

# The Programming Historian

The Programming Historian is an open-access introduction to programming in Python, aimed at working historians (and other humanists) with little previous experience. There are two editions available here; the second is currently under development. We are constantly adding new material, much of it driven by reader request. We welcome questions, corrections and suggestions for improvement. At this point we are still figuring out how best to allow community participation, while maintaining the coherence and direction of a more monographic work. If you e-mail us at [wturkel@uwo.ca](mailto:wturkel@uwo.ca), [acrymbl@uwo.ca](mailto:acrymbl@uwo.ca) and/or [amaceach@uwo.ca](mailto:amaceach@uwo.ca), we are happy to respond to you personally and try to incorporate your comments. In the future we may come up with something more elegant... but, hey, it's a work in progress.

- William J. Turkel, Adam Crymble and Alan MacEachern, *The Programming Historian*, 2nd ed. NiCHE: Network in Canadian History & Environment (2009-).
- William J. Turkel and Alan MacEachern, *The Programming Historian*, 1st ed. NiCHE: Network in Canadian History & Environment (2007-08).

Introductory lessons teach you how to

- install Zotero, the Python programming language and other useful tools
- read and write data files
- save web pages and automatically extract information from them
- count word frequencies
- remove stop words
- automatically refine searches
- make n-gram dictionaries
- create keyword-in-context (KWIC) displays
- make tag clouds, and
- harvest sets of hyperlinks

## Table of Contents

0. About this book.....	3
1. Do you need to learn how to program?.....	4
Techniques that don't involve programming.....	4
Why you might want to learn to program.....	4
What kind of techniques you will learn.....	5
2. Getting started.....	5
Install and set up software.....	5
Linux instructions.....	6
Mac instructions.....	7
Windows instructions.....	8
"Hello world" in Python.....	9
Interacting with a Python shell.....	9
Linux instructions.....	9
Mac instructions.....	9
Windows instructions.....	10
"Hello world" in JavaScript.....	11
Viewing HTML files.....	11
"Hello World" in HTML.....	12

"Hello World" in embedded JavaScript.....	13
Back up your work.....	13
Keep in touch with us.....	13
Other resources.....	14
Suggested readings.....	14
3. Working with files and web pages.....	14
Making use of your ability to do close reading.....	14
Sending information to text files.....	15
Getting information from text files.....	15
Splitting code into modules and functions.....	16
About URLs.....	17
Opening URLs with Python.....	18
Saving a local copy of a web page.....	19
Suggested Readings.....	20
4. From HTML to a list of words.....	20
Getting rid of HTML formatting.....	20
More about Python strings.....	20
Looping.....	22
Branching.....	22
The stripTags routine.....	23
Python lists.....	23
Suggested Readings.....	25
5. Computing frequencies.....	25
Useful measures of a text.....	25
Cleaning up the list.....	25
Our first use of regular expressions.....	26
Python dictionaries.....	27
Counting word frequencies.....	28
From HTML to a dictionary of word-frequency pairs.....	29
Removing stop words.....	30
Putting it all together.....	31
Suggested Readings.....	32
6. Wrapping output in HTML.....	32
Putting new information where you can use it.....	32
Python string formatting.....	33
Creating HTML output.....	33
Sending HTML output to Firefox.....	34
Self-documenting data files.....	34
Python comments.....	35
Building an HTML wrapper.....	35
Putting it all together.....	36
Using word frequencies to refine a Google search.....	37
Suggested Readings.....	38
7. Keyword in context (KWIC).....	38
N-grams.....	38
From text to n-grams.....	39
Making an n-gram dictionary.....	40
Pretty printing a KWIC.....	40
From HTML to KWIC.....	42
Turning each KWIC into a Google search link.....	43
8. Tag clouds.....	44
Visualizing term frequency.....	44
Mapping one range onto another.....	44

A little bit of CSS.....	45
Functions to write HTML divs and spans.....	46
Other dimensions for visualization.....	47
Putting it all together.....	48
Combining the tag cloud with KWIC.....	49
9. Harvesting links and downloading pages.....	51
The idea of text mining.....	51
Selecting a group of biographies.....	51
Extracting hyperlinks with Beautiful Soup.....	52
Scraping with regular expressions.....	53
Working with accented characters.....	54
Some helper functions.....	55
Putting it all together.....	56
10. Indexing a document collection.....	58
An overview.....	58
Getting a list of filenames from a directory.....	59
Normalizing the files.....	59
Mapping an anonymous function over a list.....	60
Replacing stopwords with a placeholder.....	61
Zip and tuples.....	62
Putting it all together.....	63
Suggested Readings.....	64
Discussion of The Programming Historian, 1st ed.....	64
Do you need to learn how to program?.....	64
Getting started.....	64
Working with files and web pages.....	66
From HTML to a list of words.....	67
Computing frequencies.....	67
Wrapping output in HTML.....	68
Keyword in context (KWIC).....	68
Tag clouds.....	69
Peer Reviewers.....	69

## 0. About this book

This book is a tutorial-style introduction to programming for practicing historians. We assume that you're starting out with no prior programming experience and only a basic understanding of computers. More experience, of course, won't hurt. Once you know how to program, you will find it relatively easy to learn new programming languages and techniques, and to apply what you know in unfamiliar situations. In order to get you to that point we've adopted the following strategy.

- You should be able to put what you learn to work in your research immediately. We think that many beginning programmers lose patience because they can't see why they're learning what they're learning.
- Digital history requires working with sources on the web. This means that you're going to be spending most of your research time working in a browser, so you should be able to put your programming skills to work there.
- You will have to be somewhat [polyglot](#). Individual programming languages can be beautiful objects in their own right, and each embodies a different way of looking at the world. In order to become a good programmer, you will eventually have to master the intricacies of one or more particular languages. When you're first getting started, however, you need something more like a pidgin.

- Open source and open access are both good things. We're providing open access to this book. As we develop it, we'll be searching for ways to best incorporate the peer review and continual improvement that characterize open source projects. We also build our work on top of other open source projects, particularly [Python](#), [Firefox](#), [Zotero](#) and the [Simile](#) tools.

We both do archival work, write monographs and journal articles, and teach undergraduate and graduate courses in history. Our backgrounds are a bit different: although we're the same age, one of us has been programming for about 30 years (WJT) whereas the other started on 1 January 2008 (AM). We share the conviction, however, that digital history represents the future of our discipline.

To some extent, this book is an extended conversation about the degree to which future historians will need to be able to program in order to do their jobs. We also hope, of course, that if you work through the book you'll learn techniques that make you a better historian.

## ***1. Do you need to learn how to program?***

### **Techniques that don't involve programming**

Do you need to be able to program? The short answer is "maybe not." You can certainly become more effective at online research with a few simple techniques that don't require any programming.

- *Citation management.* Install [Zotero](#) and learn how to use it. Make sure to [backup your Zotero database](#) regularly.
- *Searching.* Always use the advanced search interface when working with search engines. Learn whatever specialized search syntax is available, and check periodically to see if features have changed. You should know, for example, that [Google](#) lets you search for exact phrases or for words in any order; that it lets you exclude words; that it can limit your search to a particular domain or help you find the pages that link to a page you're interested in. You should also know that there are separate Google searches for [books](#), [images](#), [historic news articles](#), [code](#) and [scholarly articles](#) among many other things.
- *Information Trapping.* Think of a search as something that you do once. When you find what you're looking for, you stop searching. You may bookmark a website, but you have to return to it explicitly whenever you want to see if something has changed. There are some kinds of information that you need to monitor on a more regular basis. In these cases, it makes more sense to subscribe to regularly-updated RSS feeds. See Tara Calishain's [Information Trapping](#) for more detail.

### **Why you might want to learn to program**

We think that at least some historians really will need to learn how to program. Think of it like learning how to cook. You may prefer fresh pasta to boxed macaroni and cheese, but if you don't want to be stuck eating the latter, you have to learn to cook or pay someone else to do it for you. Learning how to program is like learning to cook in another way: it can be a very gradual process. One day you're sitting there eating your macaroni and cheese and you decide to liven it up with a bit of Tabasco, Dijon mustard or Worcestershire sauce. Bingo! Soon you're putting grated cheddar in, too. You discover that the ingredients that you bought for one dish can be remixed to make another. You begin to linger in the spice aisle at the grocery store. People start buying you cookware. You get to the point where you're willing and able to experiment with recipes. Although few people become master chefs, many learn to cook well enough to meet their own needs.

If you don't program, your research process will always be at the mercy of those who do.

At this point you might object that some of your primary sources are not in digital form and won't be for the foreseeable future. We get this. We're not suggesting that historians no longer need to know how to use material sources in real archives. What we're suggesting is that the rest of your scholarly life has already gone digital. You communicate electronically using e-mail and mailing lists; you search library catalogs and archival finding aids online; you submit drafts of monographs and articles electronically; you present yourself to the world on one or more websites; you have to put up lecture notes or submit grades online; an awful lot of the information that you need daily is already on the web. To use another food metaphor, imagine that digital sources are like sugar (and who wouldn't like to think of them that way?) In medieval Europe, sugar was a rare and expensive spice. Although some people might know how to use it in a dish, most people didn't ever need to think about it. Fast forward to the late 19th century, when sugar made up a relatively large proportion of many European diets. Not everyone needed to know how to make dessert, but it was no longer a rare skill. In the 21st century, some forms of sugar (e.g., high-fructose corn syrup) have become very difficult to avoid.

## **What kind of techniques you will learn**

Many books about programming fall into one of two categories: (1) books about particular programming languages, and (2) books about computer science that demonstrate abstract ideas using a particular programming language. When you're first getting started, it's easy to lose patience with both of these kinds of books. On the one hand, a systematic tour of the features of a given language and the style(s) of programming that it supports can seem rather remote from the tasks that you'd like to accomplish. On the other hand, you may find it hard to see how the abstractions of computer science are related to your specific application. Once you know how to program, of course, both kinds of book are very useful. You can use books about programming languages as references, or to transfer your knowledge of one language to another. And you can use computer science books as a source of inspiration and deeper understanding.

Our goal is to introduce programming techniques that will be immediately useful in your work as a (digital) historian. Although we will provide links to programming language reference books and computer science texts as necessary, we won't be concerned with giving you a full tour of any particular programming language or a systematic introduction to the [algorithms](#) and [data structures](#) of introductory computer science.

We're going to assume that you are connected to the web, and that there are a vast number of online primary and secondary sources that are relevant to your research, if only you could find and make use of them. We will start by developing techniques to find new textual sources, download batches of them, convert them from one format to another, characterize them individually and cluster them automatically into useful groups.

Programming is for digital historians what sketching is for artists or architects: a mode of creative expression and a means of exploration.

