter

Transparency 0%

Width: 1.00000

Symbol layers: Line — Simple line

Your map should now look like this:

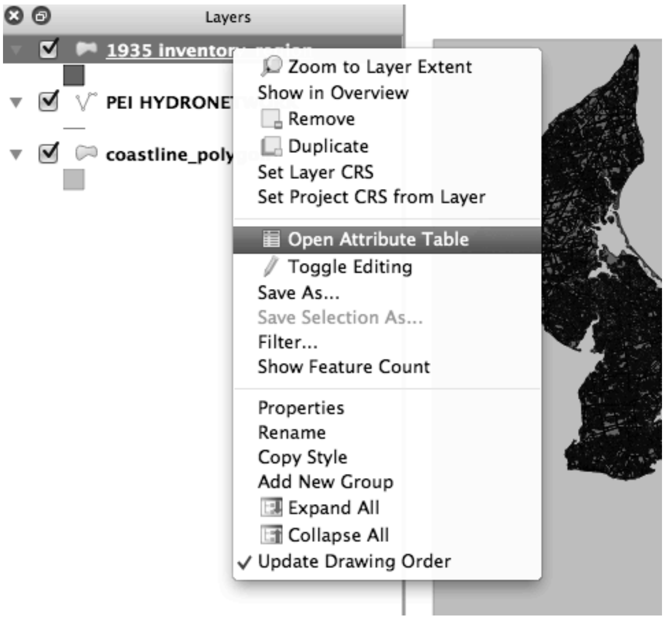
Choose Add Vector Layer again.

click Browse, find your downloaded Prince Edward Island shapefiles in the folder

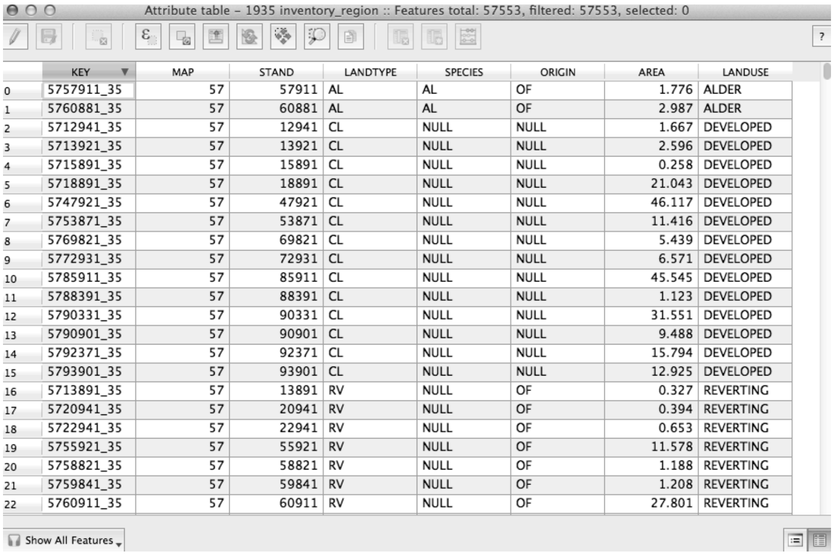
double-click on '1935 inventory_region.shp' and then hit 'Open'

This will add a dense map showing the different forest cover in 1935. However, to see the different categories, you will need to change the symbology to represent the different categories of forest with different colours. We will need to know which column of the database tables includes the forest category information, so the first step is to open and inspect the attribute table.

right click on the 1935_inventory_region layer in the Layers window on the left and click on Open Attribute Table



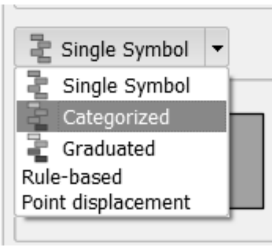
An Attribute Table will open. It has a number of categories and identifiers. Of particular interest is the LANDUSE category which provides information on the forest cover in 1935. We will now show you how to display these categories on the map.


Attribute table – 1935 inventory_region :: Features total: 57553, filtered: 57553, selected: 0

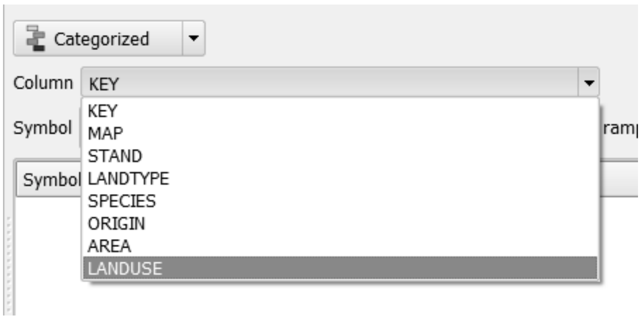
	KEY	MAP	STAND	LANDTYPE	SPECIES	ORIGIN	AREA	LANDUSE
0	5757911_35	57	57911	AL	AL	OF	1.776	ALDER
1	5760881_35	57	60881	AL	AL	OF	2.987	ALDER
2	5712941_35	57	12941	CL	NULL	NULL	1.667	DEVELOPED
3	5713921_35	57	13921	CL	NULL	NULL	2.596	DEVELOPED
4	5715891_35	57	15891	CL	NULL	NULL	0.258	DEVELOPED
5	5718891_35	57	18891	CL	NULL	NULL	21.043	DEVELOPED
6	5747921_35	57	47921	CL	NULL	NULL	46.117	DEVELOPED
7	5753871_35	57	53871	CL	NULL	NULL	11.416	DEVELOPED
8	5769821_35	57	69821	CL	NULL	NULL	5.439	DEVELOPED
9	5772931_35	57	72931	CL	NULL	NULL	6.571	DEVELOPED
10	5785911_35	57	85911	CL	NULL	NULL	45.545	DEVELOPED
11	5788391_35	57	88391	CL	NULL	NULL	1.123	DEVELOPED
12	5790331_35	57	90331	CL	NULL	NULL	31.551	DEVELOPED
13	5790901_35	57	90901	CL	NULL	NULL	9.488	DEVELOPED
14	5792371_35	57	92371	CL	NULL	NULL	15.794	DEVELOPED
15	5793901_35	57	93901	CL	NULL	NULL	12.925	DEVELOPED
16	5713891_35	57	13891	RV	NULL	OF	0.327	REVERTING
17	5720941_35	57	20941	RV	NULL	OF	0.394	REVERTING
18	5722941_35	57	22941	RV	NULL	OF	0.653	REVERTING
19	5755921_35	57	55921	RV	NULL	OF	11.578	REVERTING
20	5758821_35	57	58821	RV	NULL	OF	1.188	REVERTING
21	5759841_35	57	59841	RV	NULL	OF	1.208	REVERTING
22	5760911_35	57	60911	RV	NULL	OF	27.801	REVERTING

Close the Attribute Table, and right click on the 1935_inventory_region layer again and this time choose Properties (alternatively, the shortcut is to double click on the 1935_inventory_region layer).

click Style along the left



on the menu bar that reads 'Single Symbol' select 'Categorized'


Categorized

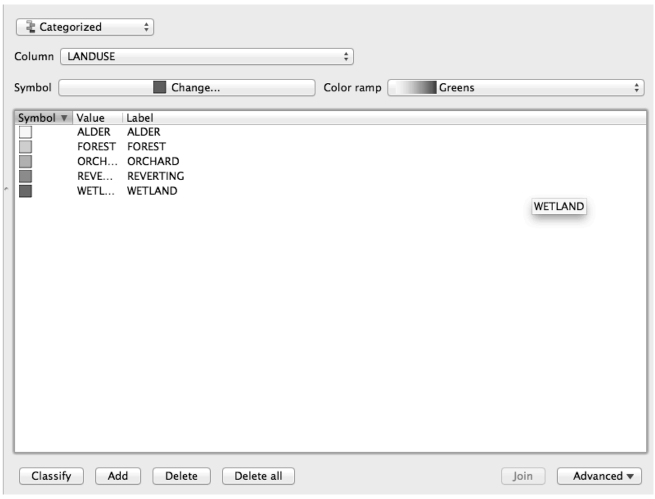
Column: KEY

beside Column choose 'Landuse'

under Color ramp choose Greens

click 'Classify' below and to the left

in Symbol Column, choose the furthest down dark green square (with no value beside it) and hit the 'Delete' button (to the right of Classify); also delete the Developed category, as we want to highlight forested areas. Click 'OK'


Categorized

Column: LANDUSE

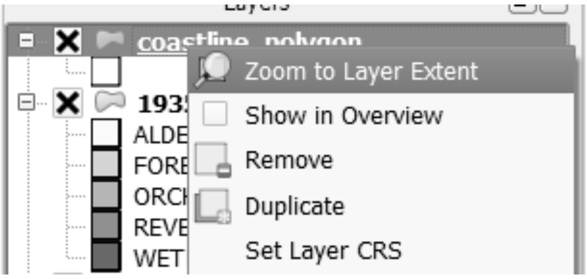
Symbol: Change...

Color ramp: Greens

Symbol	Value	Label
□	ALDER	ALDER
■	FOREST	FOREST
■	ORCH...	ORCHARD
■	REVE...	REVERTING
■	WETL...	WETLAND

- in Layers sidebar menu, click on the little arrow beside 1935_inventory_region to view the legend.
- You can now see the extent of the forests in 1935. Try using the magnifying glass tool to zoom in and inspect the different landuses.

To get back to the full island, right click on any of the layers and choose '**Zoom to Layer Extent.**'



Next, we will add a layer of roads.

under Layer on toolbar, choose Add Vector Layer

click Browse, find your downloaded Prince Edward Island shapefiles in the folder

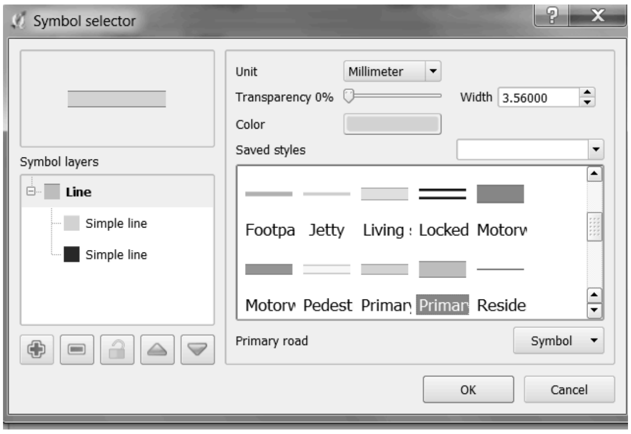
select 'PEI_highway.shp'

in the Layers menu on the left, double-click 'PEI_highway_ship' and select Style from the menu on the left (if it isn't already selected)

click on 'Single Symbol' on top left and select 'Categorized'

beside Column choose 'TYPE'

click Classify



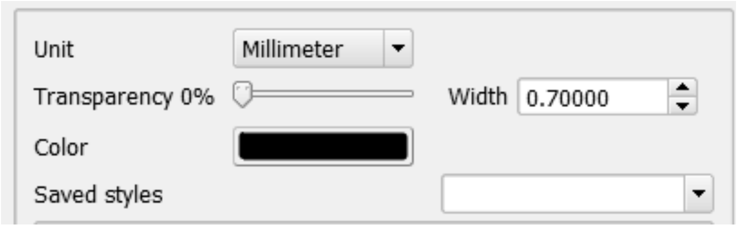
Unit Millimeter

Transparency 0%

Width 3.56000

Primary road

- in the Symbol column, double-click beside 'primary' – in the ensuing window, there is a box with different symbols. Scroll down and find 'primary road'.
- You are back in the Style window. Repeat for the item that called 'primary_link' in the Label column.



Unit Millimeter

Transparency 0%

Width 0.70000

Color

Saved styles

click Symbol beside secondary and change color to black and width to 0.7

repeat for secondary link

click OK. You will now have the highways and other major roads represented on the map

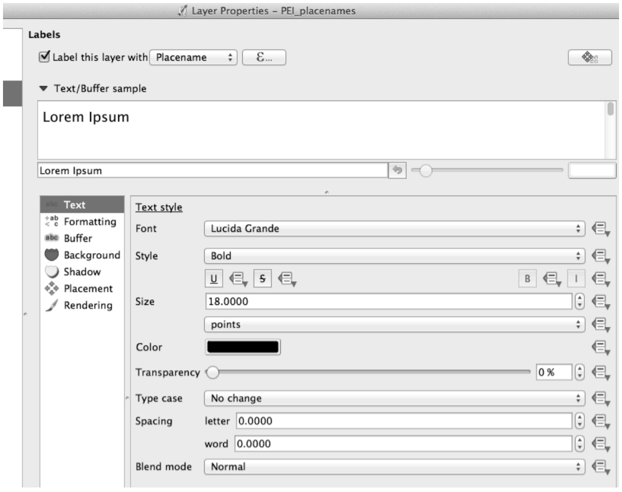
under Layer on toolbar, choose Add Vector Layer

click Browse, find your downloaded Prince Edward Island shapefiles in the folder

select 'PEI_placenames_shp'

double click on 'PEI_placenames' and select 'Open'

in the Layers window, double-click on the PEI_placenames layer. Choose Labels tab along the left (under Style). At the top, select the box beside 'Label this layer with' and in the dropdown box beside that select 'Placename'



Layer Properties – PEI_placenames

Labels

☑ Label this layer with Placename

Text/Buffer sample

Lorem Ipsum

Text style

Font: Lucida Grande

Style: Bold

Size: 18.0000 points

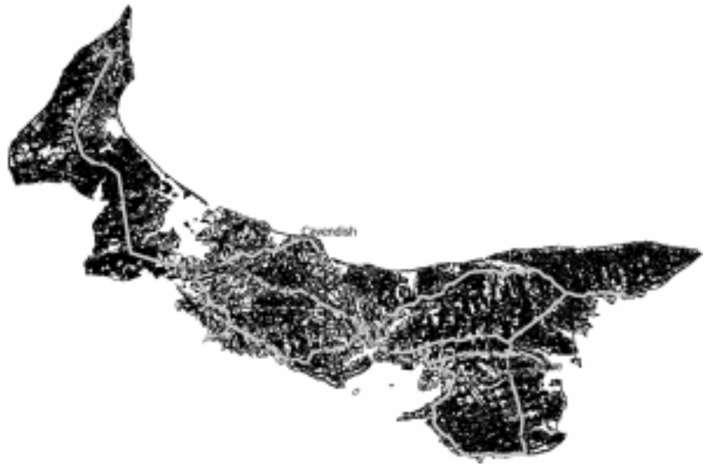
Transparency: 0 %

Type case: No change

Spacing: letter 0.0000, word 0.0000

Blend mode: Normal

Change Font size to '18′
Click 'OK' and examine the results on the map

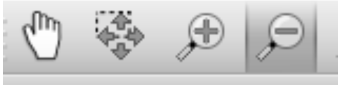


Labelling is where QGIS falls well short of real cartography – it will take tinkering to adjust settings to display the detail desired for a presentation. Try going back to the Labels tab and changing the different settings to see how symbols and displays change.

Note that in the Layers menu you can add and remove the various layers we've added to the map much the same way you did in Google Earth. Click on the check boxes to remove and add the various layers. Drag and drop layers to change the the order they appear. Dragging a layer to the top will place it above the rest of the layers and make it the most prominent. For example, if you drag 'coastline_polygon' to the top, you have a simplified outline of the province along with place names.

Cavendish
Summerside
Charlottetown
Georgetown

Along the toolbar on the top left of the main window are icons that allow you to explore the map. The hand symbol, for example, allows you to click on the map and move it around, while the magnifying glass symbols with plus and minus on them allow you to zoom in and out. Play with these and familiarize yourself with the various functions



having created a map using vector layers, we will now add or use our first raster layer. This is a good time to save your work.

Opening Rasters: **Raster** data are digital images made up of grids. All remote sensing data such as satellite images or aerial photos[464] are rasters, but usually you can't see the grids in these images because they are made up of tiny pixels. Each pixel has its own value and when those values are symbolized in colour or greyscale they make up an image that is useful for display or topographical analysis. A scanned historical map is also brought into GIS in raster format.

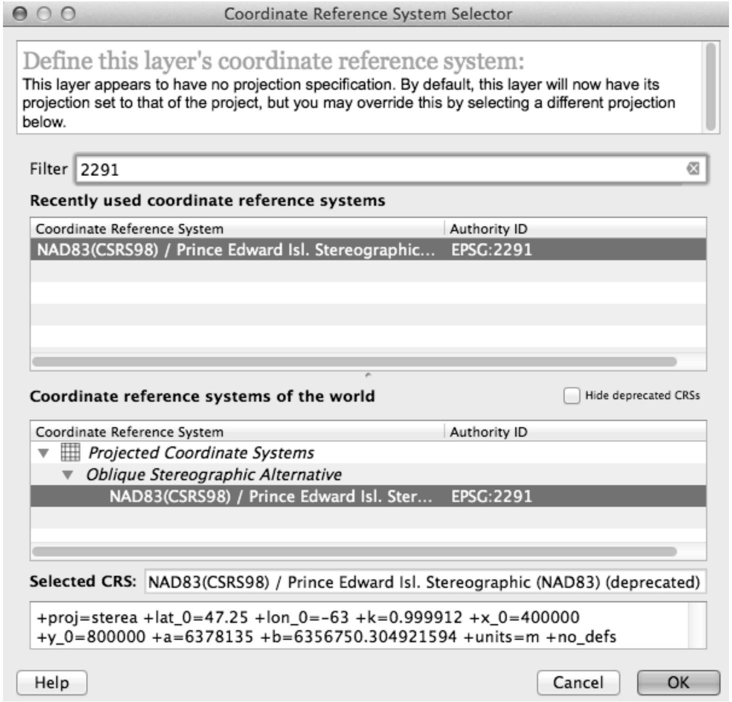
download: 'http://programminghistorian.org/ /assets/PEI_CumminsMap1927_compLZW.tif' to your project folder.

under Layer on toolbar, choose Add Raster Layer (alternatively the same icon you see next to 'Add Raster Layer' can also be selected from the tool bar along the left side of the window)

find the file you have downloaded titled 'PEI_CumminsMap1927.tif'

you will be prompted to define this layer's coordinate system. In the Filter box search for '2291', then in the box below select 'NAD83(CSRS98) / Prince Edward Isl. (Stereographic)…'

[464] 'Orthophoto', *Wikipedia*: https://en.wikipedia.org/wiki/Orthophoto



Define this layer's coordinate reference system:

This layer appears to have no projection specification. By default, this layer will now have its projection set to that of the project, but you may override this by selecting a different projection below.