## ***2. Getting started***

### **Install and set up software**

In order to work through the techniques in this book, you will need to download and install some freely available software. As much as possible, we've tried to make everything compatible with Linux, Mac and Windows PCs. We assume that the majority of our readers will probably be using Windows, so we've taken the approach of getting a Windows XP version working first, then a Mac version and finally a Linux version. We'd be happy to include instructions for specific platforms, especially if you want to send them to us. We've also included peer feedback and commentary on the discussion page. If you run into trouble with our

instructions or find something that doesn't work on your platform, please let us know. Since this is very much a work-in-progress, **we will occasionally make comments and indicate things that are provisional in purple.**

## Linux instructions

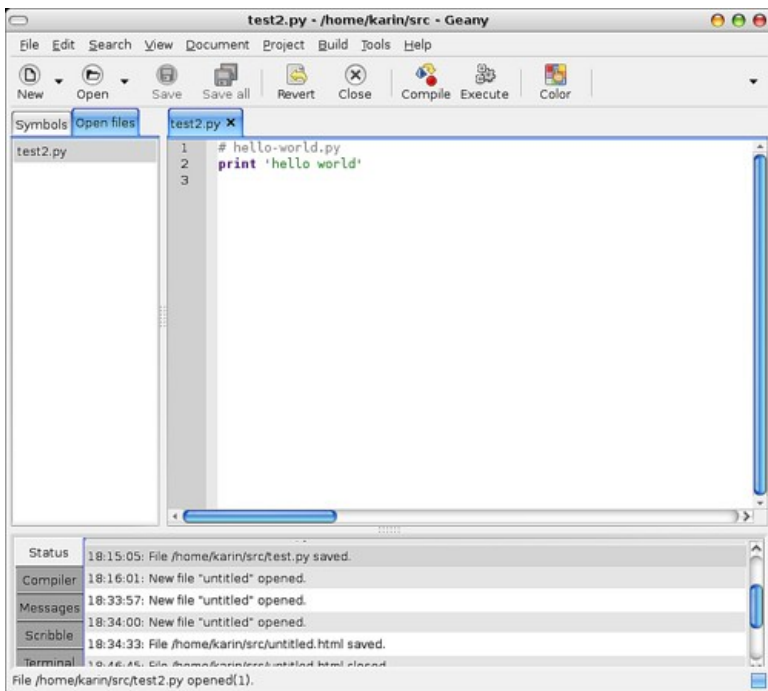
- Thanks to Karin Dalziel! For more info, read the latest version of her [notes](#).
- These instructions are for [Ubuntu 7.10 "Gutsy Gibbon"](#). **When these instructions were written, Zotero was not yet compatible with Firefox 3. Since it now is, you can probably work with a later version of Ubuntu. We welcome feedback on this.**
- **Back up your computer**
- Install the following Firefox extensions:
  - [Web developer toolbar](#)
  - [Extension Developer's Extension](#) **If you are using Firefox 3 you can't install this extension for security reasons. Skip it for now.**
  - If you are not already using it, install [Zotero](#).
- To install [Python](#):
  - Click on "system" (upper left of the toolbar) -> Administration -> Synaptic Package Manager
  - Go to "Settings" -> "Repositories" and make sure all the boxes are checked under the "Ubuntu software" tab
  - Enter in your password
  - Search for "Python" or "Python2.5" (searching just for "Python" helps find the most recent packages, and you can see other useful Python related packages).
  - Check the packages "python" and "python2.5" (or whatever the latest number is). You might want to add "python2.5-doc" and "python2.5-examples" too.
  - Note, Python is already installed for some (all?) Ubuntu installations.
  - Create a directory where you will keep your Python programs. One option is to name it "src" and put it in your home folder (/home/username/src/)
- Again, through synaptic, install the package "python-beautifulsoup"
- As with the Mac and PC versions, you can install the program [Komodo Edit](#). Just go to the website, download the Linux version, double click the file to decompress it, and then read the installation instructions for Linux.
- Start 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
- Now you need to set up the editor so that you can run Python programs
- Choose Toolbox->Add->New Command. This will open a new dialog window. Rename your command to "Run Python". Under "Command," use the pulldown menu to select

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

- and under "Start in," enter

```
%D
```

- Click OK. Your new Run Python command should appear in the Toolbox pane
- Alternately, you can use Geany, an integrated development environment available through the Synaptic Package manager. **The instructions throughout the tutorials will be slightly different if you do this.**
- If you use Geany, instead of the "Run Python" button, you will save your file as "filename.py" and then click the "execute" button at the top instead.



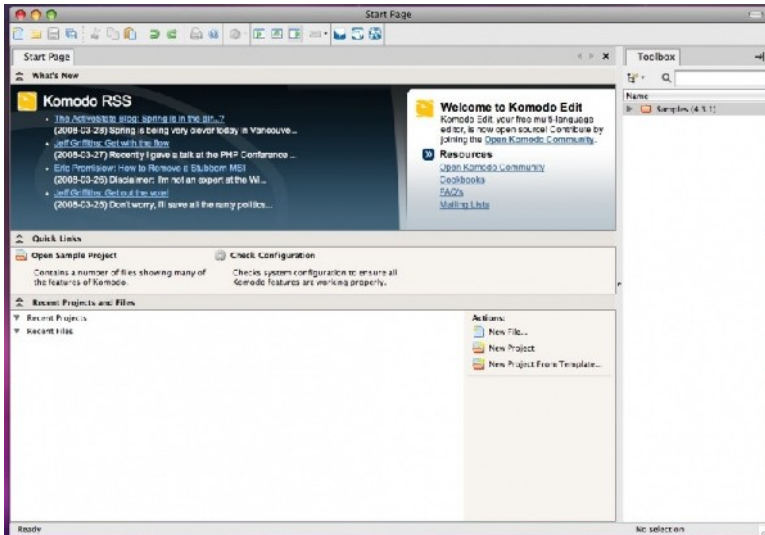
- When you run a program it will look like this:



### Mac instructions

- **Back up your computer**
- If you are not already using it, install the [Firefox](#) web browser
- Install the following Firefox extensions:
  - [Web developer toolbar](#)
  - [Extension Developer's Extension](#) If you are using Firefox 3 you can't install this extension for security reasons. Skip it for now.
  - If you are not already using it, install [Zotero](#)
- Go to the [Python website](#), download the latest stable release of the Python programming language (Version 2.5.2 as of Mar 2008) and install it
  - The OS X installation makes use of a .DMG (Disk Image) file. When this file has finished downloading to your machine, you can double click it to open a folder that contains a ReadMe.txt file and a MacPython installer
  - Double click the MacPython.mpkg file to start the universal installer

- Create a directory where you will keep your Python programs (e.g., programming-historian)
- Download the latest version of [Beautiful Soup](#) and copy it to the directory where you are going to put your own programs
- Although MacPython includes an integrated development environment, we will be using a free and open source editor called [Komodo Edit](#). Install it from the .DMG file
- Start Komodo. It should look something like this



- 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
- Now you need to set up the editor so that you can run Python programs
- Choose Toolbox->Add->New Command. This will open a new dialog window. Rename your command to "Run Python". Under "Command," use the pulldown menu to select

`%(python) %f`

- and under "Start in," enter

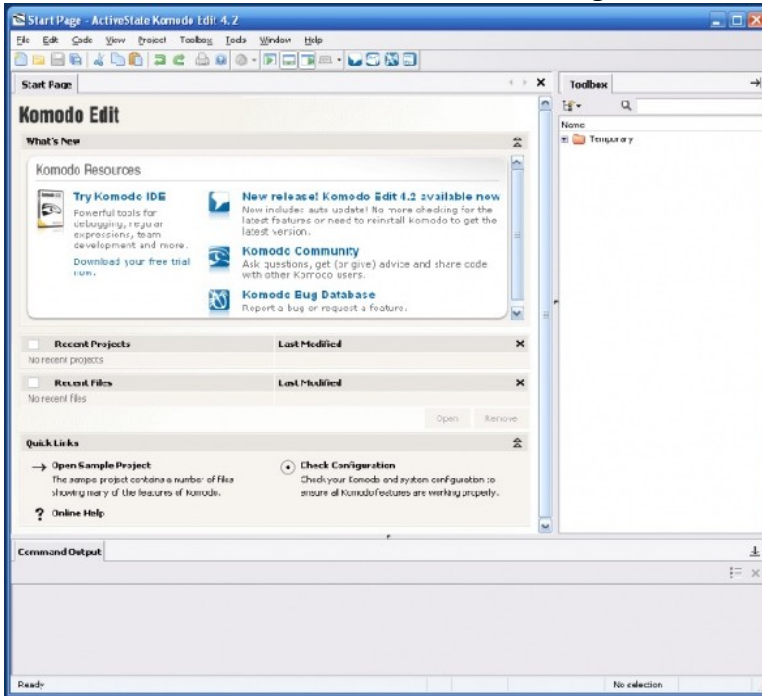
`%D`

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

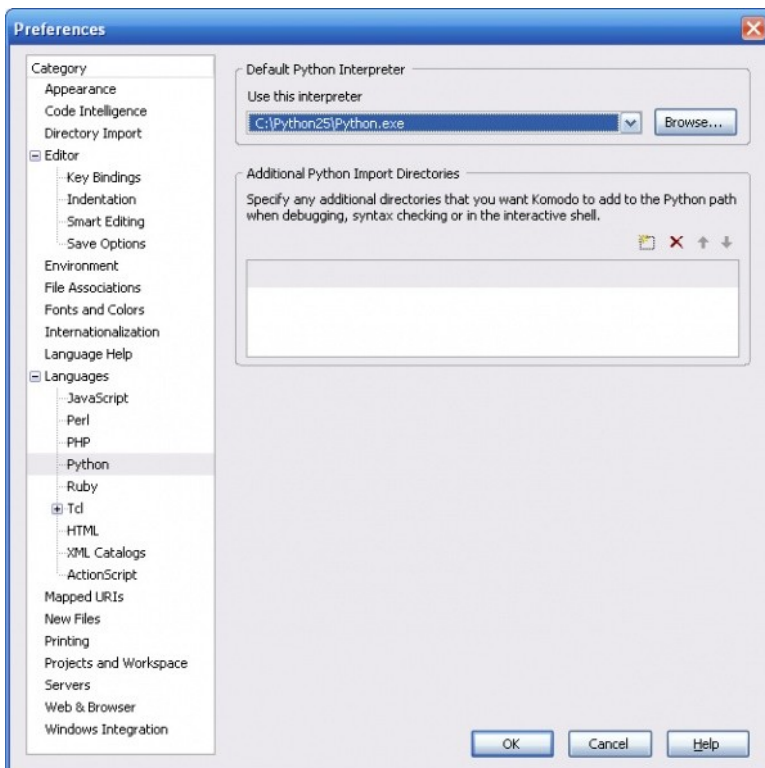
### *Windows instructions*

- **Back up your computer**
- If you are not already using it, install the [Firefox](#) web browser
- Install the following Firefox extensions:
  - [Web developer toolbar](#)
  - [Extension Developer's Extension](#) If you are using Firefox 3 you can't install this extension for security reasons. Skip it for now.
  - If you are not already using it, install [Zotero](#)
- Go to the [Python website](#), download the latest stable release of the Python programming language (Version 2.5.2 as of April 2008) and install it
- Download the latest version of [Beautiful Soup](#) and copy it to the Python library directory (usually C:\Python25\Lib)

- Install [Komodo Edit](#)
- Start Komodo. It should look something like this



- 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
- Now you need to set up the editor so that you can run Python programs
- Choose Edit->Preferences. This will open a new dialog window.
- Select the Python category and set the "Default Python Interpreter" (it should be C:\Python25\Python.exe)
- If it looks like this, click OK:



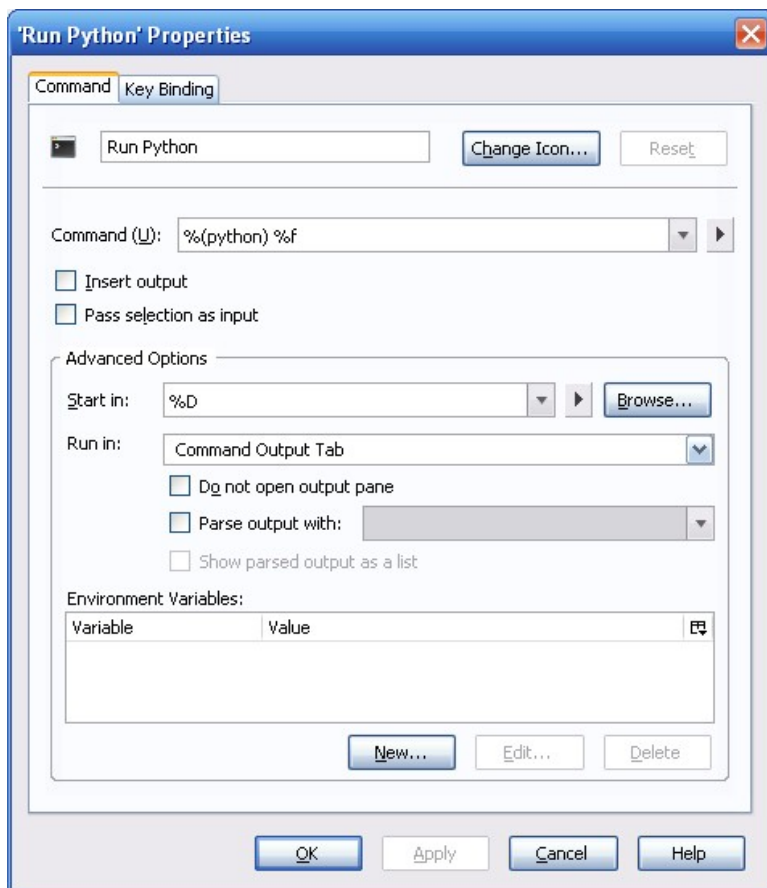
- Next choose Toolbox->Add->New Command. This will open a new dialog window. Rename your command to "Run Python". Under "Command," use the pulldown menu to select

`%(python) %f`

- and under "Start in," enter

`%D`

- N.B. If you forget the `%f` in the first command, Python will hang mysteriously because it isn't receiving a program as input
- If it looks like this, click OK:



- Your new command should appear in the Toolbox pane
- N.B. Some people have reported that you have to restart your machine before Python will work with Komodo Edit

## "Hello world" in Python

It is traditional to begin programming in a new environment by trying to create a program that says "hello world" and terminates. In keeping with our polyglot approach, we will do this in a number of different ways using a few different programming languages.