Filter: 2291

Recently used coordinate reference systems

Coordinate Reference System	Authority ID
NAD83(CSRS98) / Prince Edward Isl. Stereographic...	EPSG:2291

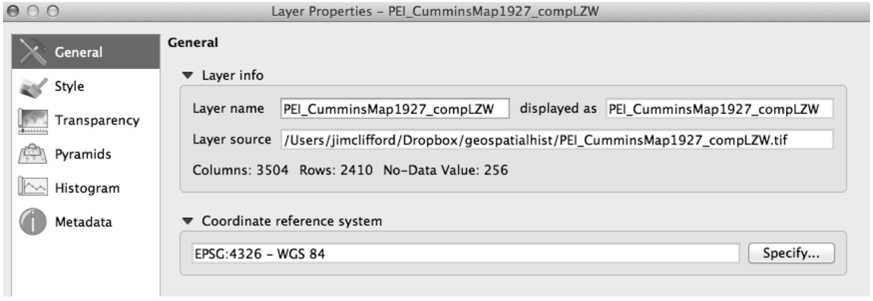
Coordinate reference systems of the world ☐ Hide deprecated CRSs

Coordinate Reference System	Authority ID
▼ Projected Coordinate Systems	
▼ Oblique Stereographic Alternative	
NAD83(CSRS98) / Prince Edward Isl. Ster...	EPSG:2291

Selected CRS: NAD83(CSRS98) / Prince Edward Isl. Stereographic (NAD83) (deprecated)

```
+proj=sterea +lat_0=47.25 +lon_0=-63 +k=0.999912 +x_0=400000
+y_0=800000 +a=6378135 +b=6356750.304921594 +units=m +no_defs
```

If the program does not prompt you for the CRS you need to change it yourself. Double click the PEI_CummingMap1927_compLZW layer and choose '**General**' from the menu on the left. Click '**Specify...**' beside the box showing the incorrect Coordinate reference system. Then follow the instructions above (choose 2291).



General

▼ Layer info

Layer name: PEI_CumminsMap1927_compLZW displayed as: PEI_CumminsMap1927_compLZW

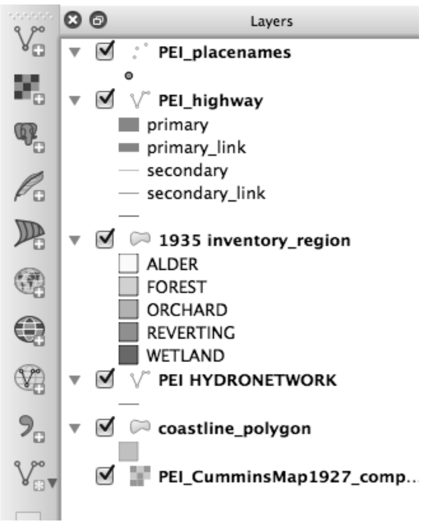
Layer source: /Users/jimclifford/Dropbox/geospatialhist/PEI_CumminsMap1927_compLZW.tif

Columns: 3504 Rows: 2410 No-Data Value: 256

▼ Coordinate reference system

EPSG:4326 – WGS 84

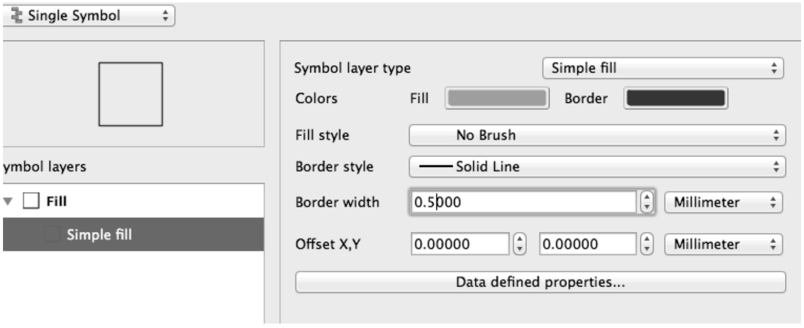
In the Layers window, the map should appear below the vector data. Move it to the bottom of the menu if necessary:



Layers

- ☑ PEI_placenames
- ☑ PEI_highway
  - primary
  - primary_link
  - secondary
  - secondary_link
- ☑ 1935 inventory_region
  - ALDER
  - FOREST
  - ORCHARD
  - REVERTING
  - WETLAND
- ☑ PEI HYDRONETWORK
- ☑ coastline_polygon
- ☑ PEI_CumminsMap1927_comp..

Now we would like to make the coastline more visible, so double-click on 'coastline_polygon' and select 'Style' on the left. In the box under Symbol layers, select 'Simple fill' and options appear in the box to the right. Click on the menu next to 'Border' and make it red, and then beside Border width change it to 0.5, and click OK.



Single Symbol

ymbol layers

- □ Fill
  - Simple fill

Symbol layer type: Simple fill

Colors: Fill, Border

Fill style: No Brush

Border style: Solid Line

Border width: 0.5000 Millimeter

Offset X,Y: 0.00000 0.00000 Millimeter

Data defined properties...

You are now able to see the background raster map through the 'coastline_polygon' layer. Zoom in for closer inspection, and you should be able to see the coastline layer clearly. Notice that the alignment is relatively good, but not perfect. We will learn more in lesson 4 about the challenges of georeferencing historical maps to give them real world coordinates.

You have learned how to install QGIS and add layers. Make sure you save your work!

About the Authors

Jim Clifford is an assistant professor in the Department of History at the University of Saskatchewan. Josh MacFadyen is a Project Coordinator at the Network in Canadian History & Environment. Daniel Macfarlane is a Visiting Scholar in the School of Canadian Studies at Carleton University.

44. Creating New Vector Layers in QGIS 2.0

Jim Clifford, Joshua MacFadyen, Daniel MacFarlane – 2013

Lesson Goals

In this lesson you will learn how to create vector layers based on scanned historical maps. In 'Intro to Google Maps and Google Earth' you used vector layers and created attributes in Google Earth. We will be doing the same thing in this lesson, albeit at a more advanced level, using QGIS software.

Vector layers are, along with raster layers, one of the two basic types of data structures that store data. Vector layers use the three basic GIS features – lines, points, and polygons – to represent real-world features in digital format. Points can be used to represent specific locations, such as towns, buildings, events, etc. (the scale of your map will determine what you represent as a point – in a map of a province, a town would be a point, whereas in a map of a town, a building might be a point). Lines can effectively represent features such as roads, canals, railways, and so on. Polygons (effectively enclosed shapes with more than a few sides) are used to represent more complex objects such as the boundaries of a lake, country, or electoral riding (again, scale will affect your choice – large buildings in a close-up map of a city might be better represented as polygons than as points).

In this lesson you will be creating shapefiles (which are a type of vector data) to represent the historical development of communities and roads in Prince Edward Island. Each shapefile can be created as one of the three types of features: line, point, polygon (though these features can't be mixed within a shapefile) . Each feature you create in a shapefile has a corresponding set of attributes, which are stored in an attribute table. You will create features and learn how to modify them, which involves not only the visual creation of the three types of features, but also the modification of their attributes. To do so, we will use the files from 'Installing QGIS 2.0 and Adding Layers' concerning Prince Edward Island.

Getting Started

Start by downloading the 'PEI_Holland map' to the project folder:

Open the file you saved at the end of 'Installing QGIS 2.0 and Adding Layers'. You should have the following layers in your Layers window:

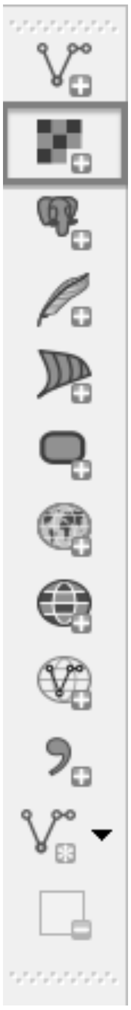
PEI_placenames
PEI_highway
PEI HYDRONETWORK
1935 inventory_region
coastline_polygon

PEI-CumminsMap1927

Uncheck all of these layer except for PEI_placenames, coastline_polygon and PEI_CumminsMap1927

PROVINCE
PRINCE EDWARD ISLAND
CANADA

We are now going to add a second historical map as a raster layer.

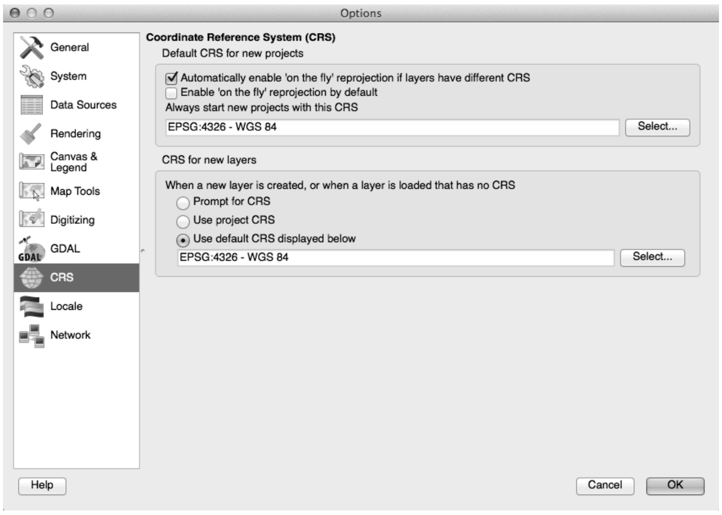


under Layer on toolbar, choose Add Raster Layer (alternatively the same icon you see next to 'Add Raster Layer' can also be selected from tool bar)

find the file you have downloaded titled 'PEI_HollandMap1798'

you will be prompted to define this layer's coordinate system. In the Filter box search for '2291' then in the box below select 'NAD83(CSRS98) / Prince Edward Isl. Stereographic'

If you are not prompted to define the layer's coordinate system, you need to change a setting. Click Settings and then Options. Click CRS on the right hand menu and then choose 'Prompt for CRS' from the options below 'When a new layer is created, or when a layer is loaded that has no CRS'. Click OK. Remove the Holland Map (right click on it and click Remove) and try adding it again. This time you should be prompted for a CRS and you can select the NAD83 option (see above).



Coordinate Reference System (CRS)

Default CRS for new projects

☑ Automatically enable 'on the fly' reprojection if layers have different CRS
□ Enable 'on the fly' reprojection by default

Always start new projects with this CRS: EPSG:4326 - WGS 84 [Select...]

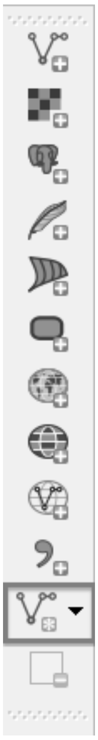
CRS for new layers

When a new layer is created, or when a layer is loaded that has no CRS
○ Prompt for CRS
○ Use project CRS
● Use default CRS displayed below

EPSG:4326 - WGS 84 [Select...]

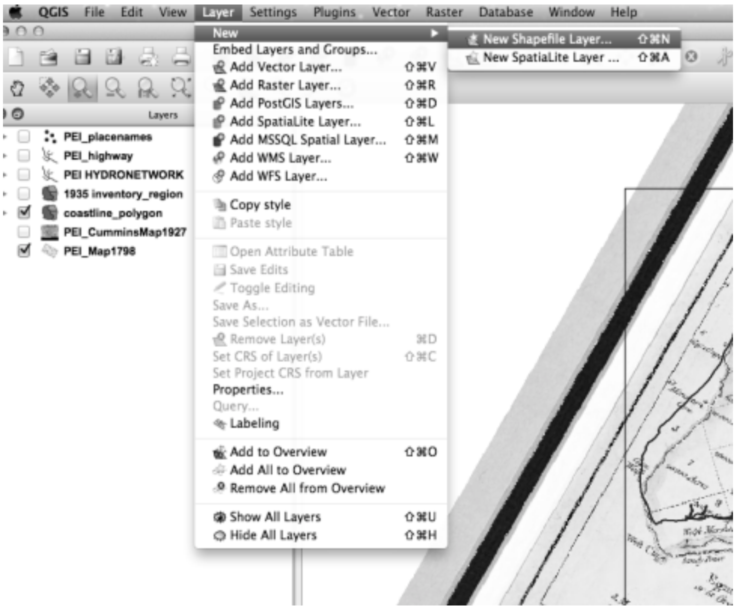
In previous steps you have selected and unselected layers in the Layers window by checking and unchecking the boxes next to them. These layers are organized in descending order of visibility – i.e. the layer at the top is the top layer in your viewer window (provided it is selected). You can drag the layers up and down in the Layer window to change the order in which they will be visible on your viewing window. The coastline_polygon raster layer is currently not visible because it is below the PEI_HollandMap1798 and PEI_Cummins1927 layers. In general it is best to keep vector layers above the raster layers.

Uncheck PEI_Cummins1927 so that the only layer you have remaining is PEI_HollandMap1798. Note that the map appears crooked on the screen; this is because it has already been georeferenced by the lesson writers to match the GIS vector layers. Learn more about georeferencing in ‘Georeferencing in QGIS 2.0’, the next lesson.



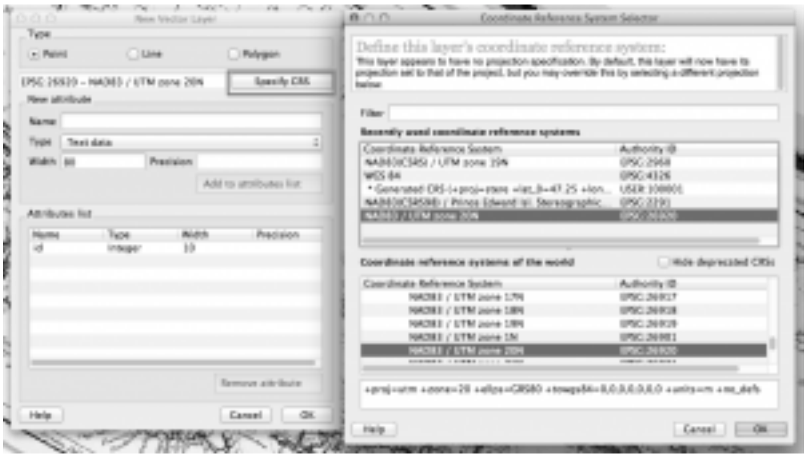
We will now create a point shapefile, which is a vector layer. Click Layer -> New -> New Shapefile Layer

alternatively you can select the New Shapefile Layer icon on the top of the QGIS toolbar window



After selecting New Shapefile Layer, a window titled New Vector Layer appears

In the Type category, Point is already selected for you. Click the Specify CRS button, and select NAD83(CSRS98) / Prince Edward Isl. Stereographic (EPSG: 2291), and then click OK (for information on understanding and selecting UTM zone: http://www.lib.uwaterloo.ca/locations/umd/digital/clump_classes.html)



Returning to the New Vector Layer window, we are going to make some attributes. To create the first attribute:

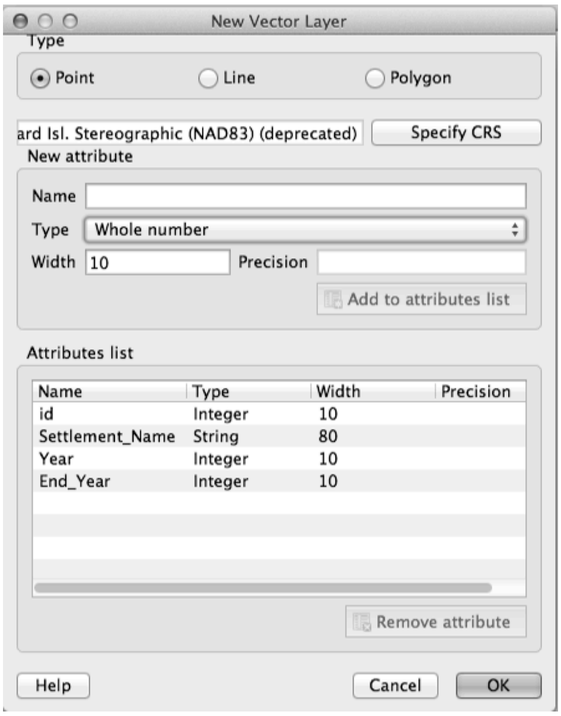
under New attribute, in the field beside Name, type in 'Settlement_Name' (note that when working in databases you cannot use empty spaces in names so the convention is to use underscores in their place) click Add to attributes list

Now we are going to create a second attribute:

under New attribute, in the field beside Name, type in 'Year'
this time, we are going to change the Type to Whole Number
click Add to attribute list

For the third attribute:

under New attribute, in the field beside Name, type in "End_Year" (GIS is not always optimal for dealing with change over time, so in some cases it is important to have a field to identify approximately when something ceased to exist)
change the Type again to Whole Number
click Add to attribute list



● Point ○ Line ○ Polygon

ard Isl. Stereographic (NAD83) (deprecated)

Name

Type: Whole number

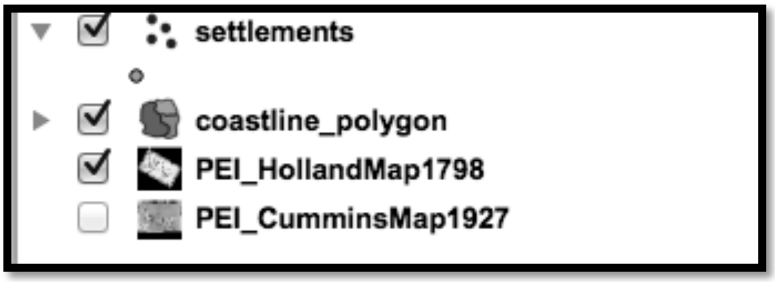
Width: 10

Precision

Name	Type	Width	Precision
id	Integer	10	
Settlement_Name	String	80	
Year	Integer	10	
End_Year	Integer	10	

When you complete these three steps, finish creating this shapefile by clicking OK on the bottom right of the New Vector Layer window. A pops up – name it 'settlements' and save it with your other GIS files.

Note that a layer called 'settlements' now appears in your Layers window. Relocate it above the raster layers.



☑ settlements

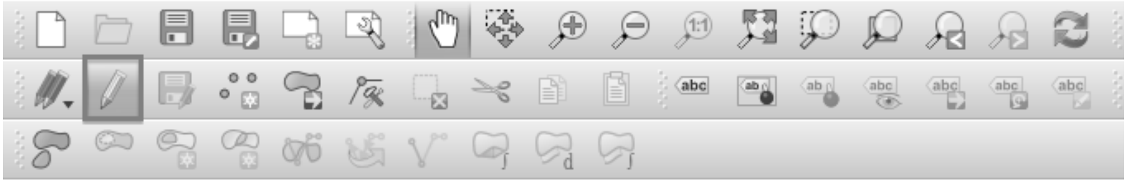
☑ coastline_polygon

☑ PEI_HollandMap1798

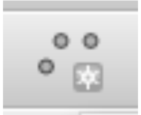
□ PEI_CumminsMap1927

Uncheck all layers except settlements. You will notice that your viewing window is now blank as we have not created any data. We will now create new data from both the `PEI_HollandMap 1798` and the `PEI_CumminsMap1927` to show the increase in settlement between the late 18th and early 20th centuries.

we will begin with the more recent, and thus usually more accurate, map. Reselect (i.e. check the boxes beside) coastline_polygon and PEI_CumminsMap1927
in your viewing window, zoom in to Charlottetown (hint: Charlottetown is near the middle of the island on the south side, at the confluence of three rivers)
select settlements layer in Layers window
on the menu bar, select Toggle Editing

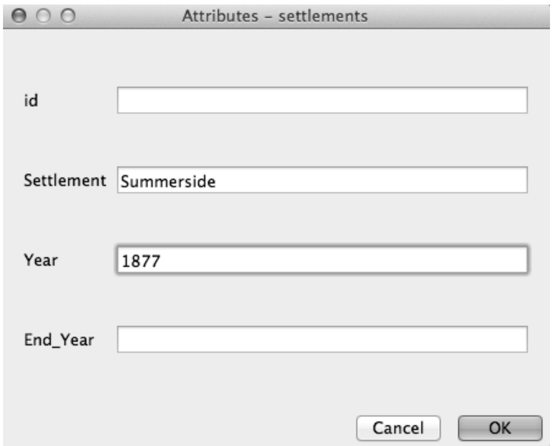


After selecting Toggle Editing, editing buttons will become available to the right along the menu bar. Select the 3 dot feature button.



Your cursor now appears as a crosshair – point the crosshair at Charlottetown (if you don't happen to know PEI's geography, you can cheat by adding on the PEI_placenames layer) keeping it within the modern day coastline, and click (digitization is always a compromise between accuracy and functionality; depending on the quality of the original map and the digitization, for most historical applications extreme accuracy is not necessary).
An Attributes window will appear. Leave id field blank (at time of writing, QGIS appears to be making two id fields and this one is unnecessary). In Settlement field, type in 'Charlottetown'. In the Year field type in 1764. Click OK

We will now repeat the steps we took with Charlottetown for Montague, Summerside, and Cavendish (again, you can find these locations by adding the PEI_placenames layers). Find Montague on the map, select the 3 dot feature button and click on Montague on the map. When the Attributes window appears, input Montague and 1732 in the appropriate fields. Repeat for Summerside (1876) and Cavendish (1790).



id

Settlement Summerside

Year 1877

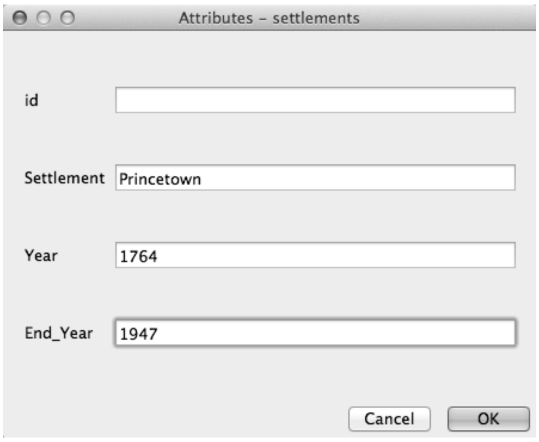
End_Year

In the Layers window, unselect the PEI_CumminsMap1927 and select PEI_HollandMap1798. We are now going to identify two settlements (Princetown & Havre-St-Pierre) that no longer exist.

- To locate Princetown, look for Richmond Bay and Cape Aylebsury (on the north coast to the west of Cavendish), here you will find Princetown (shaded-in) near the boundary between the yellow and the blue

- If you look at the Wikipedia entry for the city[465] you will notice that because of a shallow harbor, Princetown did not become a major settlement. It was renamed in 1947 and later downgraded to a hamlet. For this reason we will include 1947 as the end date for this settlement.

- With the crosshair click on Princetown. In the Attribute table that appears, put Princetown in the Settlement field, put 1764 into the Year field, and put 1947 into the End_Year. Click OK



id

Settlement Princetown

Year 1764

End_Year 1947

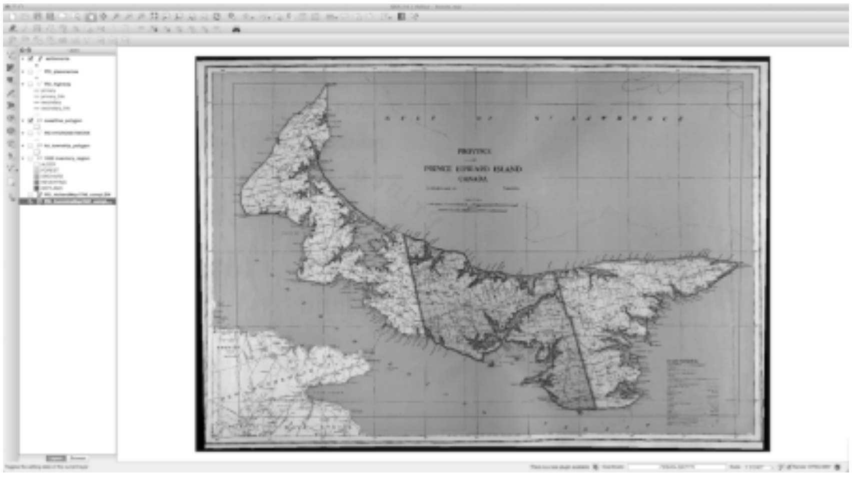
- Click on Save Edits icon on the menu bar (it is between Toggle and Add Feature)

[465] 'Prince Royalty, Prince Edward Island', *Wikipedia*: https://en.wikipedia.org/wiki/Prince_Royalty,_Prince_Edward_Island

- Double-click on settlements layer in the Layers window, choose Labels tab at the top of the ensuing window. Click on the box beside Display labels. In Field containing label select Year (if necessary), change font size to 18.0, change Placement to Above Left, and then click OK

On the northern coast of Lot 39 between Britain's Pond and St. Peters Bay, we will now put a dot for the location of a long lost village called Havre-St-Pierre.

- Havre-St-Pierre was the island's first Acadian settlement but has been uninhabited since the Acadian deportation of 1758.
- With the crosshair click on Havre-St. Pierre. In the Attribute table that appears, put Havre-St-Pierre in the Settlement field, put 1720 into the Year field, and put 1758 into the End_Year. Click OK



We will now now create another vector layer – this layer will be a line vector. Click Layer -> New -> New Shapefile Layer. The New Vector Layer window will appear (in the Type category at the top, select Line)

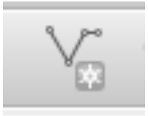
Click the Specify CRS button, and select NAD83(CSRS98) / Prince Edward Isl. Stereographic (EPSG: 2291), and then click OK
under New attribute, in the field beside Name, type in 'Road_Name'
click Add to attributes list

Create a second attribute

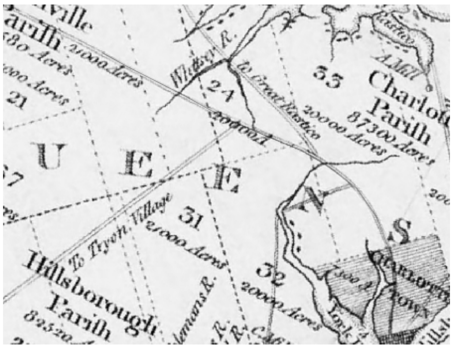
under New attribute, in the field beside Name, type in Year
change the Type to Whole Number
click Add to attribute list
To finish creating this shapefile, click OK on the bottom right of the New Vector Layer window. A 'save' screen pops up – name it 'roads' and save it with your other GIS files.

We are now going to trace the roads from the 1798 map so that we can compare them to the modern roads. Make that you have the PEI_Holland1798 and settlements layers checked in the Layers window.

Select road layer in the layers window, select Toggle Editing on the top toolbar, and then select Add Feature

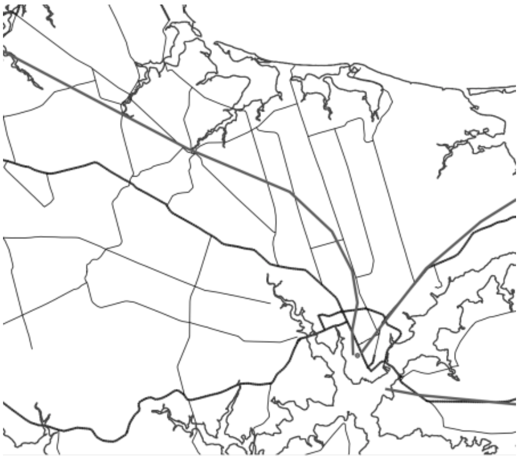


First trace the road from Charlottetown to Princetown. Click on Charlottetown and then click repeatedly at points along the road to Princetown and you will see the line being created. Repeat until you arrive at Princetown, then right-click. In the resulting Attributes – road window, in the Name field enter "to Princetown" and in the Year field enter 1798. Click OK



repeat this step for 3 to 4 more roads found on the PEI_HollandMap1798. click Save Edits and then click Toggle Editing to turn it off

Deselect the PEI_HollandMap1798 in the Layers window and select the PEI_highway map. Compare the roads represented in the PEI_highway map (the red dotted lines) to the roads you have just traced.



We can see that some of these roads correspond closely to modern roads, while others do not at all correspond. It would take further historical research to determine whether this is simply because the Holland map did not sufficiently survey roads at the time, or if roads have changed considerably since then.

Now create a third type of vector layer: a polygon vector. Click Layer -> New -> New Shapefile Layer. The New Vector Layer window will appear – in the Type category at the top, select Polygon

Click the Specify CRS button, and select NAD83(CSRS98) / Prince Edward Isl. Stereographic (EPSG: 2291), and then click OK

under New attribute, in the field beside Name, type in 'lot_name' in the field beside Year

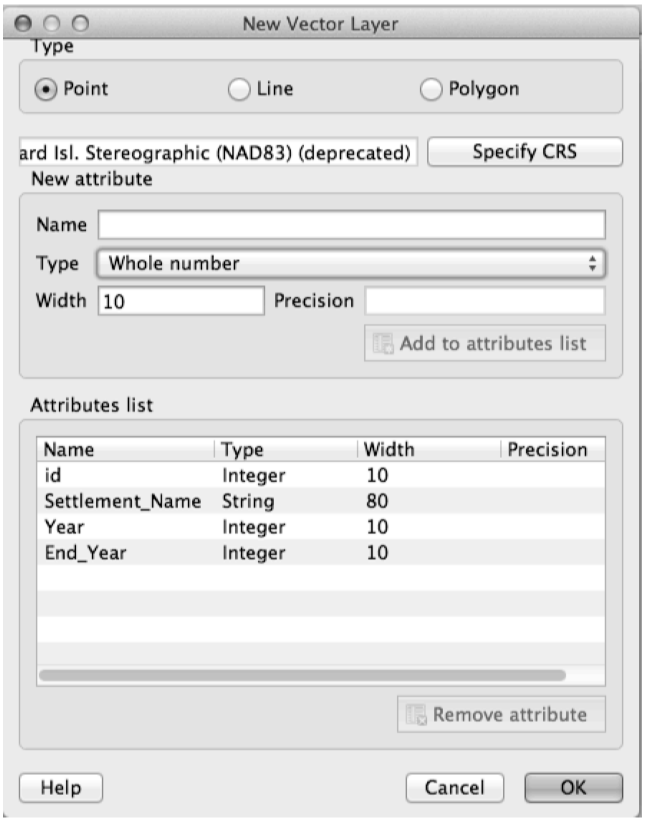
click Add to attributes list

Create a second attribute

under New attribute, in the field beside Name, type in Year

change the Type to Whole Number

click Add to attribute list



● Point ○ Line ○ Polygon

ard Isl. Stereographic (NAD83) (deprecated)

Name

Type: Whole number

Width: 10

Precision

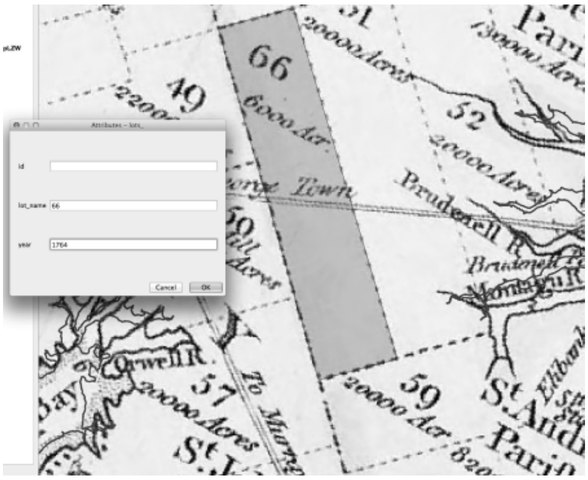
Name	Type	Width	Precision
id	Integer	10	
Settlement_Name	String	80	
Year	Integer	10	
End_Year	Integer	10	

Start by creating a polygon for lot 66, which is the only rectangular lot on the island

Click on Toggle Editing on top tool bar, and then click on Add Feature

click on all four corners of lot 66 and you will see a polygon created

right-click on the final corner and an Attributes window will appear. Add 66 to lot_names field and add 1764 (the year these lots were surveyed) to the Year field



Attributes – lots_

id

lot_name 66

year 1764

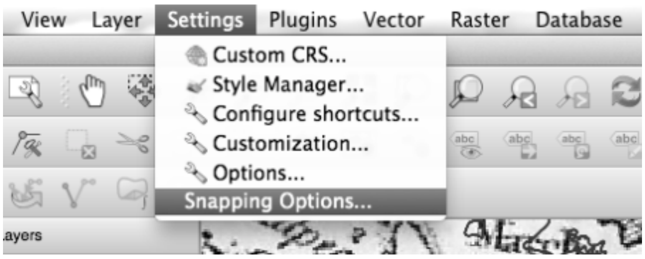
Cancel OK

We are now going to trace lot 38, which is just west of Havre-St-Pierre. Make sure that there is a check mark in the box beside PEI_HollandMap1798 layer in the Layers window

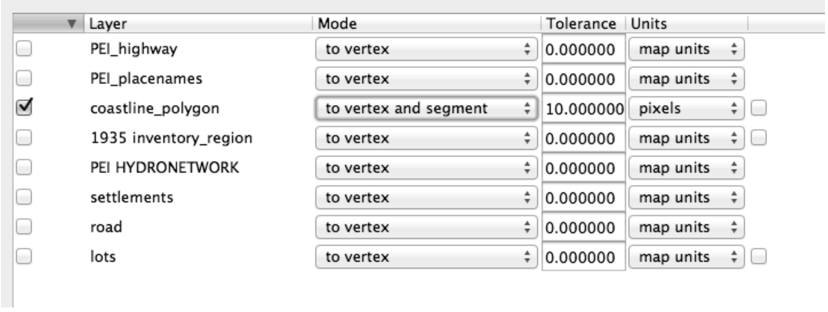
Click on Toggle Editing on top tool bar, and then click on Add Feature

Trace the outline of Lot 38, which is more difficult because of the coastline, as accurately as possible. In order to show you the Snap feature, we want you to trace along the modern coastline (snapping is an automatic editing operation that adjusts the feature you have drawn to coincide or lineup exactly with the coordinates and shape of another nearby feature)

select Settings-> Snapping Options

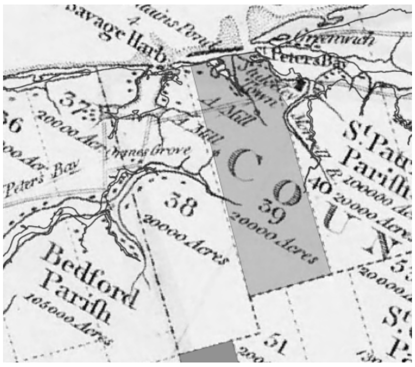


a Snapping options window will open: click on the box beside coastal_polygon, for the Mode category select "to vertex and segment", for Tolerance select 10.0, and for Units select 'pixels'. Click OK



	Layer	Mode	Tolerance	Units	
□	PEI_highway	to vertex	0.000000	map units	
□	PEI_placenames	to vertex	0.000000	map units	
☑	coastline_polygon	to vertex and segment	10.000000	pixels	□
□	1935 inventory_region	to vertex	0.000000	map units	□
□	PEI HYDRONETWORK	to vertex	0.000000	map units	
□	settlements	to vertex	0.000000	map units	
□	road	to vertex	0.000000	map units	
□	lots	to vertex	0.000000	map units	□

Make sure that the lots layer is selected in Layers window, and select Add Feature from the tool bar with your cursor click on the two bottom corners of your polygon just as you did with Lot 38. At the coastline you will notice that you have a collection of lines to trace around Savage Harbour. This is where the Snapping features becomes helpful. As you work to trace along the modern coastline it will significantly improve your accuracy by snapping your clicks directly on top of the existing line. The more clicks you make the more accurate it will be, but keep in mind that for many HGIS purposes obtaining extreme accuracy sometimes produces diminishing returns.