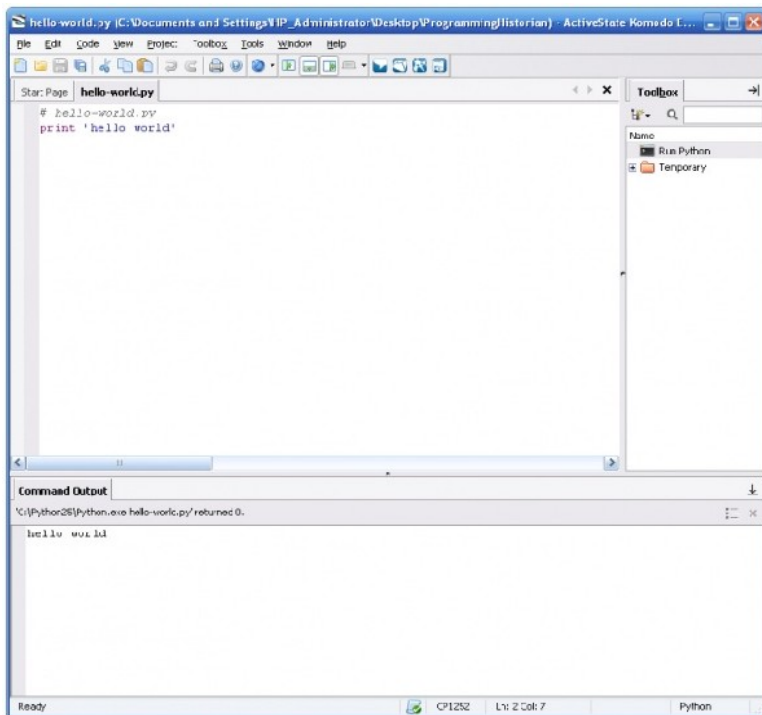
The languages that we will be using are all interpreted. This means that there is a special computer program

(known as an interpreter) that knows how to follow instructions written in the 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 Komodo, create a new file, enter the following two-line program and save it as hello-world.py

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

You should then be able to double-click the "Run Python" button that you created in the previous step to execute your program. If all went well, it should look something like this:



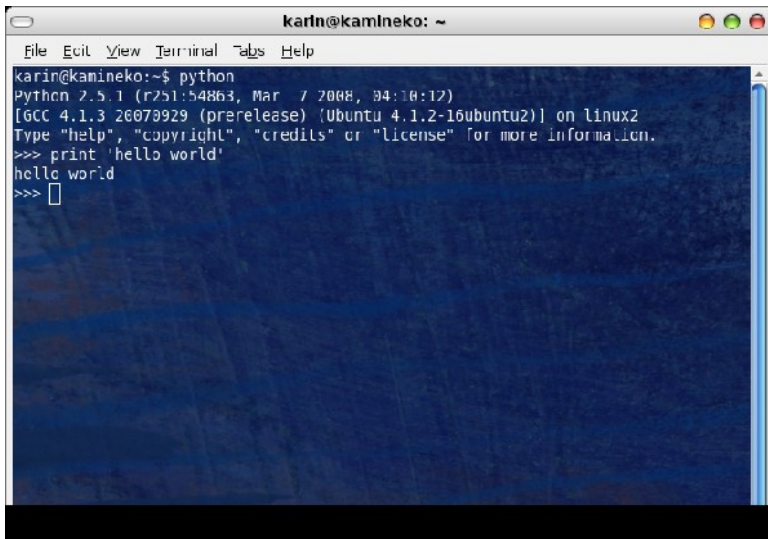
Notice that the output of your program was printed to the "Command Output" pane.

## 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 interpreter 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.

### *Linux instructions*

Linux instructions are pretty much the same as Mac. Just go to Applications (again, upper left of toolbar) -> Accessories -> terminal



```
karn@kamineko: ~  
File Edit View Terminal Tabs Help  
karn@kamineko:~$ python  
Python 2.5.1 (r251:54863, Mar 7 2008, 04:10:12)  
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print 'hello world'  
hello world  
>>> █
```

### *Mac instructions*

You can run a Python interpreter by going to the Finder and double-clicking on Applications->Utilities->Terminal then typing "python" into the window that opens on your screen. At the Python interpreter prompt, type

```
print 'hello world'
```

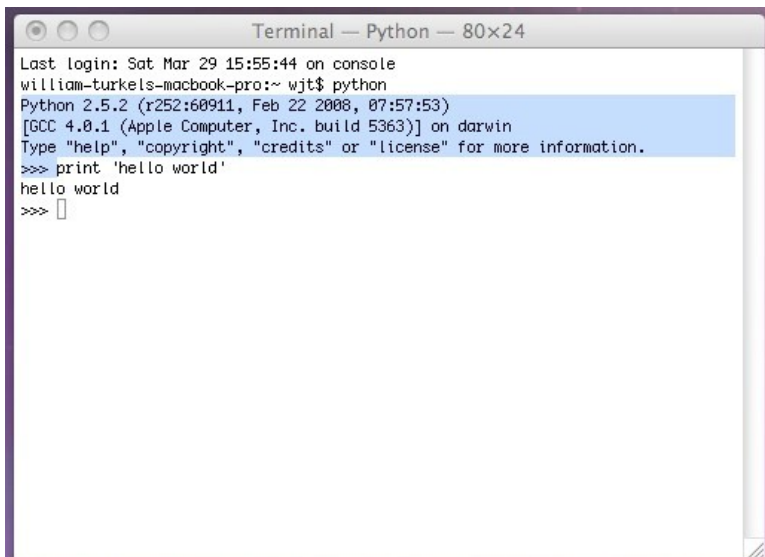
and press Enter. The computer will respond with

```
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:



```
Terminal — Python — 80x24  
Last login: Sat Mar 29 15:55:44 on console  
william-turkels-macbook-pro:~ wjt$ python  
Python 2.5.2 (r252:60911, Feb 22 2008, 07:57:53)  
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print 'hello world'  
hello world  
>>> █
```

## Windows instructions

You can get access to a Python shell by double-clicking on C:\Python25\python.exe. A new window will open on your screen. In the shell window, type

```
print 'hello world'
```

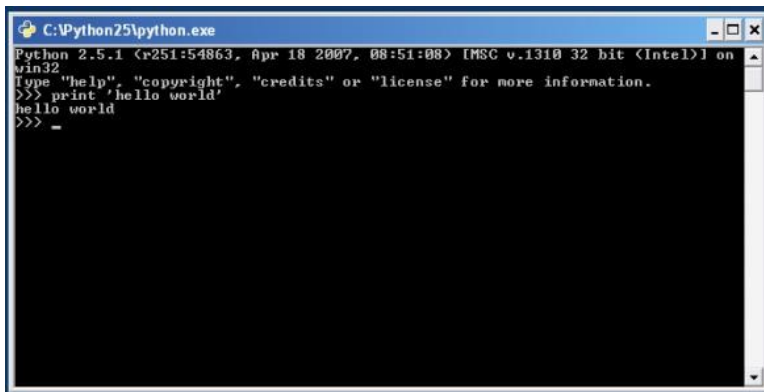
and press Enter. The computer will respond with

```
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 like this:

A screenshot of a Windows command prompt window titled "C:\Python25\python.exe". The window shows the Python 2.5.1 shell interface. The text inside the window is: Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32. Type "help", "copyright", "credits" or "license" for more information. >>> print 'hello world' hello world >>> \_

The reason that we will be using Python for many of our programming tasks is that 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.

## "Hello world" in JavaScript

A second programming language that we will be using is JavaScript. Like Python, JavaScript is an interpreted language. One of the things that makes JavaScript special is that the browser is a JavaScript interpreter. So it is possible to write programs that control the behavior of your browser. In fact, that is what Zotero is, a program written (mostly) in JavaScript that adds some powerful functionality to Firefox.

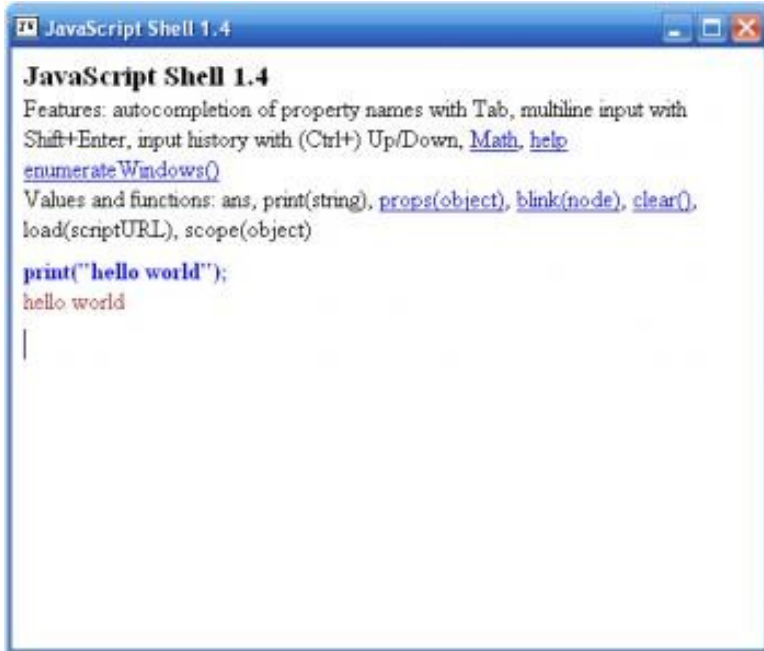
Being able to program the browser makes it possible to do many interesting things, but it also introduces some important limitations. Imagine if someone else were able to use JavaScript to program your browser so that it erased all of the files on your hard drive? Not good. For this reason, the JavaScript language has no mechanisms for creating, opening, or deleting files. The language also prevents information from being exchanged outside of well-defined and fairly limited boundaries.

Hence our polyglot approach. For some tasks, we will want to use Python, for others, JavaScript. Sometimes we will mix code from both languages to get the best results. Most of the work that we do at the beginning will be in Python, however.

In Firefox, choose Tools->Extension Developer->Javascript Shell. A window should open on your screen. In that window type the following statements and press Enter.

```
print("hello world");
```

If all went well, it should look something like this:

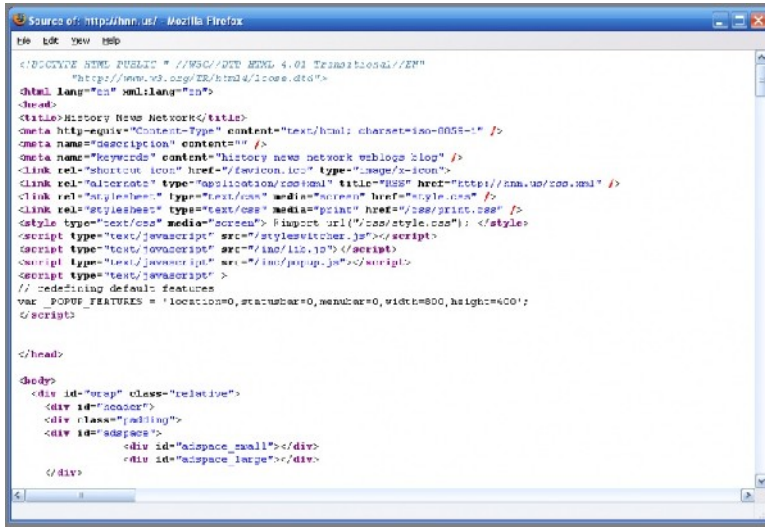


## Viewing HTML files

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 for any page that you visit. The two images below show a typical web page (the History News Network) and the HTML source used to generate that page, which you can see with the View->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 study HTML source as you write programs to manipulate web pages or automatically extract information from them.





(To learn more about HTML, you may find it useful at this point to work through the W3 Schools [HTML tutorial](#). Detailed knowledge of HTML isn't necessary to continue reading, but any time that you spend learning HTML will be amply rewarded in your work as a digital historian.)