When you finish tracing and creating the polygon, select and deselect the various layers you have created, comparing and seeing what relationships you can deduce.

In Google Earth there were limitations on the types of features, attributes, and data provided by Google, and Google Earth did much of the work for you. That is fine when you are learning or want to quickly create maps. The advantage of using QGIS software to create new vector layers is that you have a great deal of freedom and control over the types of data you can use and the features and attributes that you can create. This in turn means that you can create custom maps far beyond what can be achieved in Google Earth or Google Maps Engine Lite. You have seen this firsthand with the points, lines, and polygons vector layers you learned how to create in this lesson. If you found data on, for example, public health records in the 18th century, you could create a new layer to work with what you already created showing the distribution of typhoid outbreaks and see if there are correlations with major roads and settlements. Moreover, GIS software allows you to not only spatially represent and present data in much more sophisticated ways, but to analyze and create new data in ways that aren't possible otherwise.

You have learned how to create vector layers. Make sure you save your work!

About the Authors

Jim Clifford is an assistant professor in the Department of History at the University of Saskatchewan. Josh MacFadyen is a Project Coordinator at the Network in Canadian History & Environment. Daniel Macfarlane is a Visiting Scholar in the School of Canadian Studies at Carleton University.

45. Georeferencing in QGIS 2.0

Jim Clifford, Joshua MacFadyen, Daniel MacFarlane – 2013

Lesson Goals

In this lesson, you will learn how to georeference historical maps so that they may be added to a GIS as a raster layer. Georeferencing is required for anyone who wants to accurately digitize data found on a paper map, and since historians work mostly in the realm of paper, georeferencing is one of our most commonly used tools. The technique uses a series of control points to give a two-dimensional object like a paper map the real world coordinates it needs to align with the three-dimensional features of the earth in GIS software (in the 'Intro to Google Maps and Google Earth' lesson we saw an 'overlay' which is a Google Earth shortcut version of georeferencing).

Georeferencing a historical map requires a knowledge of both the geography and the history of the place you are studying to ensure accuracy. The built and natural landscapes change over time, and it is important to confirm that the location of your control points — whether they be houses, intersections, or even towns — have remained constant. Entering control points in a GIS is easy, but behind the scenes, georeferencing uses complex transformation and compression processes. These are used to correct the distortions and inaccuracies found in many historical maps and stretch the maps so that they fit geographic coordinates. In cartography this is known as rubber-sheeting[466] because it treats the map as if it were made of rubber and the control points as if they were tacks 'pinning' the historical document to a three dimensional surface like the globe.

To offer some examples of georeferenced historical maps, we prepared some National Topographic Series maps hosted on the University of Toronto Map Library website courtesy of Marcel Fortin, and we overlaid them on a Google web map. Viewers can adjust the transparency with the slider bar on the top right, view the historical map as an overlay on terrain or satellite images, or click 'Earth' to switch into Google Earth mode and see 3D elevation and modern buildings (in Halifax and Dartmouth). Note: these historical images are large and will appear on the screen slowly, especially as you zoom into the Google map.

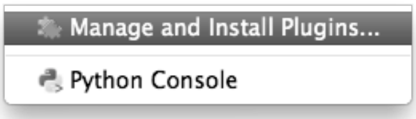
National Topographic System Maps – Halifax, 1920s[467]
National Topographic System Maps – Western PEI, 1939-1944
National Topographic System Maps – Eastern PEI 1939-1944

[466] 'Rubbersheeting', *Wikipedia*: https://en.wikipedia.org/wiki/Rubbersheeting

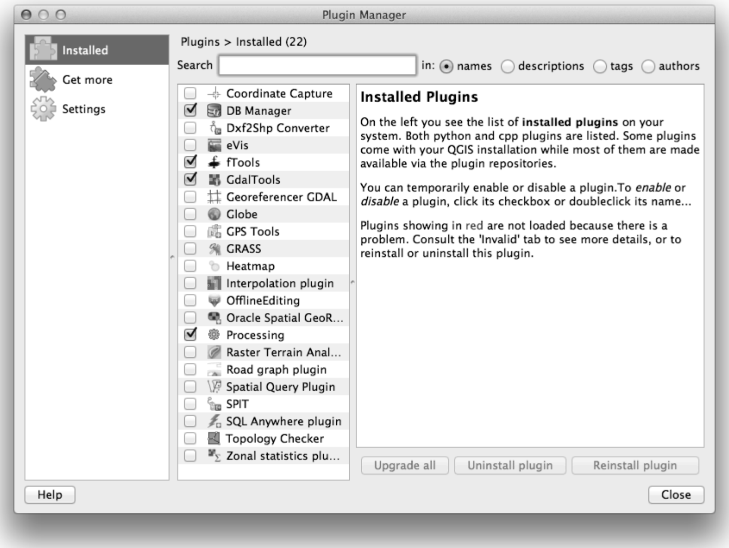
[467] 'National Topographic Systems Map – Halifax, 1920x': http://maps.library.utoronto.ca/datapub/digital/3400s_63_1929/maptile/Halifax/googlemaps.html

Getting Started

Before proceeding with georeferencing in Quantum GIS, we need to activate the appropriate Plugins. On the toolbar go to Plugins -> Manage and Install Plugins



A window titled Plugin Manager will open. Scroll down to Georeference GDAL and check the box beside it, and click OK.



Plugins > Installed (22)

☐ Coordinate Capture
☑ DB Manager
☐ Dxf2Shp Converter
☐ eVis
☑ fTools
☑ GdalTools
☐ Georeferencer GDAL
☐ Globe
☐ GPS Tools
☐ GRASS
☐ Heatmap
☐ Interpolation plugin
☐ OfflineEditing
☐ Oracle Spatial GeoR...
☑ Processing
☐ Raster Terrain Anal...
☐ Road graph plugin
☐ Spatial Query Plugin
☐ SPIT
☐ SQL Anywhere plugin
☐ Topology Checker
☐ Zonal statistics plu...

Installed Plugins

On the left you see the list of **installed plugins** on your system. Both python and cpp plugins are listed. Some plugins come with your QGIS installation while most of them are made available via the plugin repositories.

You can temporarily enable or disable a plugin.To *enable* or *disable* a plugin, click its checkbox or doubleclick its name...

Plugins showing in red are not loaded because there is a problem. Consult the 'Invalid' tab to see more details, or to reinstall or uninstall this plugin.

At this point, you need to shut down and relaunch QGIS. For the purposes of this example, and to keep things as simple as possible, don't reload your existing project but instead start a new project.

Set up the Coordinate Reference System (CRS)[468] correctly (see 'Installing QGIS 2.0 and adding Layers'[469] for a reminder)

Save this new project (under File menu, select Save Project) and call it 'georeferencing.'

Add the 'coastline_polygon' layer (see 'Installing QGIS 2.0 and adding Layers' for a reminder)

Open the Necessary GIS Layers

For the Prince Edward Island case study, we are going to use the township boundaries as control points because they were established in 1764 by

[468] 'Spatial reference system', *Wikipedia*: https://en.wikipedia.org/wiki/Spatial_reference_system

[469] Jim Clifford, Josh MacFadyen, and Daniel Macfarlane, 'Installing QGIS 2.0 and Adding Layers', *The Programming Historian*, (2013).

Samuel Holland, they are identified on most maps of PEI, and they have changed very minimally since then.

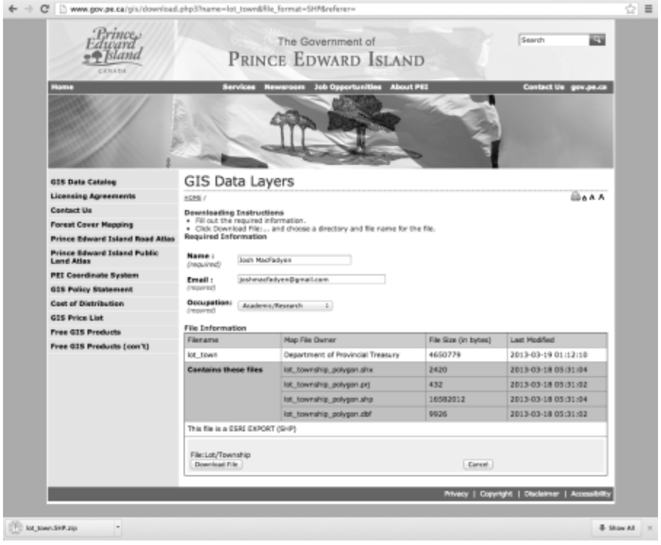
Download lot_township_polygon:

This is the shapefile containing the modern vector layer we are going to use to georeference the historical map. Note that townships were not given names but rather a lot number in 1764, so they are usually referred to as 'Lots' in PEI. Hence the file name 'lot_township_polygon'.

Navigate to the link below in your web browser, read/accept the license agreement, and then download the following (they will ask for your name and email before you can download the file).

http://www.gov.pe.ca/gis/license_agreement.php3?name=lot_town&file_format=SHP

After downloading the file called 'lot_township_polygon', move it into a folder that you can find later and unzip the file. (Remember to keep the files together as they are all required to open this layer in your GIS)



GIS Data Layers

Downloading Instructions

- Fill out the required information.
- Click Download File:... and choose a directory and file name for the file.

Required Information

Name : (required) Josh MacFadyen

Email : (required) joshmacfadyen@gmail.com

Occupation: (required) Academic/Research

File Information

Filename	Map File Owner	File Size (in bytes)	Last Modified
lot_town	Department of Provincial Treasury	4650779	2013-03-19 01:12:10
Contains these files	lot_township_polygon.shx	2420	2013-03-18 05:31:04
	lot_township_polygon.prj	432	2013-03-18 05:31:02
	lot_township_polygon.shp	16582012	2013-03-18 05:31:04
	lot_township_polygon.dbf	9926	2013-03-18 05:31:02

This file is a ESRI EXPORT (SHP)

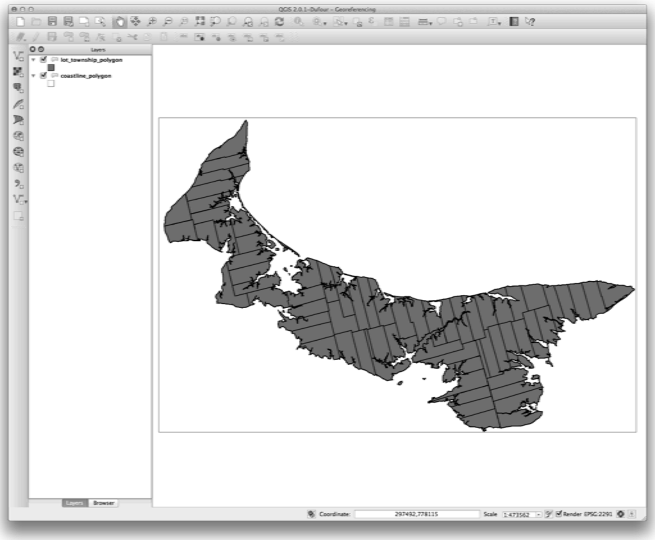
File:Lot/Township

Download File   Cancel

Add lot_township_polygon to QGIS:

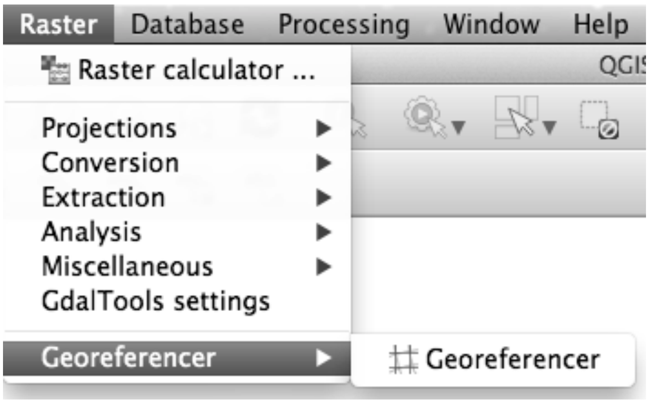
under Layer on the toolbar, choose Add Vector Layer (alternatively the same icon you see next to 'Add Vector Layer' can also be selected from the tool bar)
Click Browse. Navigate to your unzipped file and select the file titled 'lot_township_polygon.shp'
Click Open

For more information on adding and visualizing layers see 'Installing QGIS 2.0 and adding Layers'.[470]



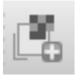
Open the Georeferencer Tool

Georeferencer is now available under the Raster menu on the toolbar – select it.



Add your historical map:

In the resulting window, click on the Open Raster button on the top left (which looks identical to the Add Raster layer).



Find the file titled 'PEI_LakeMap1863.jpg' on your computer and select Open (the file can be downloaded from the Geospatial Historian website,[471] or in its original location at the Island Imagined online map repository)[472]

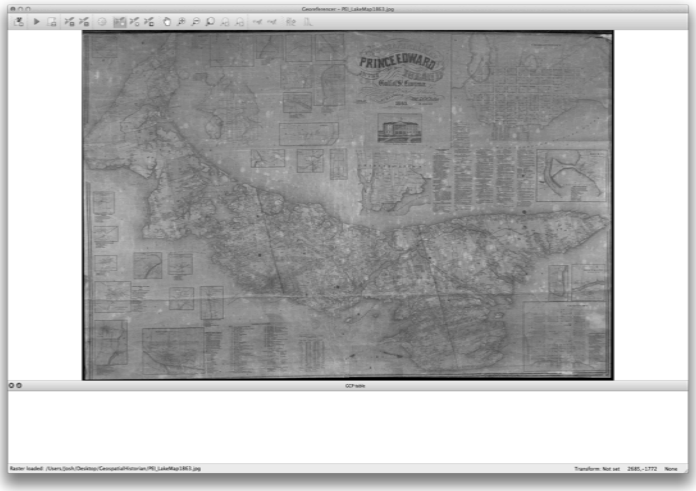
[470] Jim Clifford, Josh MacFadyen, and Daniel Macfarlane, 'Installing QGIS 2.0 and Adding Layers', *The Programming Historian*, (2013).

[471] Available at: https://geospatialhistorian.files.wordpress.com/2013/02/pei_lakemap1863.jpg

[472] 'Island Imagined': http://www.islandimagined.ca/fedora/repository/imagined%3A208687

You will be prompted to define this layer's coordinate system. In the Filter box search for '2291', then in the box below select 'NAD83(CSRS98) / Prince Edward ...'

The result will look like this:



Raster loaded: /Users/josh/Desktop/GeospatialHistorian/PEI_LakeMap1863.jpg

Transform: Not set 2685,-1772 None

Adding control points:

Plan the locations you are going to use as control points in advance of the steps that follow. It is much easier to navigate around the historical map first, so get a good idea of the best points to use and keep them in mind.

Some tips for choosing control points:

How many points do you need? Usually the more points you assign the more accurate your georeferenced map will be. Two control points will tell the GIS to scale and rotate the map to those two points, but in order to truly rubber-sheet the historical document you need to add more points.

Where should you put control points? Select areas as close as possible to the four corners of your map so that these outer areas do not get omitted in the rubber-sheeting.

Select additional control points close to your area of interest. Everything in between the four corner control points should georeference evenly, but if you are concerned about the accuracy of one place in particular, make sure to select additional control points in that area.

Select the middle of intersections and roads, because the edges of roads changed a certain amount over time as road improvements were made.

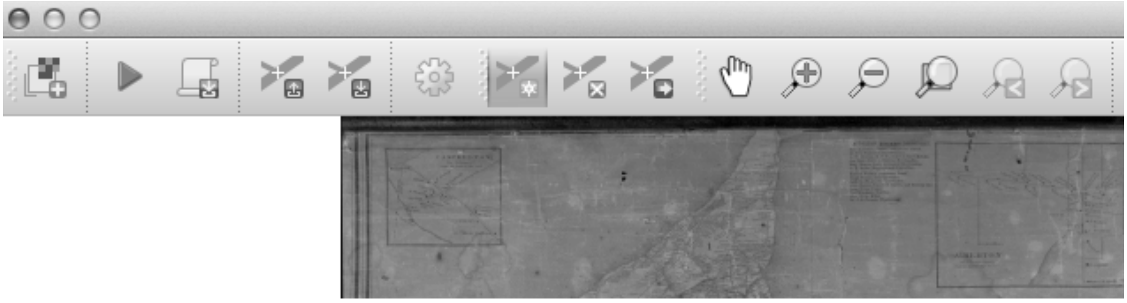
Check that your control points did not change location over time. Roads were often re-routed, and even houses and other buildings were moved, especially in Atlantic Canada![473]

Add your first control point:

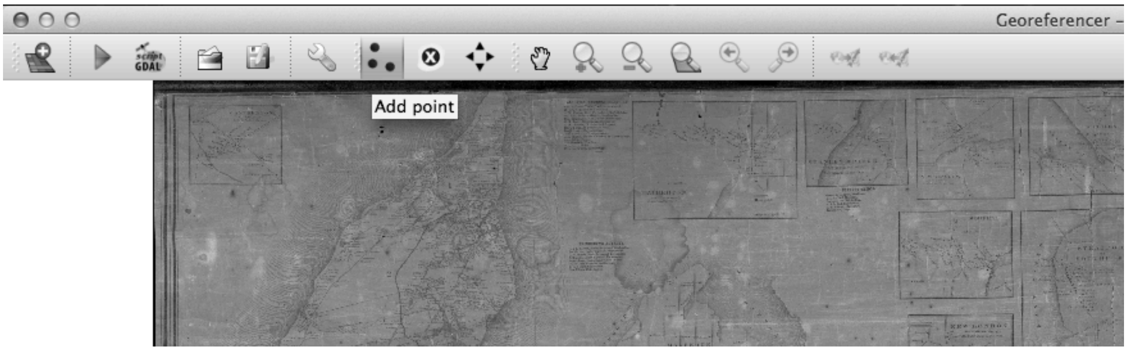
[473] Robert Melin, 'Tilting: House Launching, Side Hauling, Potato Trenching, and Other Tales from a Newfoundland Fishing Village' (2008).

First, navigate to the location of your first control point on the **historical map**.

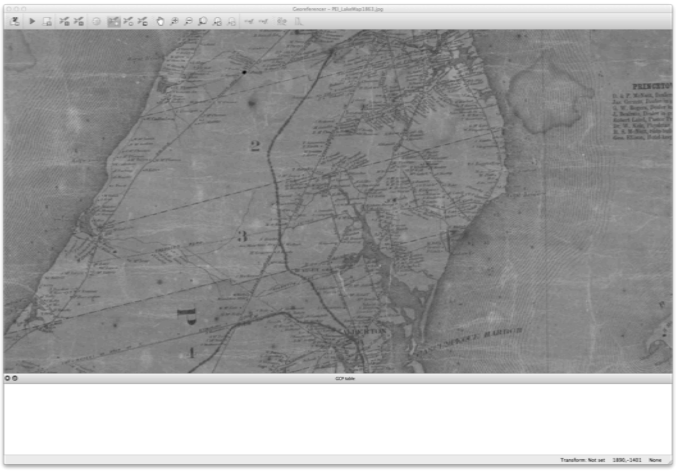
click on Zoom In Magnifying Glass on the window tool bar or use the mouse roller wheel to zoom in



- zoom in to a point which you can recognize on both your printed map and your GIS
- Click on Add Point on toolbar

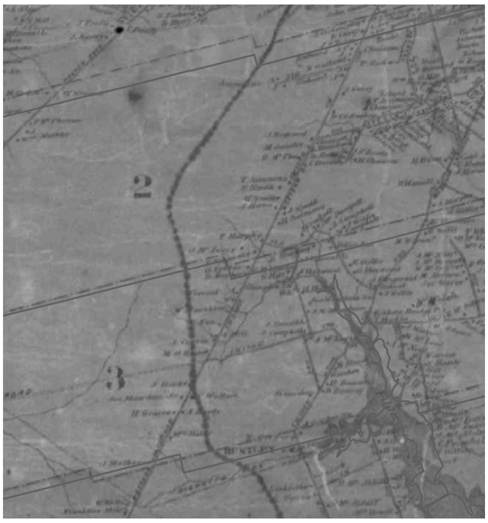


Click on the place in the printed map that you can locate in your GIS (i.e. the control point). The Georeferencer window will now minimize automatically. If it does not (some versions have a bug in this plugin) manually minimize the window
Click on the place in the GIS which matches the control point



At this stage we identified a problem in lot boundaries. We planned to use the location where the southern border of Lot 1 at the West end of the

Province contains a "dog leg" near the middle of the land mass. However, it was clear that not all the dog legs on these lots matched the historical map. It is possible that lot boundaries have changed somewhat in the 250 years since they were established, so it is best to choose the point you are most sure of. In this case the dog leg between Lot 2 and Lot 3 was fine (see arrow). It was the border of Lots 3 and 4 that has changed. The discrepancy at the border of 1 and 2 shows that more control points are needed to properly rubber-sheet this somewhat distorted 1863 map to the Provincial GIS layer

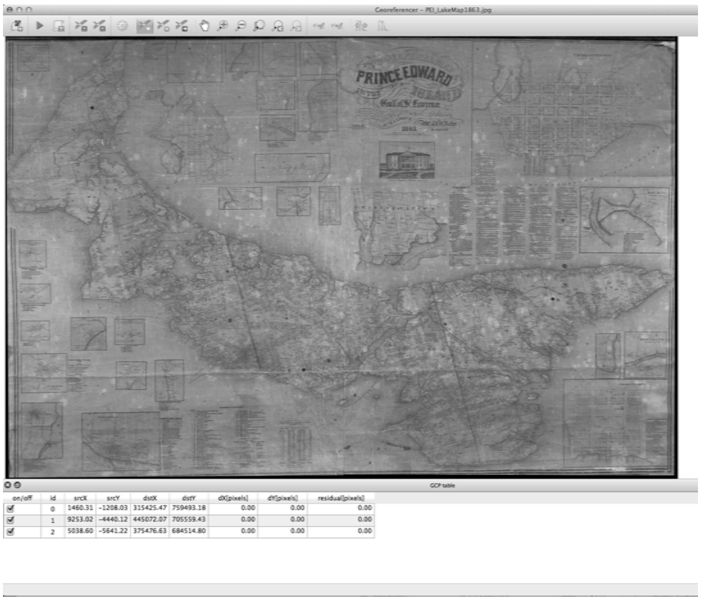


Add at least one more control point:

return to the Georeferencer window and repeat the steps under '*Add your first control point*' above, to add additional control points.

Add a point close to the opposite side of your printed map (the further apart your control points are placed the more accurate the georeferencing process) and another one near Charlottetown

return to the Georeferencer window. You should see three red dots on the printed map, and three records in the GCP table at the bottom of your window (outlined in red on the following image)



on/off	id	srcX	srcY	dstX	dstY	dX[pixels]	dY[pixels]	residual[pixels]
☑	0	1460.31	-1208.03	315425.47	759493.18	0.00	0.00	0.00
☑	1	9253.02	-4440.12	445072.07	705559.43	0.00	0.00	0.00
☑	2	5038.60	-5641.22	375476.63	684514.80	0.00	0.00	0.00

Determine the transformation settings:

Before you click Play and start the automated georeferencing process you need to tell QGIS where to save the new file (this will be a raster file), how it should interpret your control points, and how it should compress the image.

Click on the Transformation Settings button

Most of these settings can be left as default: linear transformation type, nearest neighbour resampling method, and LZW compression. (The world file[474] is not necessary, unless you want to georeference the same image again in another GIS or if someone else needs to georeference the image and does not have access to your GIS data, coordinate reference system, etc.) The target SRS is not important, but you could use this feature to give the new raster a different reference system.

Assign a folder for your new georeferenced raster file. Tif[475] is the default format for rasters georeferenced in QGIS.

Be aware that a Tif file is going to be much larger than your original map, even with LZW compression, so make sure you have adequate space if you are using a jump drive. (*warning:* the Tif file produced from this 6.8 Mb .jpg will be **over 1GB** once georeferenced. One way to manage the size of the georeferenced raster file while maintaining a high enough resolution for legibility is to crop out only the area needed for the map project. In this

[474] 'World File', *Wikipedia*: https://en.wikipedia.org/wiki/World_file

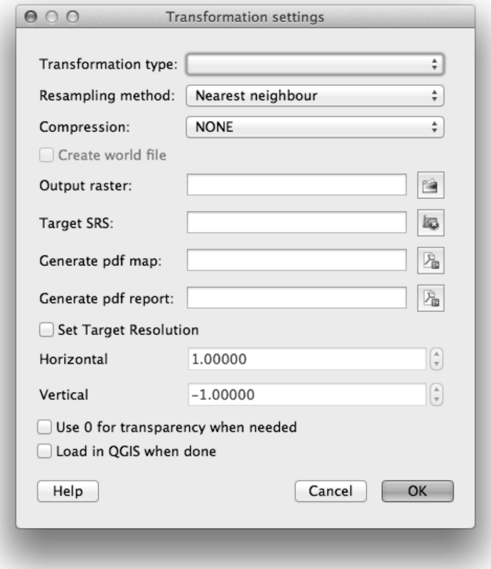
[475] 'Tagged Image File Format', *Wikipedia*: https://en.wikipedia.org/wiki/Tagged_Image_File_Format

case, a lower resolution option is also available from the Island Imagined[476] online map repository.)

Leave the target resolution at the default

You can select 'Use 0 transparency when needed' to eliminate black spaces around the edges of the map, but this is not necessary and you can experiment as needed

Make sure 'Load in QGIS' is selected to save a step. This will automatically add the new file to your GIS's Table of Contents so that you don't have to go looking for the Tif file later


Transformation settings

Transformation type:

Resampling method: Nearest neighbour

Compression: NONE

□ Create world file

Output raster:

Target SRS:

Generate pdf map:

Generate pdf report:

□ Set Target Resolution

Horizontal 1.00000

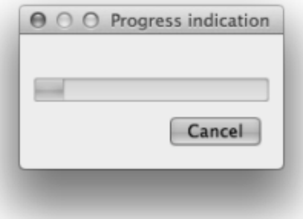
Vertical -1.00000

□ Use 0 for transparency when needed

□ Load in QGIS when done

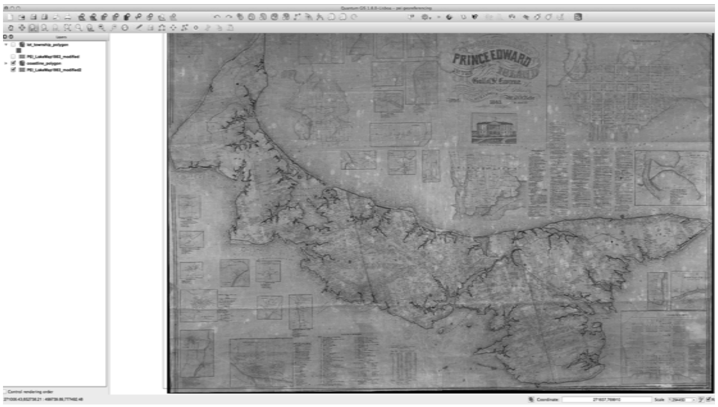
Georeference!

Click on the Play button on the toolbar (beside Add Raster) – this begins the georeferencing process



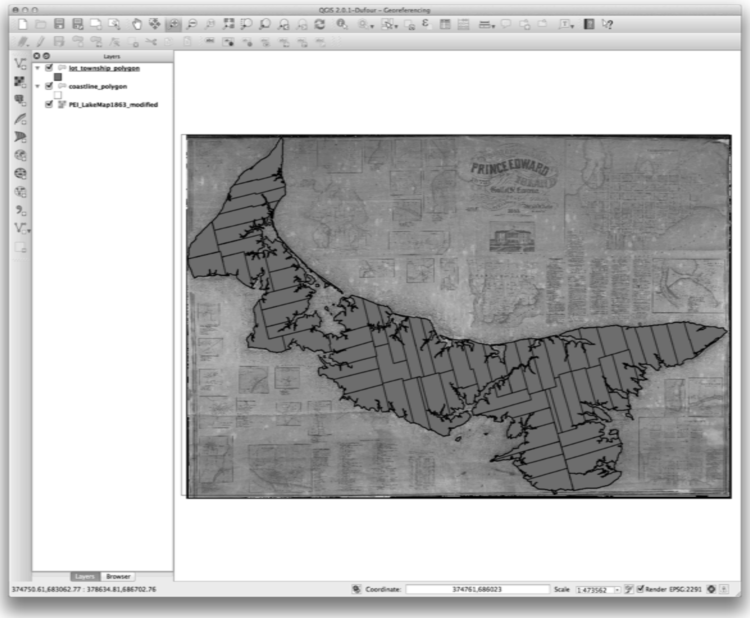
A window will appear titled Define CRS: select 2291, click OK

[476] 'Island Imagined': http://www.islandimagined.ca/fedora/repository/imagined%3A208687

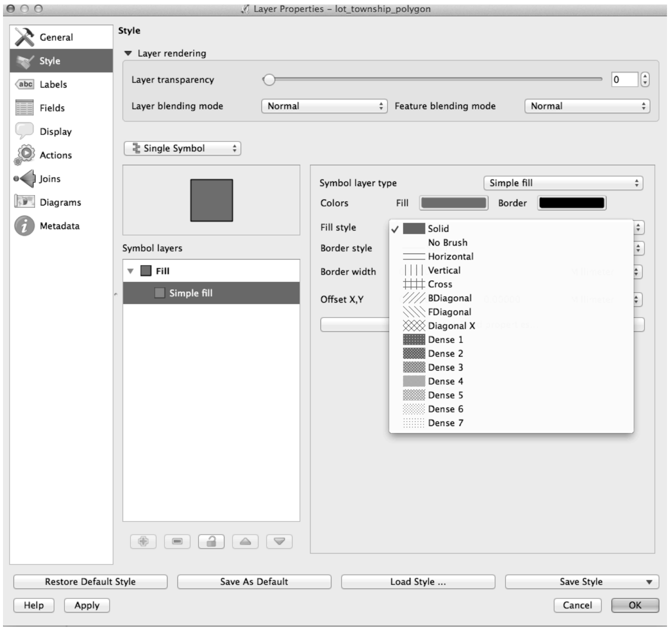


Explore your map:

Drag the new layer 'PEI_LakeMap1863_modified' down to the bottom of your Table of Contents (i.e. below the 'lot_township_polygon' layer



Change the fill of the lot_township_polygon layer to 'no brush' by Selecting the layer, clicking on Layer -> Properties, and clicking on Symbol Properties. Click OK



Layer transparency: 0

Layer blending mode: Normal

Feature blending mode: Normal

Symbol layer type: Simple fill

Colors: Fill, Border

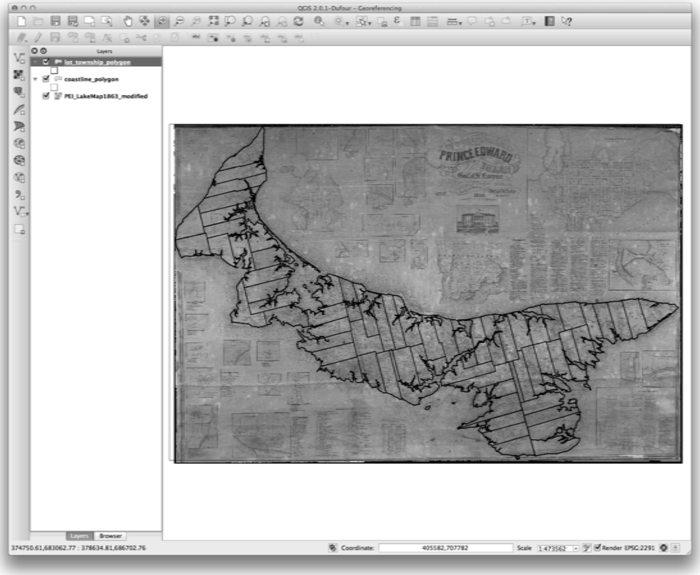
Fill style: ✓ Solid

Border style

Border width

Offset X,Y

Now you should see the modern GIS layer with the historical map in behind



☑ lot_township_polygon

☑ coastline_polygon

☑ PEI_LakeMap1863_modified

374750.61,683062.77 : 378634.81,686702.76

Coordinate: 405582,707782

Scale 1:473562

☑ Render EPSG:2291

Now that you have a newly georeferenced map in your GIS you can explore the layer, adjust the transparency, contrast and brightness, and go back through 'Creating New Vector Layers in QGIS'[477] to digitize some of the historical information that you have created. For instance, this georeferenced map of PEI shows the locations of all homes in 1863, including the name of the head of household. By assigning points on the map you can enter home locations and owner names and then analyze or share that new geospatial layer as a shapefile.

[477] Jim Clifford, Josh MacFadyen, and Daniel Macfarlane, 'Creating New Vector Layers in QGIS', *The Programming Historian* (2013).

By digitizing line vectors such as roads or coastlines you can compare the location of these features with other historical data, or simply compare them visually with the lot_township_polygon layer in this GIS.

In more advanced processes you can even drape this georeferenced image over a DEM (digital elevation model) to give it a hillshade terrain or 3D effect and perform a 'fly-over' of PEI homes in the nineteenth century.

About the Authors

Jim Clifford is an assistant professor in the Department of History at the University of Saskatchewan. Josh MacFadyen is a Project Coordinator at the Network in Canadian History & Environment. Daniel Macfarlane is a Visiting Scholar in the School of Canadian Studies at Carleton University.

Part Six: Sustaining Data

We spend a lot of time and energy (as well as money) building digital history resources. The lessons in this section provide tips and methods for helping to ensure that energy is preserved for the future. This is about good practice.

46. Preserving Your Research Data

James Baker – 2014.

Background

In his 2003 essay 'Scarcity or Abundance' Roy Rosenzweig sought to alert historians to what he called 'the fragility of evidence in the digital era' (Rosenzweig, 736).[478] And whilst his concerns were focused on sources available on the open web, they can easily be extended to the born-digital materials – or data – historians create during their research.

It is this research data that the present guide will focus upon. But why?

Well, historians are moving toward using computers as the default means of storing all of their research data, their stuff. Their manuscripts have been digital objects for some time and their research is moving accordingly – be that in the form of typed notes, photographs of archives, or tabulated data. Moreover research data held in a digital form has clear advantages over its physical antecedents: it can be browsed and searched, hosted in ways that enable access in many places, and merged with or queried against other research data.

Merely putting research data into digital form does not guarantee it will survive. Here by survival I neither mean survive in a literal sense nor in a survival as readable by the next version of Microsoft Word sense, but rather in a usable by people sense. For if not a problem solved, the nuts and bolts of how to preserve research data for the future is a problem whose potential solutions have already been addressed at length, both with and without historians in mind. So too have data management experts, services and the like talked about scholarly best practice with regards to documenting, structuring and organising research data. In spite of all this, research data generated by an individual historian is at risk of loss if that historian is not able to generate and preserve it in a form they can understand and find meaningful years or decades after the fact, let alone someone else wading through the idiosyncrasies of their research process. In short, there is a risk of loss as a consequence of data being detached from the context of its creation, from the tacit knowledge that made it useful at the time of preparing talk X or manuscript Y. As William Stafford Noble puts it:

> The core guiding principle is simple: Someone unfamiliar with your project should be able to look at your computer files and understand in detail what you did and why […]Most commonly, however, that "someone" is you. A few months from now, you may not remember what you were up to when you created a particular set of files, or you may not remember what

[478] Roy Rosenzweig, 'Scarcity or Abundance? Preserving the Past in a Digital Era': http://www.islandimagined.ca/fedora/repository/imagined%3A208687

conclusions you drew. You will either have to then spend time reconstructing your previous experiments or lose whatever insights you gained from those experiments.[479]

Drawing on the lessons and expertise of research data experts, the present guide will suggest ways in which historians can document and structure their research data so as to ensure it remains useful in the future. The guide is not intended to be prescriptive, rather it is assumed readers will iterate, change, and adapt the ideas presented to best fit their own research.