## "Hello World" in HTML

HTML consists of text and tags which typically indicate the beginning and ending of particular elements. 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

... 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 Komodo, and choose File->New. Copy the code below into the editor. The first line tells the browser what kind of file it is. The *html* tag has the lang property (for language) set to en (for 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 PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
<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 as *hello-world.html*. Now go to Firefox and choose File->New Tab and then File->Open File. Choose *hello-world.html*. Your message should appear in the browser.

## "Hello World" in embedded JavaScript

Remember that we said that your browser already knows how to interpret both HTML and JavaScript. In fact, it also understands when you mix the two, as long as you tell it what you are doing. We are going to make extensive use of this capability later on, so let's see how it works.

If you want to include JavaScript within HTML, you use the *script* tag to tell the browser that you are doing so. You can then embed the script right in the body of your HTML file like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
<head>
  <title>Hello World! Script</title>
</head>
<body>
  <script type="text/javascript">
    document.write("Hello World!");
  </script>
</body>
</html>
```

Create a new empty HTML file in Komodo and modify the title and body to match the example above. Save it as *hello-world-js.html*. When you open it with Firefox, your message should appear as before.

We've now gotten the same result using HTML in two very different ways, so we should be clear about the difference. In the first case we created a very basic static web page using pure HTML. The body of the page says "Hello World!" and nothing else. In the second case, we created a blank HTML page and then ran a short JavaScript program to print "Hello World!" onto that blank page. From the point of view of the person reading the page, they look the same and it may not matter to them how the page was created. From our perspective, however, the difference is crucial, because the second method allows us to embed our JavaScript programs in HTML files which can be viewed in the browser. Anything that can be viewed in the browser can be indexed and annotated with Zotero. This means that you can keep track of the programs that you write and their output using the same system that you use to keep track of the rest of your research.

## Back up your work

Once you begin to program, it is crucial that you make backups of your work regularly. Each day before you do any programming, make sure to back up your Zotero database. At the end of a day's work, make another

backup of the Zotero database and of any programs that you've written that day. You should back up your whole computer at least weekly, and preferably more frequently.

## Keep in touch with us

As you work through the examples in this book you will, no doubt, want to apply similar techniques to your own sources. If you come up with a variation or generalization, e-mail us to let us know about it. Likewise, if you run into trouble or can't figure out how to modify one of our programs so it applies to your situation, we'd like to hear from you. We can try to help you get something running, or try to add some new material to *The Programming Historian* to cover situations like yours.

## Other resources

As you're working through the tutorials here, you will want to have a few key resources open in your browser. Until you become familiar with the programming languages that we're using, it is nice to have a few different introductory treatments to look at. There are many good online resources like

- \* [Python for Non-programmers](#)
- \* [W3 Schools HTML Tutorial](#)

As you proceed (or if you already have some programming experience) you'll probably prefer more general references like:

- \* [Python for Programmers](#)
- \* [Python documentation page](#)
- \* [Python tutorial](#)
- \* [Python library reference](#)
- \* Pilgrim, [Dive into Python](#)

We also like to have a few printed books ready-to-hand, especially

- \* Lutz, [Learning Python](#)
- \* Lutz, [Programming Python](#)
- \* Martelli, Ravenscroft and Ascher, [Python Cookbook](#)

Other references will be cited as we make use of them.

## Suggested readings

Some of our readers have expressed an interest in using *The Programming Historian* for formal or informal coursework. To get a solid foundation in Python programming, it is probably best to pair these exercises with some additional readings. We like Mark Lutz's *Learning Python*, 3rd ed. Sebastopol, CA: O'Reilly, 2008.

- Lutz, [Learning Python](#)  
(optional) Ch. 1: A Python Q&A Session  
Ch. 2: How Python Runs Programs  
Ch. 3: How You Run Programs

### 3. Working with files and web pages

#### Making use of your ability to do close reading

From now on, you will be seeing more and more samples of code. Try to get into the habit of reading each one closely, the way that you would read a particularly important primary source. If there is something in the code that you haven't seen before or don't understand, try to make an explicit hypothesis about how it must work. Sometimes your hypothesis will be correct, and sometimes it won't, but it is much easier to make progress if you are mindful about your own assumptions. This is also the stance that you will need to take when you begin to debug code that doesn't work. One of the advantages that historians have when they turn to programming is that they are already in the habit of interrogating sources rather than taking them at face value.

#### Sending information to text files

In a previous section, you saw how to send information to the "Command Output" pane of Komodo Edit by using Python's print command.

```
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 processing that data. In the example above, we see one kind of object, the string hello world. A string object is a sequence of characters; we'll learn more about string methods soon. Print is a command that prints objects in textual form.

You will use print like this in cases where you want to create information that you are going to act on 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 Komodo Edit and save it as *file-output.py*.

```
# file-output.py
f = open('helloworld.txt', 'w')
f.write('hello world')
f.close()
```

In this program *f* is a file object, and *open*, *write* and *close* are file methods. In the open method, 'helloworld.txt' is the name of the file that you are going to create, and the 'w' parameter says that you are opening the file to write to it. Note that both the file name and the parameter are strings in this case. Your program writes the message (another string) to the file and then closes the file. (For more information about these statements, see the section on File Objects in the *Python Library Reference*.)

Double-click on your "Run Python" button to execute the program. Although nothing will be printed to the "Command Output" pane, you will see a status message that says

```
`/usr/bin/python file-output.py` returned 0.
```

on the Mac, or

```
'C:\Python25\Python.exe file-output.py' returned 0.
```

on Windows. This means that your program executed successfully. If you use File->Open->File in 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 with a text editor like Komodo Edit.

## Getting information from text files

Python also has statements which allow you to get information from files. Type the following program into Komodo Edit and save it as `file-input.py`. When you double-click "Run Python" to execute it, it will open the text file, 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. *Read* is another file method. The contents of the file (the one-line message) are copied into *message*, which is a string, and then the *print* command is used to send the contents of *message* to the "Command Output" pane.

## Splitting code into modules and functions

You 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. Suppose, for example, that you keep all of your bibliographic references in Zotero and you have a tag to indicate which ones you need to get on your next trip to the library. It would be useful to have a program that selected only those tagged items and sorted them by call number (so you don't have to waste time wandering from one part of the library to the next when you're retrieving them). Since this is part of your research practice, you'll want to be able to re-run this program before each trip to the library. A program, in other words, is a mechanism for bundling a collection of statements together to facilitate re-use. Zotero itself is a bundle of useful statements, as is Firefox.

When programs are small, 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*. In essence, this modularization allows programmers to re-use code for tasks that they have to do over and over. Below, for example, you'll see that commands for working with web pages have been put into a separate Python module. Python has a special *import* statement that allows one program to gain access to the contents of another program file. (As you work through the examples below, make sure that you understand the difference between loading a data file and importing a program file.)

At a finer level of detail, 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 and a module. 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`. This file is your module.

```
# greet.py
```

```
def greetEntity (x):  
    print "hello " + x
```

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 being defined. You will learn more about this as we go along; for now, make sure to keep indentation the way we show it.

Now you can create another program that imports code from your module and makes use of it. Copy this code to Komodo Edit and save it as *using-greet.py*. This file is your program.

```
# using-greet.py  
  
import greet  
greet.greetEntity("everybody")  
greet.greetEntity("programming historian")
```

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. (Note that from this example and the previous ones, you might infer that strings in Python can be delimited with single or double quotes. That is true.) If all went well, you should see

```
hello everybody  
hello programming historian
```

in the command output pane of Komodo Edit.

You can think of the granularity of code in two ways:

**Top-down.** If you think of all the things that you want to use a computer for, you can decompose the problem into recurring sub-problems. You need to work with files (operating system), documents (word processor), numbers (spreadsheet), data (database), pictures (image processing program), web pages (browser) and so on. A particular program will need to be able to open, manipulate, and store files. You may want the ability to check your spelling in documents, e-mail or presentations. In order to check spelling, you need some kind of dictionary and the ability to look up each word in it. Looking up words involves being able to compare them character-by-character, and so on. Each task can be partitioned into smaller ones.

**Bottom-up.** Suppose you start with a simple task, like adding two numbers together ( $a+b$ ). Once you know how to do that, it is possible to generalize your ability to add any number of numbers together  $(a+b)+c = (a+b+c)$ . From adding you can get multiplication  $(a*3) = (a+a+a)$ . Being able to add numbers is such a useful function, that it recurs constantly. Your operating system will need addition to determine how much file space is left on your hard drive. Your word processor will need it to keep track of word counts and page numbers. Your spreadsheet will need to do a lot of addition. Useful building blocks can be combined and recombined at every level of complexity.

## 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 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) into a web page. 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://niche-canada.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 NiCHE website. The default assumption is that the main page in a given directory will be named index, usually index.html. The NiCHE website is written in a different language than HTML, however, so the name of the main page is index.php. ([PHP](#) is another web programming language. If you'd like to learn more about it, there is a [W3 Schools tutorial](#).)

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://niche-canada.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 table of contents for this book has the following URL. Note that we don't need to specify the filename.

<http://niche-canada.org/programming-historian/>

Finally, some web pages allow you to enter queries. The NiCHE 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 the main page for the NiCHE Digital Infrastructure.

<http://niche-canada.org/?q=node/12>

## Opening URLs with Python

In order to be able to automatically harvest and process web pages, 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 Adam Dollard Des Ormeaux (1635-60), a controversial figure in Canadian historiography. With Google, it's easy to locate his biographical entry in the online *Dictionary of Canadian Biography*.

**IMPORTANT NOTE:** The DCB website was updated at the end of June 2008, and can no longer be used for the example code we have here. As a temporary solution, we've changed our code to link to a few files on the NiCHE server that have the same formatting as the DCB site used to have. When we get a chance, we'll rewrite the sections so they are compatible with the new online DCB. In the meantime, please e-mail us if you find something that doesn't work!

Français	Contact Us	Help	Search	Canada Site
Library and Archives Canada	Université Laval	University of Toronto	Links	

You are here: [Home](#) | [Search Results](#) | [Biography Display](#)

## DICTIONARY OF CANADIAN BIOGRAPHY ONLINE

[Show printable page](#)

**DOLLARD DES ORMEAUX** (called **Daulat** in his death certificate and **Daulac** by some historians), **ADAM**, soldier, "garrison commander of the fort of Ville-Marie (Montreal)", b. 1635, killed by the Iroquois at the Long Sault in May 1660

Nothing is known of Dollard's activities prior to his arrival in Canada except that he had held some command in the armies of France. "Having come to Montreal as a volunteer very probably in 1658, he continued his military career there. In 1659 and 1660 he was described as an "officer" or "garrison commander of the fort of Ville-Marie," a title that he shared with Pierre PICOTÉ de Belestre. We do not however know what his particular responsibility was. Dollard was perhaps contemplating becoming a settler. At the end of 1659 Chouérou de Maisonneuve gave him a piece of land comprising 30 arpents. In 1661 the sum that Dollard had devoted "to having work done on the aforementioned grant" was calculated at 79 livres, 10 sols, "for 53 days' labour."

Dollard had an excellent reputation at Montreal. First-hand evidence, it is true, is rare: the *Relation* calls him "a man of accomplishments and generalship," and Dollard de Casson calls him "a youth of courage and of good family." But Dollard had earned the governor's confidence and the esteem of his fellow-townsmen. For anyone who is acquainted with the social and religious climate of Ville-Marie in 1660, is any better recommendation needed? It would have been unthinkable for example for Maisonneuve to promote to garrison commander an officer whose conduct had not been impeccable. Would Lambert Closse have chosen him to be godfather to his daughter Elisabeth (3 Oct. 1658)? Would his presence have been sought, a score of times, to witness the signature before BÉNIGNE BASSER of contracts of all sorts, if Dollard had not been a thoroughly honourable man? Finally, would Maisonneuve have let him leave for the Long Sault in April 1660 if he had not had complete confidence in him?

To be sure, much ill has been spoken of Dollard, accused of stealing furs and of being headstrong. These accusations, however, are not based upon any documentary proof and in addition are contradicted by the facts. But the temptation to criticize was great. Dollard de Casson states that Dollard "may have been very glad of an opportunity to distinguish himself, to be of use to him on account of something which was said to have happened to him in France." What was this "something," and how serious had it been? We know nothing of it. It would be unreasonable to construct hypotheses upon a piece of information so fragile and which seems to be pure hearsay. Let it suffice to record that Dollard led an orderly life at Montreal and that he was well thought of by his superiors and his fellow-townsmen.

The URL for the main entry is (i.e., used to be... just play along)

<http://www.biographi.ca/EN/ShowBio.asp?BioId=34298>

By studying the URL we can learn a few things. First, the DCB website uses Microsoft Active Server Pages (ASP) and it's possible to retrieve individual biographies by making use of the query string. Each is apparently given a unique 5-digit ID number. From the presence of EN in the path (presumably standing for the English language entries), we can infer there might be a corresponding French-language entry at

<http://www.biographi.ca/FR/ShowBio.asp?BioId=34298>

In fact, this inference is correct. Looking at the webpage, we also notice that there is a printable version of the page.

**DOLLARD DES ORMEAUX** (called **Daulat** in his death certificate and **Daulac** by some historians), **ADAM**, soldat, "garrison commander of the fort of Ville-Marie (Montreal)", b. 1635. Killed by the Iroquois at the Long Sault in May 1660.

Nothing is known of Dollard's activities prior to his arrival in Canada except that he had held some command in the armies of France. "Having come to Montreal as a volunteer, very probably in 1658, he continued his military career there. In 1659 and 1660 he was described as an "officer" or "garrison commander of the fort of Ville-Marie," a title that he shared with Pierre PICOTÉ de Belestre. We do not however know what his particular responsibility was. Dollard was perhaps contemplating becoming a settler. At the end of 1659 Chouérou de Maisonneuve gave him a piece of land comprising 30 arpents. In 1661 the sum that Dollard had devoted "to having work done on the aforementioned grant" was calculated at 79 livres, 10 sols, "for 53 days' labour."

Dollard had an excellent reputation at Montreal. First-hand evidence, it is true, is rare: the *Relation* calls him "a man of accomplishments and generalship," and Dollard de Casson calls him "a youth of courage and of good family." But Dollard had earned the governor's confidence and the esteem of his fellow-townsmen. For anyone who is acquainted with the social and religious climate of Ville-Marie in 1660, is any better recommendation needed? It would have been unthinkable for example for Maisonneuve to promote to garrison commander an officer whose conduct had not been impeccable. Would Lambert Closse have chosen him to be godfather to his daughter Elisabeth (3 Oct. 1658)? Would his presence have been sought, a score of times, to witness the signature before BÉNIGNE BASSER of contracts of all sorts, if Dollard had not been a thoroughly honourable man? Finally, would Maisonneuve have let him leave for the Long Sault in April 1660 if he had not had complete confidence in him?

To be sure, much ill has been spoken of Dollard, accused of stealing furs and of being headstrong. These accusations, however, are not based upon any documentary proof and in addition are contradicted by the facts. But the temptation to criticize was great. Dollard de Casson states that Dollard "may have been very glad of an opportunity to distinguish himself, to be of use to him on account of something which was said to have happened to him in France." What was this "something," and how serious had it been? We know nothing of it. It would be unreasonable to construct hypotheses upon a piece of information so fragile and which seems to be pure hearsay. Let it suffice to record that Dollard led an orderly life at Montreal and that he was well thought of by his superiors and his fellow-townsmen.

This then was the man who, in the spring of 1660, assumed the leadership of an expedition to the Ottawa. Like him, his 16 companions all came from Montreal and all were unmarried. Eight of them had landed at Ville-Marie in 1653: Jacques Brassier, aged 25, François Chouérou de Maisonneuve, 24; René Duressin, 30, a miller and soldier; Nicolas Jassein, 25, originally from Solesmes in Normandy; Jean Lecompte, 26, a cigger and woodcutter from the parish of Chambré-en-Charnie in Le Maine; Etienne Robin of Des Forges; 27, Sieur Jean Tavemier de La Forest, of La Lochetière, 26, an armoué, originally from Rozé in Le Maine; and Jean Valets, 27, a ploughman from the parish of Thoiré (or Teil) in

Its URL is

<http://www.biographi.ca/EN/ShowBioPrintable.asp?BioId=34298>

When you are processing web resources automatically, it is often a good idea to work with printable versions, as they tend to have less formatting.

Now let's try opening the printable version of the page. Copy the following program into Komodo Edit and save it as *open-html.py*. When you execute it, it will open the biography file, read its contents into a Python string called `html` and then print the first three hundred characters of the string to the "Command Output" pane. Use the 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. (See the Python library reference to learn more about `urllib2`.)

```
# open-html.py

import urllib2

# note that we have to grab a copy of the old page since the new DCB website doesn't
work the way it used to
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'

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

print html[0:300]
```

## 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 `html` string to a local file rather than the "Command Output" pane. Copy the following program into Komodo Edit, save it as *save-html.py* and execute it. Using the File->Open File command in Firefox, open the local file that it creates (`dcb-34298.html`) to confirm that your saved copy is the same as the online copy.

```
# save-html.py

import urllib2

# note that we have to grab a copy of the old page since the new DCB website doesn't
work the way it used to
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'

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

f = open('dcb-34298.html', 'w')
f.write(html)
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 biography ID numbers, for example, and make your own copies of a whole bunch of them? Yep. We'll get there soon.

## Suggested Readings

Lutz, *Learning Python*

Ch. 4: Introducing Python Object Types

### 4. From HTML to a list of words

#### Getting rid of HTML formatting

Often we're interested in keeping the textual content of an online source for processing, but we'd like to get rid of the HTML tags and metadata. We're going to start by doing this the quick and dirty way. In the HTML that you've seen so far, there have been a few basic kinds of tags. In each case, it looks as if we will be safe ignoring everything between a matching pair of angle brackets.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- This is a comment -->
<title>Title of page</title>
```

Our *algorithm* is going to be as follows

1. Start with an empty string to store our text in
2. Look at every character in the html string, one at a time
3. If the character is a left angle bracket (<) we are now inside a tag so ignore the character
4. If the character is a right angle bracket (>) we are now leaving the tag
5. If we're inside a tag ignore the character, otherwise append it to the text string

An algorithm is a procedure that has been specified in enough detail that it can be implemented on a computer. We turn to the implementation now.

#### More about Python strings

So far you've seen two ways that strings can be delimited, using either a matching pair of single or double quotes:

```
message1 = 'hello world'
message2 = "hello world"
```

Python has a third kind of string that can span multiple lines. This will be useful later.

```
message3 = """hello
hello
hello world"""
```

Python includes a number of statements for manipulating strings. If you'd like to experiment with these statements, you can write and execute short programs as we've mostly been doing, or you can open up a Python shell.

You can concatenate strings (i.e., join them together) using the plus operator. Note that you have to be explicit about where you want blank spaces to occur. You can also create multiple copies of strings by using the multiplication operator.

```
message4 = 'hello' + ' ' + 'world'  
print message4
```

-> hello world

```
message5a = 'hello ' * 3  
message5b = 'world'  
print message5a + message5b
```

-> hello hello hello world

What if you want to successively add material to the end of a string? There is a special operator for that.

```
message6 = 'howdy'  
message6 += ' '  
message6 += 'world'  
print message6
```

-> howdy world

You can determine the number of characters in a string using *len*. Note that the blank space counts as a separate character.

```
message7 = 'hello' + ' ' + 'world'  
print len(message7)
```

-> 11

Finally, you are occasionally in a situation where you need to include quotation marks of various kinds within a string, and 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
```

To return to our algorithm, we first have the problem of creating an empty string to store text in.

```
text = ''
```

OK, that was easy. We already know how we're going to append characters to this string when we need to:

```
text += char
```

## Looping

Now we need a way to look at every character in the html string, one at a time. Like many programming languages, Python includes a number of looping mechanisms. The one that we want is called a *for* loop. The version below tells the interpreter to do something for each character in a string named html. In effect, it creates a one-character-long string named char, which will contain each character from html in succession.

```
for char in html:
    # do something with char
```

## Branching

Next we 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 we want is called an *if* statement. The version below tests to see whether the string char contains a left angle bracket.

```
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 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
```

Just to avoid confusion, 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.

How will we keep track of whether or not we're inside a tag? We can use a number variable called inside which will be 1 (true) if we're inside a tag and 0 (false) if we're not.

## The stripTags routine

Putting it all together, the final version of our routine is shown below. Copy this code and paste it into Komodo edit. Save it in a file called *dh.py*. This file is going to contain all of the code that we will wish to re-use. In other words, *dh.py* is a module. (More in the discussion page).

```
# Given a string containing HTML, remove all characters
# between matching pairs of angled brackets, inclusive.
```

```

def stripTags(html):
    inside = 0
    text = ''
    for char in html:
        if char == '<':
            inside = 1
            continue
        elif (inside == 1 and char == '>'):
            inside = 0
            continue
        elif inside == 1:
            continue
        else:
            text += char
    return text

```

As you look over this code, you will notice that we needed one final command to make it work. The Python *continue* statement tells the interpreter to jump back to the top of the enclosing loop. So if the character is a left angle bracket, once you've made a note that you're inside a tag, you're finished processing that character. You want to go get the next character in the html string, rather than continuing to process the one you've already dealt with.

## Python lists

Now that we have the ability to extract raw text from web pages, we're going to want to get the text in a form that is easy to process. So far, when we've needed to store information in our Python programs, we've usually used strings. There were a couple of exceptions, however. In the *striptags* routine, we also made use of an integer named "inside" to store a 1 when we were processing a tag and a 0 when we weren't.

```
inside = 1
```

And whenever we've needed to read from or write to a file, we'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 of object that Python provides, however, is the list, an ordered collection of other objects (including, potentially, other lists). The fact that lists can contain lists makes them ideal for storing tree-like structures, something that we will explain soon and come back to repeatedly. It is also straightforward to turn a string into a list of characters or a list of words, as shown in the following program. Copy it into Komodo Edit, save it as *string-to-list.py* and execute it. Compare the two lists that are printed to the "Command Output" pane.

```

# 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, returns and similar characters). Actually, it is a bit of a simplification to refer to the objects in the second list as 'words'. Try changing *s2* in the above program to 'howdy world!' and running it again. What happened to the exclamation mark?

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-list-1.py

import urllib2
import dh

# note that we are using a copy of the old web page because the DCB site doesn't work
the way it used to
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'

response = urllib2.urlopen(url)
html = response.read()
text = dh.stripTags(html)
wordlist = text.split()
print wordlist[0:120]
```

You should get something like the following.

```
['Dictionary', 'of', 'Canadian', 'Biography', 'DOLLARD', 'DES',
'ORMEAUX', '(called', 'Daulat', 'in', 'his', 'death', 'certificate',
'and', 'Daulac', 'by', 'some', 'historians)', 'ADAM', 'soldier',
'\x93garrison', 'commander', 'of', 'the', 'fort', 'of',
'Ville-Marie', '[Montreal]\x94;', 'b.', '1635,', 'killed', 'by',
'the', 'Iroquois', 'at', 'the', 'Long', 'Sault', 'in',
'May1660.', '\xa0\xa0\xa0\xa0\xa0', 'Nothing', 'is', 'known',
'of', 'Dollard\x92s', 'activities', 'prior', 'to', 'his', 'arrival',
'in', 'Canada', 'except', 'that', '\x93he', 'had', 'held', 'some',
'commands', 'in', 'the', 'armies', 'of', 'France.\x94', 'Having',
'come', 'to', 'Montreal', 'as', 'a', 'volunteer,', 'very',
'probably', 'in', '1658,', 'he', 'continued', 'his', 'military',
'career', 'there.', 'In', '1659', 'and', '1660', 'he', 'was',
'described', 'as', 'an', '\x93officer\x94', 'or', '\x93garrison',
'commander', 'of', 'the', 'fort', 'of', 'Ville-Marie,\x94', 'a',
'title', 'that', 'he', 'shared', 'with', 'Pierre', 'Picot\xe9',
'de', 'Belestre.', 'We', 'do', 'not', 'however', 'know', 'what',
'his', 'particular', 'responsibility', 'was.']
```

Simply having a list of words doesn't buy us much yet. As human beings, we already have the ability to read. We're getting much closer to a representation that our programs can process, however.

## Suggested Readings

Lutz, *Learning Python*  
Ch. 7: Strings

Ch. 8: Lists and Dictionaries

Ch. 10: Introducing Python Statements

Ch. 15: Function Basics

## 5. Computing frequencies

### Useful measures of a text

In a previous section, you wrote a Python program called *html-to-list-1.py* which downloaded a web page, stripped out the HTML formatting and metadata and returned a list of 'words', like the one shown below.