Documenting research data

Birkwood, Katie (girlinthe). "Victory is mine: while ago I worked out some Clever Stuff ™ in Excel. And I MADE NOTES ON IT. And those notes ENABLED ME TO DO IT AGAIN." 7 October 2013, 3:46 a.m.. Tweet.[480]

The purpose of documentation is to capture the process of data creation, changes made to data, and tacit knowledge associated with data. Project management methodologies, such as PRINCE2,[481] place great emphasis on precise, structured, and verbose documentation. Whilst there are benefits to this approach, especially for large, complex, multi-partner projects, the average working historian is more likely to benefit from a flexible, bespoke approach to documentation that draws on, but is not yoked to, project management principles. In the case of historical research, the sort of documentation that might be produced to preserve the usefulness of research data includes:

documentation describing notes taken whilst examining a document in an archive, such as the archival reference for the original document, how representative the notes are (e.g. full transcriptions, partial transcriptions, or summaries), how much of the document was examined, or decisions taken to exclude sections of the document from the research process.

documentation describing tabulated data, such as how it was generated (e.g. by hand or in an automated manner), archival references for the original sources some data came from, or what attributes of the original sources were retained (and why).

documentation describing a directory of digital images, such as how each image was created, where those images were downloaded from, or research notes that refer to them.

As the last example suggests, one of the key purposes of documentation is to describe the meaningful links that exist between research data, links that may not remain obvious over time.

[479] William Stafford Noble (2009) A Quick Guide to Organizing Computational Biology Projects. PLoSComputBiol 5(7): e1000424. doi:10.1371/journal.pcbi.1000424

[480] Tweet by Katie Birkwood, https://twitter.com/Girlinthe/status/387166944094199809

[481] 'PRINCE2', *Wikipedia*: https://en.wikipedia.org/wiki/PRINCE2

When to document is very much up to the individual and the rhythm of their research. The main rule is to get into a habit of writing and updating documentation at regular intervals, ideally every time a batch of work is finished for the morning, afternoon, or day. At the same time it is important not to worry about perfection, rather to aim to write consistent and efficient documentation that will be useful to you, and hopefully someone else using your research data, years after the fact.

File formats

Research data and documentation should ideally be saved in platform agnostic formats[482] such as .txt for notes and .csv (comma-separated values) or .tsv (tab-seperated values) for tabulated data. 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, from a preservation perspective, 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 machine-readable and hence can neither be queried and searched nor are appropriate for large quantities of information. 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.

It is likely that on many occasions these notation schemes will emerge from existing individual practice (and as a consequence will need to be documented), though existing schema such as Markdown[483] are available (Markdown files are saved as .md). 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, though by no means essential, for working with .md files. Mac or Unix users may find 'Komodo Edit' or 'Text Wrangler' helpful.[484]

[482] 'Cross-platform', *Wikipedia*: https://en.wikipedia.org/wiki/Cross-platform

[483] 'Markdown', *Wikipedia*: https://en.wikipedia.org/wiki/Markdown

[484] 'Komodo Edit': http://komodoide.com/komodo-edit/; 'TextWrangler': https://itunes.apple.com/gb/app/id404010395?mt=12

Recap 1

To recap, the key points about documentation and file formats are:

Aim for documentation to capture in a precise and consistent manner the tacit knowledge surrounding a research process, be that with relation to note taking, generating tabulated data, or accumulating visual evidence.

Keep documentation simple by using file formats and notation practices that are platform agnostic and machine-readable.

Build time for updating and creating documentation into your workflow without allowing documentation work to become a burden.

Make an investment in leaving a paper trail now to save yourself time attempting to reconstruct it in the future.

Structuring research data

Documenting your research is made easier by structuring your research data in a consistent and predictable manner.

Why?

Well, every time we use a library or archive catalogue, we rely upon structured information to help us navigate data (both physical and digital) the library or archive contains. Without that structured information, our research would be much poorer.

Examining URLs is a good way of thinking about why structuring research data in a consistent and predictable manner might be useful in your research. Bad URLs are not reproducible and hence, in a scholarly context, not citable. On the contrary, 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:

website name/year(4 digits)/month (2 digits)/day (2 digits)/words-of-title-separated-by-hyphens

http://cradledincaricature.com/2014/02/06/comic-art-beyond-the-print-shop/

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

website name/section subdivision/year (4 digits)/month (3 characters)/day (2 digits)/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:

website name/record/reference number
http://www.cartoons.ac.uk/record/SBD0931

And the Old Bailey Online uses the format:

website name/browse.jsp?ref=reference number
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 consistent and predictable data structures readable both by humans and machines. Transferring this to digital data accumulated during the course of historical research makes research data easier to browse, to search and to query using the standard tools provided by our operating systems (and, as we shall see in a future lesson, by more advanced tools).

In practice (for OS X and Linux users, replace all backslashes hereafter with forward slash), the structure of a good research data archive might look something like this:

A base or root directory, perhaps called 'work'.

```
\work\
```

A series of sub-directories.

```
     \work\events\
     \research\
     \teaching\
     \writing\
```

Within these directories are series of directories for each event, research project, module, or piece of writing. Introducing a naming convention that includes a date elements keeps the information organised without the need for subdirectories by, say, year or month.

```
\work\research\2014-01_Journal_Articles
              \2014-02_Infrastructure
```

Finally, further sub-directories can be used to separate out information as the project grows.

```
\work\research\2014_Journal_Articles\analysis
                                    \data
                                    \notes
```

Obviously not all information will fit neatly within any given structure and as new projects arise taxonomies will need to be revisited. Either way, idiosyncrasy is fine so long as the overall directory structure is consistent and predictable, and so long as anything that isn't is clearly documented:

for example, the 'writing' sub-directory in the above structure might include a .txt file stating what it contained (drafts and final version of written work) and what it didn't contain (research pertaining to that written work).

The name of this .txt file, indeed any documentation and research data, is important to ensuring it and its contents are easy to identify. 'Notes about this folder.docx' is not a name that fulfils this purpose, whilst '2014-01-31_Writing_readme.txt' is as it replicates the title of the directory and included some date information (North American readers should note that I've chosen the structure year_month_date). A readme file I made for a recent project https://www.dropbox.com/s/i12cv5rdnfbdoz3/network_analysis_of_Isaac_Cruikshank_and_his_publishers_readme.txt contains the sort of information that you and other users of your data might find useful.

An cautionary tale should be sufficient to confirm the value of this approach. During the course of a previous research project, I collected some 2,000 digital images of Georgian satirical prints from a number of online sources, retaining the file names upon download. Had I applied a naming convention to these from the outset (say 'PUBLICATION YEAR_ARTIST SURNAME_TITLE OF WORK.FORMAT') I would be able to search and query these images. Indeed starting each filename with some version of YYYYMMDD would have meant that the files could be sorted in chronological order on Windows, OS X and Linux. And ensuring that all spaces or punctuation (except dash, dot and underscore) were removed from the filenames in the process of making them consistent and predictable, would have made command line work with the files possible. But I did not, and as it stands I would need to set aside a large amount of time to amend every filename individually so as to make the data usable in this way.

Further, applying such naming conventions to all research data in a consistent and predictable manner assists with the readability and comprehension of the data structure. For example for a project on journal articles we might choose the directory…

```
\work\research\2014-01_Journal_Articles\
```

…where the year-month elements captures when the project started.

Within this directory we include a \data\ directory where the original data used in the project is kept.

```
2014-01-31_Journal_Articles.tsv
```

Alongside this data is documentation that describes 2014-01-31_Journal_Articles.tsv.

```
2014-01-31_Journal_Articles_notes.txt
```

Going back a directory level to \2014-01_Journal_Articles\ we create the \analysis\ directory in which we place:

```
2014-02-02_Journal_Articles_analysis.txt
2014-02-15_Journal_Articles_analysis.txt
```

Note the different month and date attributes here. These reflect the dates on which data analysis took place, a convention described briefly in 2014-02-02_Journal_Articles_analysis_readme.txt.

Finally, a directory within \data\ called \derived_data\ contains data derived from the original 2014-01-31_Journal_Articles.tsv. In this case, each derived .tsv file contains lines including the keywords, 'africa', 'america', 'art' et cetera, and are named accordingly.

```
2014-01-31_Journal_Articles_KW_africa.tsv

2014-01-31_Journal_Articles_KW_america.tsv

2014-02-01_Journal_Articles_KW_art .tsv

2014-02-02_Journal_Articles_KW_britain.tsv
```

Recap 2

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

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

Summary

This lesson has suggested ways for documenting and structuring research data, the purpose of which is to ensure that data is preserved by capturing tacit knowledge gained during the research process and thus making the information easy to use in the future. It has recommended the use of platform agnostic and machine-readable formats for documentation and research data. It has suggested that URLs offer a practice example of both good and bad data structures that can be replicated for the purposes of a historian's research data.

These suggestions are intended merely as guides; it is expected that researchers will adapt them to suit their purposes. In doing so, it is recommended that researchers keep digital preservation strategies and project management best practice in mind, whilst ensuring that time spent documenting and structuring research does not become a burden. After all, the purpose of this guide is to make more not less efficient historical research that generates data. That is, your research.

Further Reading

Ashton, Neil, 'Seven deadly sins of data publication', School of Data blog (17 October 2013) http://schoolofdata.org/2013/10/17/seven-deadly-sins-of-data-publication/

Hitchcock, Tim, 'Judging a book by its URLs', Historyonics blog (3 January 2014) http://historyonics.blogspot.co.uk/2014/01/judging-book-by-its-url.html

Howard, Sharon, 'Unclean, unclean! What historians can do about sharing our messy research data', Early Modern Notes blog (18 May 2013) http://earlymodernnotes.wordpress.com/2013/05/18/unclean-unclean-what-historians-can-do-about-sharing-our-messy-research-data/

Noble, William Stafford, A Quick Guide to Organizing Computational Biology Projects.PLoSComputBiol 5(7): e1000424 (2009) http://dx.doi.org/10.1371/journal.pcbi.1000424

Oxford University Computing Services, 'Sudamih Project. Research Information Management: Organising Humanities Material' (2011) http://dspace.jorum.ac.uk/xmlui/handle/10949/14725

Pennock, Maureen, 'The Twelve Principles of Digital Preservation (and a cartridge in a repository…)', British Library Collection Care blog (3 September 2013) http://britishlibrary.typepad.co.uk/collectioncare/2013/09/the-twelve-principles-of-digital-preservation.html

Pritchard, Adam, 'Markdown Cheatsheet' (2013) https://github.com/adam-p/markdown-here

Rosenzweig, Roy, 'Scarcity or Abundance? Preserving the Past in a Digital Era', The American Historical Review 108:3 (2003), 735-762.

UK Data Archive, 'Documenting your Data' http://data-archive.ac.uk/create-manage/document

About the Author

James Baker is a lecturer of digital history at the University of Sussex.

47. Getting Started with Markdown

Sarah Simpkin – 2015

Lesson goals

In this lesson, you will be introduced to Markdown, a plain text-based syntax for formatting documents. You will find out why it is used, how to format Markdown files, and how to preview Markdown-formatted documents on the web.

Since Programming Historian lessons are submitted as Markdown files, I have included PH-specific examples whenever possible. It is my hope that this guide will be useful to you if you are considering authoring a lesson for this site.

What is Markdown?

Developed in 2004 by John Gruber,[485] Markdown refers to both (1) a way of formatting text files, as well as (2) a Perl utility to convert Markdown files into HTML. In this lesson, we'll focus on the first part and learn to write files using the Markdown syntax.

Plain text files have many advantages over other formats. For one, they are readable on virtually all devices. They have also withstood the test of time better than other file types -- if you've ever tried to open a document saved in a legacy word processor format, you'll be familiar with the compatibility challenges involved.

By following Markdown syntax, you'll be able to produce files that are both legible in plain text and ready to be styled on other platforms. Many blogging engines, static site generators, and sites like GitHub[486] also support Markdown, and will render these files into HTML for display on the web. Additionally, tools like Pandoc can convert files into and out of Markdown. For more on Pandoc, visit the lesson on 'Sustainable authorship in plain text using Pandoc and Markdown' by Dennis Tenen and Grant Wythoff.[487]

Markdown Syntax

Markdown files are saved with the extension `.md`, and can be opened in a text editor such as TextEdit, Notepad, Sublime Text, or Vim. Many websites and publishing platforms also offer web-based editors and/or extensions for entering text using Markdown syntax.

[485] John Gruber, 'Markdown', *Daring Fireball*: http://daringfireball.net/projects/markdown/

[486] 'Github': https://github.com/

[487] Dennis Tenen and Grant Wythoff, 'Sustainable Authorship in Plain Text using Pandoc and Markdown', *The Programming Historian*, (2014).

In this tutorial, we'll be practicing Markdown syntax in the browser using StackEdit.[488] You'll be able to enter Markdown-formatted text on the left and immediately see the rendered version alongside it on the right.

Since all Programming Historian lessons are written in Markdown, we can examine these files in StackEdit too. From the StackEdit editor,[489] click on the # in the upper left corner for a menu. Choose `Import from URL`, then paste the following URL to display the "Intro to Bash" lesson in the editor:

```
https://raw.githubusercontent.com/programminghistorian/jekyll/gh-pages/lesso
ns/intro-to-bash.md
```

You'll notice that while the right panel features a more elegant rendering of the text, the original Markdown file on the left is still fairly readable.

Now, let's dive into the lesson by writing our own Markdown syntax. Create a new document in StackEdit by clicking the folder icon in the upper right and choosing `New document`. You may enter a title for the document in the textbox on the top of the page.

Headings

Four levels of headings are available in Markdown, and are indicated by the number of # preceding the heading text. Paste the following examples into the textbox on your left:

```
# First level heading
## Second level heading
### Third level heading
#### Fourth level heading
```

First and second level headings may also be entered as follows:

```
First level heading
=======

Second level heading
----------
```

These will render as:

First level heading

Second level heading

Third level heading

Fourth level heading

[488] 'Stackedit': https://stackedit.io/

[489] 'Stackedit Editor': https://stackedit.io/editor

First level heading

Second level heading

Notice how the Markdown syntax remains understandable even in the plain text version.

Paragraphs & Line Breaks

Try typing the following sentence into the textbox:

```
Welcome to the Programming Historian.

Today we'll be learning about Markdown syntax.
This sentence is separated by a single line break from the preceding one.
```

This renders as:

Welcome to the Programming Historian.

Today we'll be learning about Markdown syntax. This sentence is separated by a single line break from the preceding one.

Paragraphs must be separated by an empty line. Leave an empty line between `syntax.` and `This` to see how this works. In some implementations of Markdown, single line breaks must also be indicated with two empty spaces at the end of each line. This is unnecessary in the GitHub Flavored Markdown variant that StackEdit uses by default.[490]

Adding Emphasis

Text can be italicized by wrapping the word in * or _ symbols. Likewise, bold text is written by wrapping the word in ** or __.

Try adding emphasis to a sentence using these methods:

```
I am **very** excited about the _Programming Historian_ tutorials.
```

This renders as:

I am **very** excited about the *Programming Historian* tutorials.

Making Lists

Markdown includes support for ordered and unordered lists. Try typing the following list into the textbox:

[490] 'Categories/Writing on Github': https://help.github.com/categories/writing-on-github/

```
Shopping List
----------
* Fruits
    * Apples
    * Oranges
    * Grapes
* Dairy
    * Milk
    * Cheese
```

Indenting the * by four spaces will allow you to created nested items.

This renders as:

Shopping List

Fruits

Apples

Oranges

Grapes

Dairy

Milk

Cheese

Ordered lists are written by numbering each line. Once again, the goal of Markdown is to produce documents that are both legible as plain text and able to be transformed into other formats.

```
To-do list
----------
1. Finish Markdown tutorial
2. Go to grocery store
3. Prepare lunch
```

This renders as:

To-do list

Finish Markdown tutorial
Go to grocery store
Prepare lunch

Code Snippets

Representing code snippets differently from the rest of a document is a good practice that improves readability. Typically, code is represented in monospaced type. Since Markdown does not distinguish between fonts, we represent code by wrapping snippets in back-tick characters like ` . For example, `<br />`. Whole blocks of code are written by typing three back-tick characters before and after each block. In the StackEdit preview window, this will render a shaded box with text in a monospaced font.

Try typing the following text into the textbox:

~~~
```html
<html>
    <head>
        <title>Website Title</title>
    </head>
    <body>
    </body>
</html>
```
~~~

This renders as:

```
    <html>
        <head>
            <title>Website Title</title>
        </head>
        <body>
        </body>
    </html>
```

Notice how the code block renders in a monospaced font.

Blockquotes

Adding a > before any paragraph will render it as a blockquote element.

Try typing the following text into the textbox:

```
> Hello, I am a paragraph of text enclosed in a blockquote. Note I am offset
 from the left margin.
```

This renders as:

> Hello, I am a paragraph of text enclosed in a blockquote. Note I am offset from the left margin.

Links

Links can be written in two styles.

Inline links are written by enclosing the link text in square brackets first, then including the URL and optional alt-text in round brackets.

```
For more tutorials, please visit the [Programming
Historian](http://programminghistorian.org/ "Programming
Historian main page").
```

This renders as:

For more tutorials, please visit the Programming Historian.

Reference-style links are handy for footnotes and may keep your plain text document neater. These are written with an additional set of square brackets to establish a link ID label.

```
One example is the [Programming Historian][1] website.
```

You may then add the URL to another part of the document:

```
[1]: http://programminghistorian.org/ "The Programming
Historian"
```

This renders as:

One example is the Programming Historian website.

Images

Images can be referenced using !, followed by some alt-text in square brackets, followed by the image URL and an optional title. These will not be displayed in your plain text document, but would be embedded into a rendered HTML page.

```
![Wikipedia
logo](http://upload.wikimedia.org/wikipedia/en/8/80/Wikipedia-
logo-v2.svg "Wikipedia logo")
```

This renders as:

Wikipedia logo

Horizontal Rules

Horizontal rules are produced when three or more -, * or _ are included on a line by themselves, regardless of the number of spaces between them. All of the following combinations will render horizontal rules:

```
___
* * *
- - - - - -
```

This renders as:

Tables

The core Markdown spec does not include tables; however, some sites and applications use variants of Markdown that may include tables and other special features. GitHub Flavored Markdown[491] is one of these variants, and is used to render `.md` files in the browser on the GitHub site.