```
['Dictionary', 'of', 'Canadian', 'Biography', 'DOLLARD', 'DES',
'ORMEAUX', '(called', 'Daulat', 'in', 'his', 'death', 'certificate',
'and', 'Daulac', 'by', 'some', 'historians)', 'ADAM', 'soldier',
'\x93garrison', 'commander', 'of', 'the', 'fort', 'of',
'Ville-Marie', '[Montreal]\x94;', 'b.', '1635', 'killed', 'by',
'the', 'Iroquois', 'at', 'the', 'Long', 'Sault', 'in',
'May1660.', '\xa0\xa0\xa0\xa0\xa0', 'Nothing', 'is', 'known',
'of', 'Dollard\x92s', 'activities', 'prior', 'to', 'his', 'arrival',
'in', 'Canada', 'except', 'that', '\x93he', 'had', 'held', 'some',
'commands', 'in', 'the', 'armies', 'of', 'France.\x94', 'Having',
'come', 'to', 'Montreal', 'as', 'a', 'volunteer', 'very',
'probably', 'in', '1658', 'he', 'continued', 'his', 'military',
'career', 'there.', 'In', '1659', 'and', '1660', 'he', 'was',
'described', 'as', 'an', '\x93officer\x94', 'or', '\x93garrison',
'commander', 'of', 'the', 'fort', 'of', 'Ville-Marie,\x94', 'a',
'title', 'that', 'he', 'shared', 'with', 'Pierre', 'Picot\xe9',
'de', 'Belestre.', 'We', 'do', 'not', 'however', 'know', 'what',
'his', 'particular', 'responsibility', 'was.']
```

By itself, this ability 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 words and other linguistic units, a classic measure of a text.

### Cleaning up the list

It is clear that our list is going to need some cleaning up before we can use it to count frequencies. For one thing, we won't want the frequencies of words to depend on capitalization: "Dollard" and "DOLLARD" should count as the same word. Typically words are folded to lowercase when counting frequencies, so we'll do that using the string method `lower`.

```
print('Hello WORLD'.lower())
```

```
-> hello world
```

There are assorted punctuation marks that will throw off the frequency counts if they are left in. We want "soldier," to be counted as "soldier" and "[Montreal]" as "Montreal", of course. Looking through the output we also find " " which is an HTML ampersand character code for a non-breaking space. Using another string method, we can replace that code with a blank space, as in the following.

```
print('hello&nbsp;world')
```

```
-> hello&nbsp;world
```

```
print('hello&nbsp;world'.replace('&nbsp;',' '))
```

```
-> hello world
```

There are also a number of accented French characters which are represented with Unicode strings like "\xe9" (which stands for "é"). We'll learn more about working with Unicode characters later; for now we'll leave them as they are.

At this point, we might look through a number of other DCB 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 dollar amounts like "\$1629" 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.

## Our first use of regular expressions

In order to eliminate special characters, we're going to make use of a very powerful mechanism called *regular expressions*. Regular expressions are provided by many programming languages in a range of different forms. To do what we want to do right now, we have to import the Python regular expression library and compile a pattern that matches anything that isn't an alphanumeric character. Copy the following function and paste it into the *dh.py* module.

```
# 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 to 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-list program, it now looks like this:

```
# html-to-list-2.py

import urllib2
import dh

url = 'http://niche.uwo.ca/programming-historian/dcb/dcb-34298.html'
```

```
response = urllib2.urlopen(url)
html = response.read()
text = dh.stripTags(html).replace('&nbsp;', ' ')
wordlist = dh.stripNonAlphaNum(text.lower())
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. As expected, it has left accented characters as codes, so words like "Picoté" appear as "picot\xe9". It has split hyphenated forms like "Ville-Marie" into two words and turned the possessive "s" into a separate word 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](#) and about [Python support for Unicode](#).)

## 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 first access a particular string, 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

print m[0][1]

-> e

print m[1][0]

-> w
```

To keep track of frequencies, we're going to need 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']
```

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.

## Counting word 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.

```
# 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 word in wordlist:  
    wordfreq.append(wordlist.count(word))
```

```
print "String\n" + wordstring + "\n"  
print "List\n" + str(wordlist) + "\n"  
print "Frequencies\n" + str(wordfreq) + "\n"  
print "Pairs\n" + str(zip(wordlist, wordfreq))
```

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)]
```

In the preceding program, we start with a string and split it into a list, as we've done before. We then go through each word in the list, and count the number of times that word appears in the whole list, and add the

count to another list of word frequencies. 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* statement converts any object to a string so that it can be printed.

Python also includes a very convenient tool called a *list comprehension*, which can be used to do the same thing as the for loop more economically.

```
# count-list-items-2.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]

print "String\n" + wordstring + "\n"
print "List\n" + str(wordlist) + "\n"
print "Frequencies\n" + str(wordfreq) + "\n"
print "Pairs\n" + str(zip(wordlist, wordfreq))
```

At this point we have a list of pairs, where each pair contains a word and its frequency. Note that this list is redundant. If "the" occurs 500 times, then this list contains five hundred copies of the pair ('the', 500). This list is also ordered by the words in the original text. We can solve both problems by converting it into a dictionary. Then all we have to do is print out the dictionary in order from the most to the least commonly occurring item.

## From HTML to a dictionary of word-frequency pairs

6 Apr 2008. [Mac tested to here.](#)

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 *dh.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 *dh.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 the output carefully before continuing.

```
# html-to-freq.py

import urllib2
import dh

# note that we are using a copy of the old DCB page again
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'

response = urllib2.urlopen(url)
html = response.read()
text = dh.stripTags(html).replace('&nbsp;', ' ')
wordlist = dh.stripNonAlphaNum(text.lower())
dictionary = dh.wordListToFreqDict(wordlist)
sorteddict = dh.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".

```
(647, 'the')
(310, 'of')
(273, 'to')
(202, 'and')
(171, 'in')
(134, 'a')
(118, 'that')
(91, 'dollard')
(78, 'was')
(78, 'their')
(75, 'were')
(72, 'they')
(71, 'his')
```

These words are usually the most common in any English language text, so they don't tell us much that is distinctive about Dollard's biography. 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. Copy it and put it at the beginning of the *dh.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 *dh.py* module, too.

```

# 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 dh

# old copy of the page again
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'

```

```
response = urllib2.urlopen(url)
html = response.read()
text = dh.stripTags(html).replace('&nbsp;', ' ')
fullwordlist = dh.stripNonAlphaNum(text.lower())
wordlist = dh.removeStopwords(fullwordlist, dh.stopwords)
dictionary = dh.wordListToFreqDict(wordlist)
sorteddict = dh.sortFreqDict(dictionary)
for s in sorteddict: print str(s)
```

If all went well, your output should look like this:

```
(91, 'dollard')
(64, 'iroquois')
(33, 'long')
(27, 'sault')
(24, 'enemy')
(24, '1660')
(20, 'time')
(20, 'seventeen')
(20, 'french')
(19, 'new')
(19, 'montreal')
(19, 'army')
(18, 'hurons')
(18, 'fort')
(17, 'france')
(15, 'men')
(14, 'marie')
(14, 'companions')
...
```

## Suggested Readings

Lutz, *Learning Python*

Ch. 9: Tuples, Files, and Everything Else

Ch. 11: Assignment, Expressions, and print

Ch. 12: if Tests

Ch. 13: while and for Loops

## 6. Wrapping output in HTML

### Putting new information where you can use it

At this point, you've started to learn how to use Python to download online sources and extract information from them automatically. Remember that your ultimate goal is to incorporate programming seamlessly into your historical practice. Since you are already using Firefox and Zotero to find and keep track of your sources, it also makes sense to use these programs to keep track of any new information that you create. The easiest way to do this is to have your Python programs output local web pages that you can read in Firefox and index and annotate with Zotero. We turn to that now, starting with a discussion of some more of the things that you can do with Python strings.

## Python string formatting

Python includes a special formatting operator that allows you to interpolate one string in another one. It is represented by a percent sign. Open a Python shell and try the following examples.

```
frame = 'This is a %s'  
print frame
```

```
-> This is a %s
```

```
print frame % 'cat'
```

```
-> This is a cat
```

```
print frame % 'dog'
```

```
-> This is a dog
```

There is also a form which allows you to interpolate a list of strings into another one.

```
frame2 = 'These are %s and %s'  
print frame2
```

```
-> These are %s and %s
```

```
print frame2 % ('cats', 'dogs')
```

```
-> These are cats and dogs
```

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, `%f` for floating-point decimal, and so on. We will introduce these later as necessary.

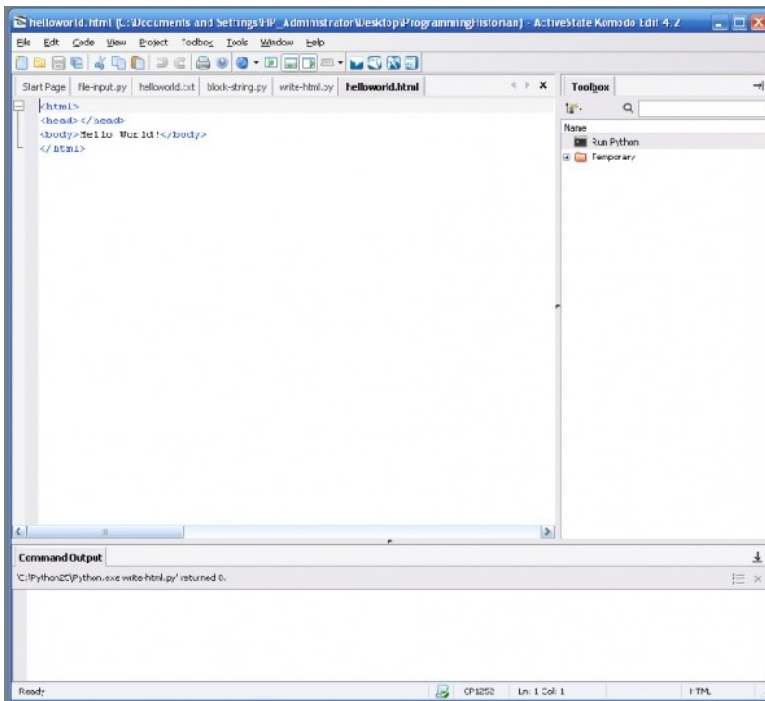
## Creating HTML output

One of the more powerful ideas in computer science is that something that is code from one perspective can be seen as data from another. It's possible, in other words, to write programs that manipulate other programs. The Python interpreter is one example. What we're going to do next is combine Python files, multiline block strings and simple HTML tags to create a Python program which outputs an HTML file. Note that we are writing to a file with an `.html` extension rather than a `.txt` extension.

```
# write-html.py  
  
f = open('helloworld.html', 'w')  
  
message = """<html>  
<head></head>  
<body>Hello World!</body>  
</html>"""  
  
f.write(message)  
f.close()
```

Save this program as `write-html.py` and execute it. Use File->Open->File in Komodo Edit to open

*helloworld.html* to verify that your program actually created the file. It should look something like this:



### "Hello World" 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.

### Sending HTML output to Firefox

We automatically created an HTML file, but then we had to leave Komodo Edit 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? Enter the code below into Komodo Edit 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!

```
# write-html-2.py
import webbrowser

f = open('helloworld.html', 'w')

message = """<html>
<head></head>
<body>Hello World!</body>
</html>"""

f.write(message)
f.close()

webbrowser.open_new_tab('helloworld.html')
```

N.B. Some people couldn't get this to work on Mac OS X (because of the different ways that the various operating systems handle web browsers). If you are getting a bunch of "MacOS.Error -673" messages, you have to comment out two lines in the function *wrapStringInHTML* defined below.

## Self-documenting data files

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 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 in-place to generate and keep track of metadata.

When you're working with digital data, it is a good idea to incorporate metadata into your files whenever possible. In later sections, we will work with the [Extensible Markup Language \(XML\)](#), which is ideal for this purpose. For now, however, we need to develop a few basic strategies for making our data files self-documenting.

## Python comments

You've already seen one example of this. 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. 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*; 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).

## Building an HTML wrapper

You've just learned how to embed a message like "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 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.

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 into the *dh.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).replace('&nbsp;', ' ')
    return text.lower()
```

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 *dh.py* module.