To create a table within GitHub, use pipes `|` to separate columns and hyphens `-` between your headings and the rest of the table content. While pipes are only strictly necessary between columns, you may use them on either side of your table for a more polished look. Cells can contain any length of content, and it is not necessary for pipes to be vertically aligned with each other.

```
| Heading 1 | Heading 2 | Heading 3 |
| --------- | --------- | --------- |
| Row 1, column 1 | Row 1, column 2 | Row 1, column 3|
| Row 2, column 1 | Row 2, column 2 | Row 2, column 3|
| Row 3, column 1 | Row 3, column 2 | Row 3, column 3|
```

This renders as:

Heading 1	Heading 2	Heading 3
Row 1, column 1	Row 1, column 2	Row 1, column 3
Row 2, column 1	Row 2, column 2	Row 2, column 3
Row 3, column 1	Row 3, column 2	Row 3, column 3

To specify the alignment of each column, colons `:` can be added to the header row as follows:

```
| Left-aligned | Centered | Right-aligned |
| :-------- | :-------: | --------: |
| Apples | Red | 5000 |
| Bananas | Yellow | 75 |
```

This renders as:

Left-aligned	Centered	Right-aligned
Apples	Red	5000
Bananas	Yellow	75

Markdown Limitations

While Markdown is becoming increasingly popular, particularly for styling documents that are viewable on the web, many people and publishers still expect traditional Word documents, PDFs, and other file formats. This can be mitigated somewhat with command line conversion tools such as

[491] 'Categories/ Writing on Github': https://help.github.com/categories/writing-on-github/

Pandoc;[492] however, certain word processor features like track changes are not supported yet. Please visit the Programming Historian lesson on 'Sustainable authorship in plain text using Pandoc and Markdown'[493] for more information about Pandoc.

Conclusion

Markdown is a useful middle ground between unstyled plain text files and legacy word processor documents. Its simple syntax is quick to learn and legible both by itself and when rendered into HTML and other document types. Finally, choosing to write your own documents in Markdown should mean that they will be usable and readable in the long-term.

About the Author

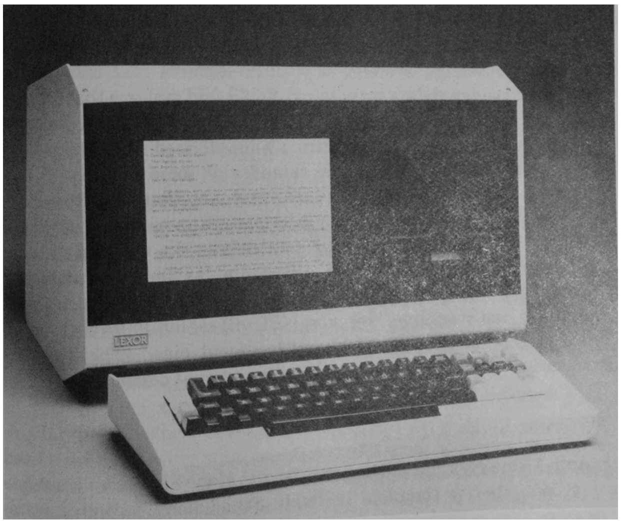
Sarah Simpkin is a GIS, Geography, and Computer Science librarian at the University of Ottawa.

[492] 'Pandoc': http://pandoc.org/

[493] Dennis Tenen and Grant Wythoff, 'Sustainable Authorship in plaint text using Pandoc and Markdown', *The Programming Historian* (2014).

48. Sustainable Authorship in Plain Text using Pandoc and Markdown

Dennis Tenen and Grant Wythoff – 2014



Objectives

In this tutorial, you will first learn the basics of Markdown—an easy to read and write markup syntax for plain text—as well as Pandoc,[494] a command line tool that converts plain text into a number of beautifully formatted file types: PDF, .docx, HTML, LaTeX, slide decks, and more. With Pandoc as your digital typesetting tool, you can use Markdown syntax to add figures, a bibliography, formatting, and easily change citation styles from Chicago to MLA (for instance), all using plain text.

The tutorial assumes no prior technical knowledge, but it scales with experience, as we often suggest more advanced techniques towards the end of each section. These are clearly marked and can be revisited after some practice and experimentation.

Instead of following this tutorial in a mechanical way, we recommend you strive to understand the solutions offered here as a *methodology*, which may need to be tailored further to fit your environment and workflow. The installation of the necessary tools presents perhaps the biggest barrier to participation. Allot yourself enough time and patience to install everything

[494] 'Pandoc': http://pandoc.org/

properly, or do it with a colleague who has a similar set-up and help each other out. Consult the 'Useful Resources' section below if you get stuck.

Philosophy

Writing, storing, and retrieving documents are activities central to the humanities research workflow. And yet, many authors base their practice on proprietary tools and formats that sometimes fall short of even the most basic requirements of scholarly writing. Perhaps you can relate to being frustrated by the fragility of footnotes, bibliographies, figures, and book drafts authored in Microsoft Word or Google Docs. Nevertheless, most journals still insist on submissions in .docx format.

More than causing personal frustration, this reliance on proprietary tools and formats has long-term negative implications for the academic community. In such an environment, journals must outsource typesetting, alienating authors from the material contexts of publication and adding further unnecessary barriers to the unfettered circulation of knowledge.

When you use MS Word, Google Docs, or Open Office to write documents, what you see is not what you get. Beneath the visible layer of words, sentences, and paragraphs lies a complicated layer of code understandable only to machines. Because of that hidden layer, your .docx and .pdf files depend on proprietary tools to be rendered correctly. Such documents are difficult to search, to print, and to convert into other file formats.

Moreover, time spent formatting your document in MS Word or Open Office is wasted, because all that formatting is removed by the publisher during submission. Both authors and publishers would benefit from exchanging files with minimal formatting, leaving the typesetting to the final typesetting stage of the publishing process.

This is where Markdown shines. Markdown is a syntax for marking semantic elements within a document explicitly, not in some hidden layer. The idea is to identify units that are meaningful to humans, like titles, sections, subsections, footnotes, and illustrations. At the very least, your files will always remain comprehensible to you, even if the editor you are currently using stops working or "goes out of business."

Writing in this way liberates the author from the tool. Markdown can be written in any plain text editor and offers a rich ecosystem of software that can render that text into beautiful looking documents. For this reason, Markdown is currently enjoying a period of growth, not just as as means for writing scholarly papers but as a convention for online editing in general.

Popular general purpose plain text editors include 'TextWrangler'[495] and 'Sublime' for Mac,[496] 'Notepad++' for Windows,[497] as well as 'Gedit' and

[495] 'TextWrangler': http://www.barebones.com/products/textwrangler/

[496] 'Sublime': http://www.sublimetext.com/

[497] 'Notepad++': https://notepad-plus-plus.org/

‘Kate’ for Linux. However, there are also editors that specialize in displaying and editing Markdown.

It is important to understand that Markdown is merely a convention. Markdown files are stored as plain text, further adding to the flexibility of the format. Plain text files have been around since the electronic typewriter. The longevity of this standard inherently makes plain text more sustainable and stable than proprietary formats. While files produced even ten years ago in Microsoft Word and Apple's Pages can cause significant problems when opened with the latest version, it is still possible to open a file written in any number of "dead" plain text editors from the past several decades: AlphaPlus, Perfect Writer, Text Wizard, Spellbinder, WordStar, or Isaac Asimov's favorite SCRIPSIT 2.0, made by Radio Shack. Writing in plain text guarantees that your files will remain readable ten, fifteen, twenty years from now. In this tutorial, we outline a workflow that frees the researcher from proprietary word processing software and fragile file formats.

It is now possible to write a wide range of documents in one format—articles, blog posts, wikis, syllabi, and recommendation letters—using the same set of tools and techniques to search, discover, backup, and distribute our materials. Your notes, blog entries, code documentation, and wikis can all be authored in Markdown. Increasingly, many platforms like WordPress, Reddit, and GitHub support Markdown authorship natively. In the long term, your research will benefit from such unified workflows, making it easier to save, search, share, and organize your materials.

Principles

Inspired by best practices in a variety of disciplines, we were guided by the following principles:

1. *Sustainability.* Plain text both ensures transparency and answers the standards of long-term preservation. MS Word may go the way of Word Perfect in the future, but plain text will always remain easy to read, catalog, mine, and transform. Furthermore, plain text enables easy and powerful versioning of the document, which is useful in collaboration and organizing drafts. Your plain text files will be accessible on cell phones, tablets, or, perhaps, on a low-powered terminal in some remote library. Plain text is backwards compatible and future-proof. Whatever software or hardware comes along next, it will be able to understand your plain text files.

2. *Preference for human-readable formats.* When writing in Word or Google Docs, what you see is not what you get. The .doc file contains hidden, automatically-generated formatting characters, creating an obfuscated typesetting layer that is difficult for the user to troubleshoot. Something as simple as pasting an image or text from the browser can have unpredictable effects on your document's formatting.

3. *Separation of form and content.* Writing and formatting at the same time is distracting. The idea is to write first, and format later, as close as possible to the time of publication. A task like switching from Chicago to MLA formatting should be painless. Journal editors who want to save time on needless formatting and copy editing should be able to provide their authors with a formatting template which takes care of the typesetting minutia.

4. *Support for the academic apparatus.* The workflow needs to handle footnotes, figures, international characters, and bibliographies gracefully.

5. *Platform independence.* As the vectors of publication multiply, we need to be able to generate a multiplicity of formats including for slide projection, print, web, and mobile. Ideally, we would like to be able to generate the most common formats without breaking bibliographic dependencies. Our workflow needs to be portable as well–it would be nice to be able to copy a folder to a thumbdrive and know that it contains everything needed for publication. Writing in plain text means you can easily share, edit, and archive your documents in virtually any environment. For example, a syllabus written in Markdown can be saved as a PDF, printed as a handout, and converted into HTML for the web, all from the same file. Both web and print documents should be published from the same source and look similar, preserving the logical layout of the material.

Markdown and LaTeX answer all of these requirements. We chose Markdown (and not LaTeX) because it offers the most light-weight and clutter free syntax (hence, mark *down*) and because when coupled with Pandoc it allows for the greatest flexibility in outputs (including .docx and .tex files).

Software Requirements

We purposefully omit some of the granular, platform- or operating system-bound details of installing the software listed below. For example, it makes no sense to provide installation instructions for LaTeX, when the canonical online instructions for your operating system will always remain more current and more complete. Similarly, the mechanics of Pandoc installation are best explored by searching for "installing Pandoc" on Google, with the likely first result being Pandoc's homepage.

- **Plain text editor**. Entering the world of plain-text editing expands your choice of innovative authoring tools dramatically. Search online for "markdown text editor" and experiment with your options. It does not matter what you use as long as it is explicitly a plain text editor. Notepad++ on Windows or TextWrangler on Macs are easy, free choices. Remember, since we are not tied to the tool, you can change editors at any time.

- **Command line terminal**. Working "in the command line" is equivalent to typing commands into the terminal. On a Mac you simply need to use your finder for "Terminal". On Windows, use PowerShell. Linux users are likely to be familiar with their terminals already. We will cover the basics of how to find and use the command line below.

- **Pandoc**. Detailed, platform-specific installation instructions are available at the Pandoc website.[498] *Installation of Pandoc on your machine is crucial for this tutorial*, so be sure to take your time and click through the instructions. Pandoc was created and is maintained by John MacFarlane, Professor of Philosophy at the University of California, Berkeley. This is humanities computing at its best and will serve as the engine of our workflow. With Pandoc, you will be able to compile text and bibliography into beautifully formatted and flexible documents. Once you've followed the installation instructions, verify that Pandoc is installed by entering `pandoc --version` into the command line. We assume that you have at least version 1.12.3, released in January 2014.

The following two pieces of software are recommended, but not required to complete this tutorial.

- **Zotero or Endnote**. Bibliographic reference software like Zotero and Endnote are indispensable tools for organizing and formatting citations in a research paper. These programs can export your libraries as a BibTeX file (which you will learn more about in Case 2 below). This file, itself a formatted plain text document of all your citations, will allow you to quickly and easily cite references using `@tags`. It should be noted that it's also possible to type all of your bibliographic references by hand, using our bibliography as a template.[499]

- **LaTeX**. Detailed, platform-specific installation instructions available at the Pandoc website.[500] Although LaTeX is not covered in this tutorial, it is used by Pandoc for .pdf creation. Advanced users will often convert into LaTeX directly to have more granular control over the typesetting of the .pdf. Beginners may want to consider skipping this step. Otherwise, type `latex -v` to see if LaTeX was installed correctly (you will get an error if it was not and some information on the version if it was).

Markdown Basics

Markdown is a convention for structuring your plain-text documents semantically. The idea is to identify logical structures in your document (a title, sections, subsections, footnotes, etc.), mark them with some

[498] 'Installing Pandoc': http://pandoc.org/installing.html

[499] 'pandoc-workflow / pandoctut.bib': https://github.com/dhcolumbia/pandoc-workflow/blob/master/pandoctut.bib

[500] 'Installing Pandoc': http://pandoc.org/installing.html

unobtrusive characters, and then "compile" the resulting text with a typesetting interpreter which will format the document consistently, according to a specified style.

Markdown conventions come in several "flavors" designed for use in particular contexts, such as blogs, wikis, or code repositories. The flavor of Markdown used by Pandoc is geared for academic use. Its conventions are described one the Pandoc's Markdown page. One of its conventions include the "YAML" block,[501] which contains some useful metadata.

Let's now create a simple document in Markdown. Open a plain-text editor of your choice and begin typing. It should look like this:

```
---
title: Plain Text Workflow
author: Dennis Tenen, Grant Wythoff
date: January 20, 2014
---
```

Pandoc-flavored Markdown stores each of the above values, and "prints" them in the appropriate location of your outputted document once you are ready to typeset. We will later learn to add other, more powerful fields to the YAML block. For now, let's pretend we are writing a paper that contains three sections, each subdivided into two subsections. Leave a blank line after last three dashes in the YAML block and paste the following:

```
# Section 1

## Subsection 1.1
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tem
por incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, qu
is nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo conseq
uat.

Next paragraph should start like this. Do not indent.

## Subsection 1.2
Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium d
oloremque laudantium, totam rem aperiam, eaque  ipsa quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta sunt explicabo.

# Section 2

## Subsection 2.1
```

Go ahead and enter some dummy text as well. Empty space is meaningful in Markdown: do not indent your paragraphs. Instead, separate paragraphs by using an blank line. Blank lines must also precede section headers.

[501] ‘Pandoc User’s Guide’: http://pandoc.org/README.html#yaml-metadata-block

You can use asterisks to add bold or italicized emphasis to your words, like this: `*italics*` and `**bold**`. We should also add a link and a footnote to our text to cover the basic components of an average paper. Type:

```
A sentence that needs a note.[^1]

[^1]: my first footnote! And a [link](https://www.eff.org/).
```

When the text of the link and the address are the same it is faster to write `<www.eff.org>` instead of `[www.eff.org](www.eff.org)`.

Let's save our file before advancing any further. Create a new folder that will house this project. You are likely to have some system of organizing your documents, projects, illustrations, and bibliographies. But often, your document, its illustrations, and bibliography live in different folders, which makes them hard to track. Our goal is to create a single folder for each project, with all relevant materials included. The general rule of thumb is one project, one paper, one folder. Name your file something like `main.md`, where "md" stands for markdown.

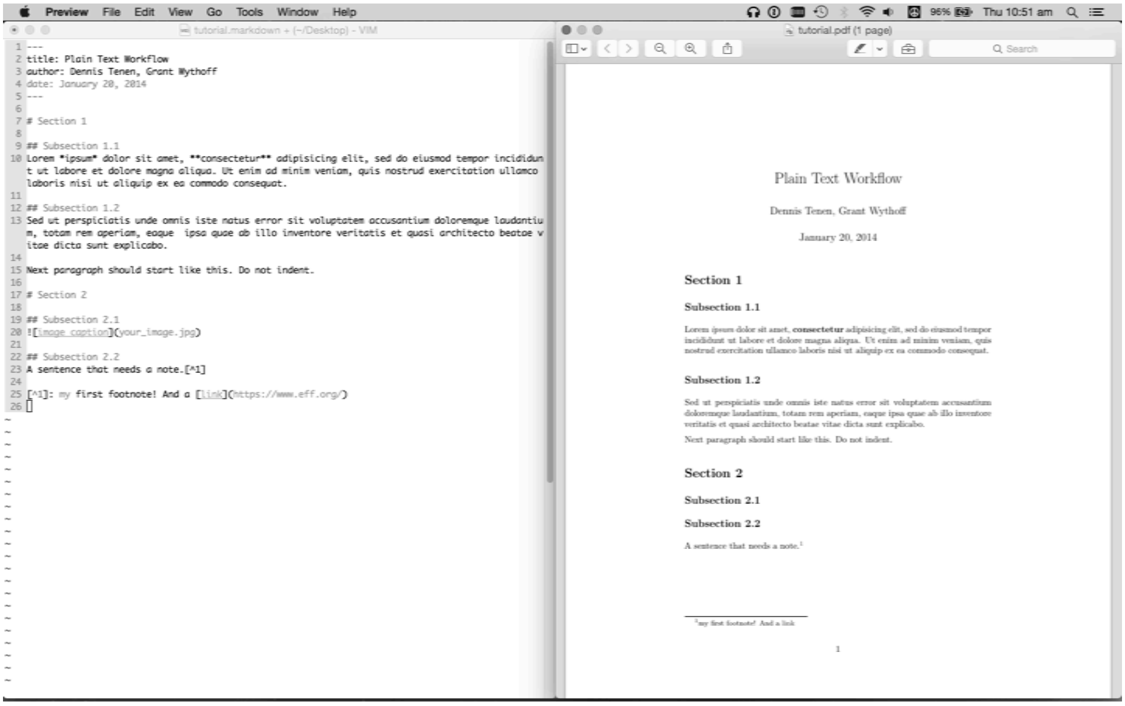
Once your file is saved, let's add an illustration. Copy an image (any small image) to your folder, and add the following somewhere in the body of the text: `![image caption](your_image.jpg)`.

At this point, your `main.md` should look something like the following. You can download this sample .md file: http://programminghistorian.org/assets/sample.md.

```
---
title: Plain Text Workflow
author: Dennis Tenen, Grant Wythoff
date: January 20, 2014
---

# Section 1

## Subsection 1.1
Lorem *ipsum* dolor sit amet, **consectetur** adipisicing elit, sed do eiusm
od tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veni
am, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat.

## Subsection 1.2
Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium d
oloremque laudantium, totam rem aperiam, eaque  ipsa quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta sunt explicabo.

Next paragraph should start like this. Do not indent.

# Section 2

## Subsection 2.1
![image caption](your_image.jpg)

## Subsection 2.2
```

```
A sentence that needs a note.[^1]

[^1]: my first footnote! And a [link](https://www.eff.org/)
```

As we shall do shortly, this plain text file can be rendered as a very nice PDF:



```
---
title: Plain Text Workflow
author: Dennis Tenen, Grant Wythoff
date: January 20, 2014
---

# Section 1

## Subsection 1.1
Lorem *ipsum* dolor sit amet, **consectetur** adipisicing elit, sed do eiusmod tempor incididun
t ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat.

## Subsection 1.2
Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantiu
m, totam rem aperiam, eaque  ipsa quae ab illo inventore veritatis et quasi architecto beatae v
itae dicta sunt explicabo.

Next paragraph should start like this. Do not indent.

# Section 2

## Subsection 2.1
![image caption](your_image.jpg)

## Subsection 2.2
A sentence that needs a note.[^1]

[^1]: my first footnote! And a [link](https://www.eff.org/)

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Plain Text Workflow

Dennis Tenen, Grant Wythoff

January 20, 2014

Section 1

Subsection 1.1

Lorem ipsum dolor sit amet, **consectetur** adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Subsection 1.2

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo.

Next paragraph should start like this. Do not indent.

Section 2

Subsection 2.1

Subsection 2.2

A sentence that needs a note.[1]

[1] my first footnote! And a link

1

Screenshot of a PDF rendered by Pandoc

If you'd like to get an idea of how this kind of markup will be interpreted as HTML formatting, try an online sandbox[502] and play around with various kinds of syntax. Remember that certain elements of *Pandoc*-flavored Markdown (such as the title block and footnotes) will not work in this web form, which only accepts the basics.

At this point, you should spend some time exploring some of other features of Markdown like quotations (referenced by > symbol), bullet lists which start with * or -, verbatim line breaks which start with | (useful for poetry), tables, and a few of the other functions listed on Pandoc's markdown page.

Pay particular attention to empty space and the flow of paragraphs. The documentation puts it succinctly when it defines a paragraph to be "one or more lines of text followed by one or more blank line." Note that "newlines are treated as spaces" and that "if you need a hard line break, put two or more spaces at the end of a line." The best way to understand what that means is to experiment freely. Use your editor's preview mode or just run Pandoc to see the results of your experiments.

[502] Available at: http://daringfireball.net/projects/markdown/dingus

Above all, avoid the urge to format. Remember that you are identifying *semantic* units: sections, subsections, emphasis, footnotes, and figures. Even `*italics*` and `**bold**` in Markdown are not really formatting marks, but indicate different level of *emphasis*. The formatting will happen later, once you know the venue and the requirements of publication.

There are programs that allow you to watch a live preview of Markdown output as you edit your plain text file, which we detail below in the Useful Resources section. Few of them support footnotes, figures, and bibliographies however. To take full advantage of Pandoc, we recommend that you stick with simple, plain text files stored locally, on your computer.

Getting in touch with your inner terminal

Before we can start publishing our `main.md` file into other formats, we need to get oriented with working on the command line using your computer's terminal program, which is the only (and best) way to use Pandoc.

The command line is a friendly place, once you get used to it. If you are already familiar with using the command line, feel free to skip this section. For others, it is important to understand that being able to use your terminal program directly will all you to use a broad range of powerful research tools that you couldn't use otherwise, and can serve as a basis for more advanced work. For the purposes of this tutorial, you need to learn only a few, very simple commands.

First, open a command line window. If you are using a Mac, open Terminal in the 'Applications/Utilities' directory. On Windows, you'll use PowerShell. On Windows 7 or later, click Start, type "powershell" in "Search programs and files," and hit enter. For a detailed introduction to using the command line, see Zed A. Shaw's excellent *Command Line Crash Course.*[503]

Once opened, you should see a text window and a prompt that looks something like this: `computer-name:~username$`. The tilde indicates your "home" directory, and in fact you can type `$ cd ~` at any point to return to your home directory. Don't type the dollar sign, it just symbolizes the command prompt of your terminal, promting you to type something into your terminal (as opposed to typing it into your document); remember to hit enter after every command.

It is very likely that your "Documents" folder is located here. Type `$ pwd` (= print working directory) and press enter to display the name of the current directory). Use `$ pwd` whenever you feel lost.

The command `$ ls` (= list), which simply lists the files in the current directory. Finally, you can use `$ cd>` (= change directory) like `$ cd DIRECTORY_NAME` (where `DIRECTORY_NAME` is the name of the directory you'd like to navigate to). You can use `$ cd ..` to automatically move up one

[503] 'The Command Line Crash Course': http://cli.learncodethehardway.org/book/

level in the directory structure (the parent directory of the directory you are currently in). Once you start typing the directory name, use the Tab key to auto complete the text—particularly useful for long directory names, or directories names that contain spaces.

These three terminal commands: `pwd`, `ls`, and `cd` are all you need for this tutorial. Practice them for a few minutes to navigate your documents folder and think about they way you have organized your files. If you'd like, follow along with your regular graphical file manager to keep your bearings.

Using Pandoc to convert Markdown to an MS Word document

We are now ready to typeset! Open your terminal window, use `$ pwd` and `$ cd` to navigate to the correct folder for your project. Once you are there, type `$ ls` in the terminal to list the files. If you see your .md file and your images, you are in the right place. To convert .md into .docx type:

```
$ pandoc -o main.docx main.md
```

Open the file with MS Word to check your results. Alternatively, if you use Open or Libre Office you can run:

```
$ pandoc -o project.odt main.md
```

If you are new to the command line, imagine reading the above command as saying something like: "Pandoc, create an MS Word file out of my Markdown file." The `-o` part is a "flag," which in this case says something like "instead of me explicitly telling you the source and the target file formats, just guess by looking at the file extension." Many options are available through such flags in Pandoc. You can see the complete list on Pandoc's website[504] or by typing

```
$ man pandoc
```

in the terminal.

Try running the command

```
$ pandoc -o project.html main.md
```

Now navigate back to your project directory. Can you tell what happened?

More advanced users who have LaTeX installed may want to experiment by converting Markdown into .tex or specially formatted .pdf files. Once LaTeX is installed, a beautifully formatted PDF file can be created using the same command structure:

```
$ pandoc -o main.pdf main.md
```

[504] 'Pandoc User's Guide': http://pandoc.org/README.html

With time, you will be able to fine tune the formatting of PDF documents by specifying a LaTeX style file (saved to the same directory), and running something like:

```
$ pandoc -H format.sty -o project.pdf --number-sections --toc project.tex
```

Working with Bibliographies

In this section, we will add a bibliography to our document and then convert from Chicago to MLA formats.

If you are not using a reference manger like Endnote or Zotero, you should. We prefer Zotero, because, like Pandoc, it was created by the academic community and like other open-source projects it is released under the GNU General Public License. Most importantly for us, your reference manager must have the ability to generate bibliographies in plain text format, to keep in line with our "everything in plain text" principle. Go ahead and open a reference manager of your choice and add some sample entries. When you are ready, find the option to export your bibliography in BibTeX (.bib) format. Save your .bib file in your project directory, and give it a reasonable title like "project.bib".

The general idea is to keep your sources organized under one centralized bibliographic database, while generating specific and much smaller .bib files that will live in the same directory as your project. Go ahead and open your .bib file with the plain-text editor of your choice.

Your .bib file should contain multiple entries that look something like this:

```
@article{fyfe_digital_2011,
    title = {Digital Pedagogy Unplugged},
    volume = {5},
    url = {http://digitalhumanities.org/dhq/vol/5/3/000106/000106.html},
    number = {3},
    urldate = {2013-09-28},
    author = {Fyfe, Paul},
    year = {2011},
    file = {fyfe_digital_pedagogy_unplugged_2011.pdf}
}
```

You will rarely have to edit these by hand (although you can). In most cases, you will simply "export" the .bib file from Zotero or from a similar reference manager. Take a moment to orient yourself here. Each entry consists of a document type, "article" in our case, a unique identifier (fyfe_digital_2011), and the relevant meta-data on title, volume, author, and so on. The thing we care most about is the unique ID which immediately follows the curly bracket in the first line of each entry. The unique ID is what allows us to connect the bibliography with the main document. Leave this file open for now and go back to your `main.md` file.

Edit the footnote in the first line of your `main.md` file to look something like the following examples, where `@name_title_date` can be replaced with one of the unique IDs from your `project.bib` file.

```
A reference formatted like this will render properly as inline-
or footnote- style citation [@name_title_date, 67].[^7](#fn:7)
"For citations within quotes, put the comma outside the
quotation mark" [@name_title_2011, 67].
```

Once we run the markdown through Pandoc, "@fyfe_digital_2011" will be expanded to a full citation in the style of your choice. You can use the `@citation` syntax in any way you see fit: in-line with your text or in the footnotes. To generate a bibliography simply include a section called `# Bibliography` at the end of document.

Now, go back to your metadata header at the top of your .md document, and specify the bibliography file to be used, like so:

```
---
title: Plain Text Workflow
author: Dennis Tenen, Grant Wythoff
date: January 20, 2014
bibliography: project.bib
---
```

This tells Pandoc to look for your bibliography in the `project.bib` file, under the same directory as your `main.md`. Let's see if this works. Save your file, switch to the terminal window and run:

```
$ pandoc -S -o main.docx --filter pandoc-citeproc main.md
```

The upper case `S` flag stands for "smart", a mode which produces "typographically correct output, converting straight quotes to curly quotes, — to em-dashes, — to en-dashes and … to ellipses." The "pandoc-citeproc" filter parses all of your citation tags. The result should be a decently formatted MS Word file. If you have LaTeX installed, convert into .pdf using the same syntax for prettier results. Do not worry if things are not exactly the way you like them—remember, you are going to fine-tune the formatting all at once and at later time, as close as possible to the time of publication. For now we are just creating drafts based on reasonable defaults.

Changing citation styles

The default citation style in Pandoc is Chicago author-date. We can specify a different style by using stylesheet, written in the "Citation Style Language" (yet another plain-text convention, in this case for describing citation styles) and denoted by the .csl file extension. Luckily, the CSL project maintains a repository of common citation styles, some even tailored for specific journals. Visit http://editor.citationstyles.org/about/ to find the .csl file for Modern Language Association, download `modern-`

`language-association.csl`, and save to your project directory as `mla.csl`. Now we need to tell Pandoc to use the MLA stylesheet instead of the default Chicago. We do this by updating the YAML header:

```
---
title: Plain Text Workflow
author: Dennis Tenen, Grant Wythoff
date: January 20, 2014
bibliography: project.bib
csl: mla.csl
---
```

You then simply use the same command:

```
$ pandoc -S -o main.docx --filter pandoc-citeproc main.md
```

Parse the command into English as you are typing. In my head, I translate the above into something like: "Pandoc, be smart about formatting, and output a Word Doc using the citation filter on my Markdown file (as you can guess from the extension)." As you get more familiar with citation stylesheets, consider adding your custom-tailored .csl files for journals in your field to the archive as a service to the community.

Summary

You should now be able to write papers in Markdown, to create drafts in multiple formats, to add bibliographies, and to easily change citation styles. A final look at the project directory will reveal a number of "source" files: your `main.md` file, `project.bib` file, your `mla.csl` file, and some images. Besides the source files you should see some some "target" files that we created during the tutorial: `main.docx` or `main.pdf`. Your folder should look something like this:

```
Pandoc-tutorial/
    main.md
    project.bib
    mla.csl
    image.jpg
    main.docx
```

Treat you source files as an authoritative version of your text, and you target files as disposable "print outs" that you can easily generate with Pandoc on the fly. All revisions should go into `main.md`. The `main.docx` file is there for final-stage clean up and formatting. For example, if the journal requires double-spaced manuscripts, you can quickly double-space in Open Office or Microsoft Word. But don't spend too much time formatting. Remember, it all gets stripped out when your manuscript goes to print. The time spent on needless formatting can be put to better use in polishing the prose of your draft.

Useful Resources

Should you run into trouble, there is no better place to start looking for support than John MacFarlane's Pandoc site[505] and the affiliated mailing list.[506] At least two "Question and Answer" type sites can field questions on 'Pandoc: Stack Overflow'[507] and 'Digital Humanities Q&A'.[508] Questions may also be asked live, on 'Freenode IRC', #Pandoc channel, frequented by a friendly group of regulars. As you learn more about Pandoc, you can also explore one of its most powerful features: filters.[509]

Although we suggest starting out with a simple editor, many (70+, according to Grace Smith's blog post)[510] other, Markdown-specific alternatives to MS Word are available online, and often free of cost. From the standalone ones, we liked 'Mou', 'Write Monkey', and 'Sublime Text'.[511] Several web-based platforms have recently emerged that provide slick, graphic interfaces for collaborative writing and version tracking using Markdown. These include: 'prose.io', 'Authorea', 'Penflip', 'Draft', and 'StackEdit'.[512]

But the ecosystem is not limited to editors. 'Gitit' and 'Ikiwiki'[513] support authoring in Markdown with Pandoc as parser. To this list we may a range of tools that generate fast, static webpages, 'Yst', 'Jekyll', 'Hakyll', and bash shell script by the historian Caleb McDaniel.[514]

Finally, whole publishing platforms are forming around the use of Markdown. Markdown to marketplace platform 'Leanpub'[515] could be an interesting alternative to the traditional publishing model. And we ourselves are experimenting with academic journal design based on GitHub and readthedocs.org (tools usually used for technical documentation).

Don't worry if you don't understand some of of this terminology yet!

[505] 'About Pandoc': http://pandoc.org/

[506] 'pandoc-discuss': https://groups.google.com/forum/#!forum/pandoc-discuss

[507] 'Stack Overflow: Pandoc': http://stackoverflow.com/questions/tagged/pandoc

[508] 'Digital Humanities Questions and Answers': http://digitalhumanities.org/answers/

[509] 'Pandoc Filters': https://github.com/jgm/pandoc/wiki/Pandoc-Filters

[510] Grace Smith, '78 Tools for Writing and Previewing Markdown', *Mashable* (24 June 2013): http://web.archive.org/web/20140120195538/http://mashable.com/2013/06/24/markdown-tools/.

[511] 'Mou': http://25.io/mou/; 'WriteMonkey': http://writemonkey.com/; 'Sublime Text': http://www.sublimetext.com/.

[512] 'Prose.io': http://prose.io/; 'Authorea': https://www.authorea.com/; 'Penflip': https://www.penflip.com/; 'Draft': https://draftin.com/; 'StackEdit': https://stackedit.io/

[513] 'Gitit': http://gitit.net/; 'Ikiwiki': https://github.com/dubiousjim/pandoc-iki

[514] 'Yst': https://github.com/jgm/yst; 'Jekyll': https://github.com/fauno/jekyll-pandoc-multiple-formats; 'Hakyll': https://jaspervdj.be/hakyll/; 'wcaleb / website': https://github.com/wcaleb/website

[515] 'Leanpub': https://leanpub.com/

The source files for this document can be downloaded from GitHub.[516] Use the "raw" option when viewing in GitHub to see the source Markdown. The authors would like to thank Alex Gil and his colleagues from Columbia's Digital Humanities Center, and the participants of openLab at the Studio in the Butler library for testing the code in this tutorial on a variety of platforms.

See Charlie Stross's excellent discussion of this topic in 'Why Microsoft Word Must Die'.[517]

Note that the .bib extension may be "registered" to Zotero in your operating system. That means when you click on a .bib file it is likely that Zotero will be called to open it, whereas we want to open it within a text editor. Eventually, you may want to associate the .bib extension with your text editor.

There are no good solutions for directly arriving at MS Word from LaTeX.

It is a good idea to get into the habit of not using spaces in folder or file names. Dashes-or_underscores instead of spaces in your filenames ensure lasting cross-platform compatibility.

Thanks to @nickbart80 (https://github.com/nickbart1980) for the correction. In response to our original suggestion, `Some sentence that needs citation.^[@fyfe_digital_2011 argues that too.]` he writes: "This is not recommended since it keeps you from switching easily between footnote and author-date styles. Better use the [corrected] (no circumflex, no final period inside the square braces, and the final punctuation of the text sentence after the square braces; with footnote styles, pandoc automatically adjusts the position of the final punctuation)."[518]

About the Authors

Dennis Tenen is an assistant professor of English and Comparative Literature at Columbia University. Grant Wythoff is a lecturer in the Department of English and Comparative Literature at Columbia University.

[516] 'dhcolumbia / pandoc-workflow': https://github.com/dhcolumbia/pandoc-workflow

[517] Charlie Stross, 'Why Microsoft Word must Die', *Charlie's Diary* (12 October 2013): http://www.antipope.org/charlie/blog-static/2013/10/why-microsoft-word-must-die.html

[518] https://github.com/programminghistorian/jekyll/issues/46#issuecomment-59219906

Epilogue

This book represents a snapshot of the *Programming Historian* project as it was in early February 2016. The very process of putting it together has given the editorial team a chance to check-over lessons to find problems that have inevitably crept into the site over the years. Links break, technologies become unsupported, screenshots cease to be representative.

It's also given us a chance to reflect on ways that we can improve the learning experience for our readers. We're constantly changing, always trying to find better ways to be the go-to place for digital history pedagogy.

With that in mind, we're always looking to hear from potential contributors. That might be authors, reviewers, or people who want to chip in and help us make *The Programming Historian* a little bit better.

We're building this for all of us, and we'd encourage you to look us up and lend a hand.

http://programminghistorian.org

February 2016

London

Adam Crymble, on behalf of the Editorial Board

Editorial Board