N.B. If you are using Mac OS X and you were unable to run the program *write-html-2.py* above, then you have to comment out two lines in the following program by putting a hash mark in front of each one:

```
...  
# from webbrowser import open_new_tab  
...  
# open_new_tab(filename)  
...
```

Once you've made the changes, you can copy the code into the *dh.py* module. Please e-mail us if this fix doesn't work for you.

```
# 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 that you learned about. It also calls the Python *datetime* library to determine the current time and date. This 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 an *.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 the "Command Output" pane in Komodo, it sends it to an HTML file which is opened in a new Firefox tab. From there, the program's output can be added easily to Zotero. Copy the following code to Komodo Edit, save it as *html-to-freq-3.py* and execute it, to confirm that it works as expected.

```
# html-to-freq-3.py  
  
import dh  
  
# create sorted dictionary of word-frequency pairs  
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'  
text = dh.webPageToText(url)  
fullwordlist = dh.stripNonAlphaNum(text)  
wordlist = dh.removeStopwords(fullwordlist, dh.stopwords)  
dictionary = dh.wordListToFreqDict(wordlist)  
sorteddict = dh.sortFreqDict(dictionary)
```

```
# compile dictionary into string and wrap with HTML
outstring = ""
for s in sorteddict:
    outstring += str(s)
    outstring += "<br />"
dh.wrapStringInHTML("html-to-freq-3", url, outstring)
```

Note that we interspersed our word-frequency pairs with the HTML break tag, which acts as a newline. If all went well, you should see the same word frequencies that you computed in the last section.

## Using word frequencies to refine a Google search

Let's go through one more cycle of refinement. Start by doing a [Google](#) search for "dollar" and counting the number of hits on the first five pages that actually refer to Adam Dollard Des Ormeaux. When we tried this on 7 Jan 2008, we ended up with three out of fifty, or six percent.

Now try doing a Google search for "dollar iroquois long sault enemy" and counting the number of hits that refer to Adam Dollard Des Ormeaux. When we tried this on 7 Jan 2008, we ended up with fifty out of fifty, or one hundred percent. So by using the words that are most characteristic of this text, we can easily find others like it. Wouldn't it be great to do this automatically?

Look at the URL of the Google search that you just did. It should begin with something like

<http://www.google.com/search?q=dollar+iroquois+long+sault+enemy>

Suppose we choose some small number  $n$ . If we take the top  $n$  keywords from our word frequency list, we can construct a query like this automatically and build it into a link that we display on our wrapped results page.

The basic form of a hyperlink in HTML is

```
<a href="URL">LINKNAME</a>
```

We want to build up the URL for a Google search automatically, then embed it in an HTML a tag like the one above. Study the following function then add it to the *dh.py* module.

```
# Given a list of keywords and a link name, return an
# HTML link to a Google search for those terms.

def keywordListToGoogleSearchLink(keywords, linkname):
    gsearch = '<a style="text-decoration:none" '
    gsearch += 'href="http://www.google.com/search?q='
    gsearch += '+'.join(keywords)
    gsearch += '\>'
    gsearch += linkname
    gsearch += '</a>'
    return gsearch
```

Note that we've added a bit of inline [CSS](#) to the HTML a tag to prevent the browser from underlining hyperlinks automatically. We'll learn more about CSS (Cascading Style Sheets) later; for now this will make the output of the next couple of programs that we write more legible.

(There is one thing about this code that is somewhat counterintuitive. In order to create a string from a list, you call a string method `join` on a string consisting of the delimiter that you want to use between list

elements. The delimiter is a plus sign in our case, since we're building the query string of a URL. [Many people expect join to be a list method](#), but it isn't.)

You can test the `keywordListToGoogleSearchLink` function in a Python shell if you'd like. Copy the function definition, paste it into the shell and press Enter. Then you can do something like the following:

```
testwords = ('this', 'is', 'a', 'test')
print keywordListToGoogleSearchLink(testwords, "Do Google Search")
```

```
-> Do Google Search
```

Now we can revise our code to include this automatically-constructed Google search link. Copy the following to Komodo Edit, save it as `html-to-freq-4.py` and execute it.

```
# html-to-freq-4.py

import dh

# create sorted dictionary of word-frequency pairs
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'
text = dh.webPageToText(url)
fullwordlist = dh.stripNonAlphaNum(text)
wordlist = dh.removeStopwords(fullwordlist, dh.stopwords)
dictionary = dh.wordListToFreqDict(wordlist)
sorteddict = dh.sortFreqDict(dictionary)

# create Google search link
keywords = []
for k in sorteddict[0:5]:
    keywords.append(str(k[1]))
gsearch = dh.keywordListToGoogleSearchLink(keywords, 'Google Search n=5')

# compile dictionary into string and wrap with HTML
outstring = gsearch + "<br /><br />"
for s in sorteddict:
    outstring += str(s)
    outstring += "<br />"
dh.wrapStringInHTML("html-to-freq-4", url, outstring)
```

When you try this program, you will see that there is now a hyperlink in the output that you can follow to submit the refined search to Google automatically. As an exercise, try modifying your script to process the biography of the explorer [Pierre-Esprit Radisson \(1640-1710\)](#). You can see that the ability to automatically generate refined searches can make it much easier to find things that are relevant to your work.

## Suggested Readings

Lutz, *Learning Python*

Re-read and review Chs. 1-17

## 7. Keyword in context (KWIC)

### N-grams

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. What we're going to do next is develop the ability to display keywords in context (often abbreviated as KWIC). Given a text and a keyword, your program will list every occurrence of the keyword in the 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. We will also use the output of our KWIC routine as the basis for a number of automatically generated Google searches.

As you work with digital representations, you will come to see that many of them have a hierarchical, or tree-like structure. This is also true of many elements of natural language, particularly at the level of phrases and sentences. People who study the statistical properties of language have found, however, that there is much to be learned by studying linear sequences of linguistic units. These are known as *bigrams* (2 units), *trigrams* (3 units), or more generally as *n-grams*. In any given text, different n-grams will occur with different frequencies. Since natural languages have a certain amount of built-in redundancy, even very large samples of text in a particular language will exhibit a distribution of different frequencies for different n-grams. We will unpack this idea as we go, but for now, we note that in English texts, q is almost always followed by u. In a sense that can be made mathematically precise, if you know that the current character is q, you can predict that the next one will be u with a high degree of certainty. Another way of saying this is that the bigram qu occurs much more frequently in English than qa, qb, and so on.

### From text to n-grams

Suppose you have a string like 'it was the best of times it was the worst of times it was the age of wisdom it was the age of foolishness'. You already know how to turn a string into a list using the *split* operation. In Python, you can retrieve a subsequence of a list by using what is called a *slice*, represented as two indexes separated by a colon. If the first index is empty, it is assumed to be the beginning of the list. If the second index is empty, it is assumed to be the end. Study the following examples.

```
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', 'of', 'foolishness']
```

We can combine the slice with a list comprehension to create a function that slides an n-gram window across a word list to create a list of n-grams. Copy the following function to your *dh.py* module.

```
# 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))]
```

If you want to test the function, you can paste the definition into a Python shell. Note that it returns an empty list if you ask for n-grams longer than the total length of your input.

```
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']]
```

## Making an n-gram dictionary

Since we want to show our keyword in the context of neighboring terms, we are going to use an n-gram window with an odd-numbered length (3,5,7,...). The next step is to put each n-gram in a dictionary, using the middle word as the key. Since Python indexes start at 0, we can compute the index of this middle word by dividing n by two and losing the remainder. If we are working with 7-grams, for example, the left context will consist of terms indexed by 0, 1, 2, the keyword will be indexed by 3, and the right context terms indexed by 4, 5, 6. Study the following function before adding it to the *dh.py* module. In particular, you should be clear about what happens when a particular keyword appears in more than one context. Notice also that we determine n by measuring the length of the first item in the list of n-grams, and from n we calculate the index of the keyword using Python's *floor division* operator.

```
# Given a list of n-grams, return a dictionary of KWICs,  
# indexed by keyword.
```

```
def nGramsToKWICDict(ngrams):  
    kwicdict = {}  
    keyindex = len(ngrams[0]) // 2  
    for k in ngrams:  
        if k[keyindex] not in kwicdict:  
            kwicdict[k[keyindex]] = [k]
```

```
    else:
        kwicdict[k[keyindex]].append(k)
    return kwicdict
```

## 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:

```
        killed by the   iroquois   at the long
        episode in the  iroquois   wars it was
           of 1660 the   iroquois   having sent forth
    terror among the    iroquois   by such a
        shape when 300   iroquois   burst forth along
           side of the   iroquois   dollard and his
                        ...
```

To get this effect, we are going to need to do a number of list and string manipulations. At this point, you will probably want to open a Python shell so you can experiment a bit. Let's start by figuring out the *slices* of the list that give us the keyword, left-hand context and right-hand context.

```
kwic = 'killed by the iroquois at the long'.split()
n = len(kwic)
print n
```

```
-> 7
```

```
keyindex = n // 2
print keyindex
```

```
-> 3
```

```
print kwic[:keyindex]
```

```
-> ['killed', 'by', 'the']
```

```
print kwic[keyindex]
```

```
-> iroquois
```

```
print kwic[(keyindex+1):]
```

```
-> ['at', 'the', 'long']
```

Now we need to format each of the three columns of our display. The right-hand context is simply going to consist of a string of terms separated by blank spaces.

```
print ' '.join(kwic[(keyindex+1):])
```

```
-> at the long
```

We want the keywords to have a bit of *whitespace* padding around them. We can achieve this by using a string method called *center* and having 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) + '#'
-> #   iroquois   #
```

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) + '#'
-> #           killed by the#
```

We can now combine these into a function that takes a KWIC and returns a pretty-printed string. Add this to the *dh.py* module.

```
# 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
```

## From HTML to KWIC

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. Copy the code to Komodo Edit, save it as *html-to-kwic.py*, and execute it.

```
# html-to-kwic.py

import dh

# create dictionary of n-grams
n = 7
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'
text = dh.webPageToText(url)
fullwordlist = ('# ' * (n//2)).split()
fullwordlist += dh.stripNonAlphaNum(text)
fullwordlist += ('# ' * (n//2)).split()
ngrams = dh.getNGrams(fullwordlist, n)
worddict = dh.nGramsToKWICDict(ngrams)

# output KWIC and wrap with HTML
target = 'iroquois'
outstr = '<pre>'
if worddict.has_key(target):
```

```

    for k in worddict[target]:
        outstr += dh.prettyPrintKWIC(k)
        outstr += '<br />'
else:
    outstr += 'Keyword not found in source'
outstr += '</pre>'
dh.wrapStringInHTML('html-to-kwic', url, outstr)

```

There are a few things in this code that need to be explained. We want the first few words and last few words in a text to show up when we generate a KWIC display. So we create some padding to put at the beginning and end of our word list. Say we are searching for the keyword 'dictionary' using a 7-gram context and it is the first word in the text. Then, since we are using hash marks for padding, the KWIC might be something like:

```

# # #    dictionary    of canadian biography

```

We're never going to need more than  $n/2$  words of padding. The second thing we'd like to point out is that we've wrapped our display in the HTML *pre* tag to indicate that our material is preformatted... we don't want the browser monkeying around with our whitespace after we've gone to such lengths to get it right. Finally, 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.

## Turning each KWIC into a Google search link

As with the previous example, we can introduce one more useful revision at this point. Why not use each KWIC as the basis for a possible Google search? Add the following routine to *dh.py*. We will discuss the *style* property of the HTML *a* tag in the next section.

```

# Given a list of keywords and a link name, return an
# HTML link to a Google search for those terms.

def keywordListToGoogleSearchLink(keywords, linkname):
    gsearch = '<a style=\"text-decoration:none\" '
    gsearch += 'href=\"http://www.google.com/search?q='
    gsearch += '+'.join(keywords)
    gsearch += '\">'
    gsearch += linkname
    gsearch += '</a>'
    return gsearch

```

Although Google tends to filter out stop words automatically, we can use our own routine to do this before creating the search link. The reason we want to filter stop words is because Google puts a limit on the number of search terms that it will pay attention to. If we have a long context for each of our keywords (say a 9-, 11-, or 13-gram) we want every meaningful word to count in our search. The revised version of our code is shown below. Copy it to Komodo Edit, save it as *html-to-kwic-2.py* and execute it.

```

# html-to-kwic-2.py

import dh

# create dictionary of n-grams
n = 7
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'
text = dh.webPageToText(url)
fullwordlist = ('# ' * (n//2)).split()
fullwordlist += dh.stripNonAlphaNum(text)
fullwordlist += ('# ' * (n//2)).split()

```

```

ngrams = dh.getNGrams(fullwordlist, n)
worddict = dh.nGramsToKWICDict(ngrams)

# output KWIC and wrap with HTML
target = 'iroquois'
outstr = '<pre>'
if worddict.has_key(target):
    for k in worddict[target]:
        linkname = dh.prettyPrintKWIC(k)
        keywords = dh.removeStopwords(k, dh.stopwords)
        ostr += dh.keywordListToGoogleSearchLink(keywords, linkname)
        ostr += '<br />'
else:
    ostr += 'Keyword not found in source'
ostr += '</pre>'
dh.wrapStringInHTML('html-to-kwic-2', url, ostr)

```

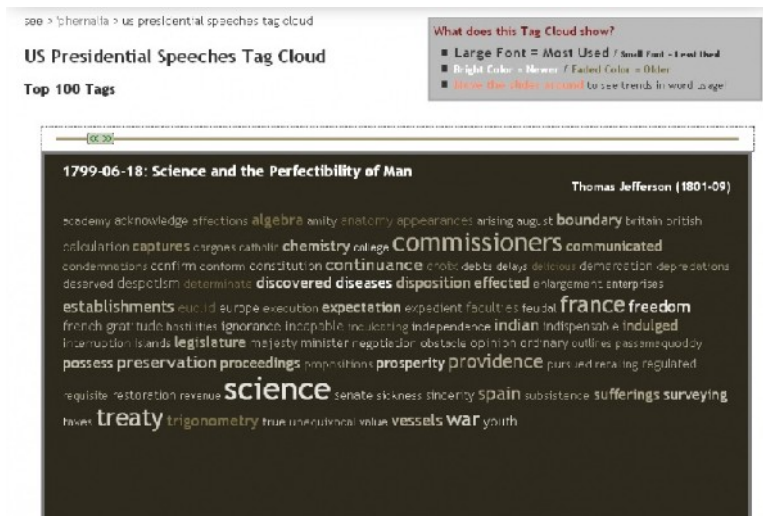
Once the output opens in Firefox, try running the mouse over different lines in the display and checking the bottom bar of your browser (over to the left of where you click to open the Zotero pane). As you pick each hyperlink, you can compare the KWIC line to the Google search that your program constructed.

Try clicking on the KWIC line that reads "ambuscades for the iroquois when returning from"... it is the twentieth one from the top. You will find a number of potentially interesting documents, including links to JSTOR articles and to transcriptions of various volumes of the *Jesuit Relations* and the works of Francis Parkman. As a historian, you'll want to evaluate these sources critically. Do they add anything that you didn't already know? Are they authoritative? Can you find more reliable versions? Should you add them to Zotero and use them as the basis for further search and analysis? As a programming historian, you have more evidence that digital sources can be profitably 'read' by both human beings and machines.

## 8. Tag clouds

### Visualizing term frequency

[Web 2.0](#) has made the *tag cloud* an ubiquitous form of text visualization. To create a tag cloud for a text, you first remove stop words, then take a couple dozen of the most frequently occurring terms, alphabetize them, and render each so that its font size is proportional to the number of times that it occurs in the text. The result typically looks something like the following:



This example is taken from [Chirag Mehta's US Presidential Speeches Tag Cloud](#) website. The site shows tag clouds for various historic presidential speeches, beginning in 1776 and running through George W. Bush's State of the Union addresses. You move through speeches by adjusting a horizontal time-line slider along the top. The display responds by providing a tag cloud for each speech. As you move back and forth through time, you can see different words become more or less prominent. Mehta also uses color to indicate the relative newness of a given term, fading from white to brown over time. Take a few moments to familiarize yourself with Mehta's website, to get some idea of the potential of this kind of visualization. We're going to start by visualizing the term frequencies of a single text; later you'll learn how to extend the method to synchronous or diachronous collections of texts.

## Mapping one range onto another

You already did most of the work that you'll need to make a tag cloud visualization when you learned how to compute word frequencies. We start by deciding how many terms we want to have in our cloud. Usually this will be somewhere on the order of 30 to 100. You want to show enough terms to capture the most interesting aspects of your text, but not so many that the distinctive features are obscured by noise. Once the size of the tag cloud is determined, you take that many elements from the top of the sorted dictionary of word-frequency pairs. Suppose we're working with Dollard's biography and we've decided that we want the size of our tag cloud to be 100 elements. Then we take the top 100 items from our dictionary. The most frequent term, 'dollard', occurs 91 times. The least frequent term occurs 6 times. Subtracting one from the other, we have a frequency range of 85.

```
cloudsize = 100
maxfreq = sorteddict[0][0]
minfreq = sorteddict[cloudsize][0]
freqrange = maxfreq - minfreq
```

Now we need to map this range onto a range of possible font sizes. We've chosen 24 pixels as our smallest font and 54 as our largest, giving us a range of 30 font sizes. So we need to map 85 different frequencies down to 30 different fonts. We could work with these exact values, but it makes more sense to come up with a general formula.

Let's start with the ends of our ranges. We want a frequency of 6 to be mapped to a font size of 24, or, more generally, we want *minfreq* to be mapped to *minfont*. Likewise, we want *maxfreq* to be mapped to *maxfont*.

Now consider the second-most frequent term, 'iroquois', which occurs 64 times in Dollard's biography. We first need to determine what proportion of the way it is between *minfreq* and *maxfreq*. In this case, it is  $(64 - 6) / 85 = 58 / 85 = 0.68235$ . In other words, 'iroquois' occurs about 68% of the way between the least frequent term and 'dollard'. More generally, let's define a frequency *scalingfactor* for term *k* as:

```
scalingfactor = (kfreq - minfreq) / float(freqrange)
```

(We have to use the *float* function to tell Python that we're working with floating point numbers and it shouldn't convert everything to integers.)

That deals with the range on the frequency side of our mapping. On the font side, we will use a similar logic:

```
import math
minfont = 24
maxfont = 54
fontrange = maxfont - minfont
fontsize = int(minfont + math.floor(fontrange * scalingfactor))
```

The *floor* function in the math module rounds a floating point number down to the nearest integer.

## A little bit of CSS

In the early days of the web, HTML tags specified both what something was (e.g., a title or a paragraph) and how something should look (e.g., italics or boldface). This made it difficult to write web pages that would look good and be usable on a variety of different computers. A very large font might make a good headline on a wide screen, but not fit on a narrow one. A page that looks good in color might be illegible when printed on a laser printer. To get a sense of the problem, compare the [Google home page](#) with the [mobile version](#) of the page that is designed to be used on devices like cellphones. The obvious solution was to separate the content of a page from its form. HTML tags are now generally used to specify what something is, and [Cascading Style Sheets \(CSS\)](#) are used to specify how it should look under various conditions.

You've already seen one example of CSS, when we used the *style* property of an HTML a tag to indicate that we didn't want the browser to underline hyperlinks. Putting CSS into HTML tags like this is known as *inline* CSS. It sort of defeats the original purpose of separating form and content, but it will make it easier for us to do so later on.

```
<a style="text-decoration:none" href=" ... "> ... </a>
```

We can bundle this into a function that returns the HTML a tag as a string, given a url and link name. Add the following function to the *dh.py* module.

```
# Given a url and link name, return a string containing
# HTML and inline CSS for an undecorated hyperlink.
```

```
def undecoratedHyperlink(url, linkname):
    astr = """<a
        style=\"text-decoration:none\" href=\"%s\">%s</a>
        """
    return astr % (url, linkname)
```

Given the *undecoratedHyperlink* function, we can rewrite our *keywordListToGoogleSearchLink* function, too. Replace the old version in *dh.py* with the following version:

```
# Given a list of keywords and a link name, return an
# HTML link to a Google search for those terms.
```

```
def keywordListToGoogleSearchLink(keywords, linkname):
    url = 'http://www.google.com/search?q='
    url += '+'.join(keywords)
    gsearch = undecoratedHyperlink(url, linkname)
    return gsearch
```

In order to apply a style to a small region of your web page, you usually use a *span* tag. For example, if you wanted to print a word in red in HTML, you could use either of the following expressions. You can test these two HTML expressions online with the W3 Schools [TryIt](#) editor.

```
this is in <span style="color: red;">red</span><br />
this is also in <span style="color: rgb(255,0,0);">red</span>
```

The first example uses a predefined color name. The second uses an RGB function that defines a color in terms of how much red, green and blue it contains, each on a scale ranging from 0 to 255. It is also possible to change font sizes using inline CSS. Try copying the following example into the HTML editor:

```

this <span style="font-size:8px;">word</span> is in an 8 pixel font<br />
this <span style="font-size:10px;">word</span> is in a 10 pixel font<br />
this <span style="font-size:12px;">word</span> is in a 12 pixel font<br />
this <span style="font-size:18px;">word</span> is in a 18 pixel font<br />
this <span style="font-size:24px;">word</span> is in a 24 pixel font<br />
this <span style="font-size:36px;">word</span> is in a 36 pixel font

```

To use a particular style for a large region of your page, you use the HTML *div* tag. Think of *div* like a generalization of a paragraph. In the program that we're developing, we are going to use *span* tags for each term in the tag cloud, but the whole cloud will be sitting inside of a *div*. Try the following code in the HTML editor. The CSS properties set the width of the *div* to 560 pixels, set the background color to a very light grey, put a solid one-pixel grey border around it, and center any text within it. As long as you end each property-value pair with a semi-colon, you can include as many as you want in the style.

```

<div style="width: 560px;
background-color: rgb(250,250,250);
border: 1px grey solid;
text-align: center;">
This is a test.
</div>

```

## Functions to write HTML divs and spans

We now have enough background information to write some Python functions that will automatically create HTML *div* and *span* tags. Copy the following function and add it to *dh.py*.

```

# Given the body of a div and an optional string of
# property-value pairs, return string containing HTML
# and inline CSS for default div.

```

```

def defaultCSSDiv(divbody, opt=''):
    divstr = """"<div style=\
width: 560px;
background-color: rgb(250,250,250);
border: 1px grey solid;
text-align: center;
%s\ ">%s</div>
""""
    return divstr % (opt, divbody)

```

If you copy this function to a Python shell and execute it, you can see that it allows you to add additional properties to a default *div*.

```

print defaultCSSDiv('This is a test', 'font-size: 24px;')

```

```

-> <div style=" width: 560px;
background-color: rgb(250,250,250);
border: 1px grey solid;
text-align: center;
font-size: 24px;">This is a test</div>

```

We will also bundle up our code to create a scaled font. Copy the following function into the *dh.py* module.

```

# Given the body of a span and a scaling factor, return
# string containing HTML span with scaled font size.

```

```

def scaledFontSizeSpan(body, scalingfactor):

```

```

import math
minfont = 24
maxfont = 54
fontrange = maxfont - minfont
fontsize = int(minfont + math.floor(fontrange * scalingfactor))
spanstr = '<span style=\"font-size:%spx;\">%s</span>'
return spanstr % (str(fontsize), body)

```

## Other dimensions for visualization

Given our *scaledFontSizeSpan* function, we now have the ability to map any scaling factor between 0 and 1 onto a range of font sizes that we've chosen. Font size is not the only variable that we could use for visualization, of course. Any range of potentially interesting differences in our data can be mapped onto any other range of perceptually salient properties. It is quite easy, for example, to create a function that adjusts the lightness or darkness of a greyscale font as well as the font size. Study the following and add it to *dh.py*.

```

# Given the body of a span and a scaling factor, return
# string containing HTML span with scaled font size and
# darkness of greyscale adjusted.

```

```

def scaledFontShadeSpan(body, scalingfactor):
    import math
    minfont = 24
    maxfont = 54
    fontrange = maxfont - minfont
    fontsize = int(minfont + math.floor(fontrange * scalingfactor))
    fontcolor = int(200 - math.ceil(200 * scalingfactor))
    spanstr = """<span style=\"font-size:%spx;
color: rgb(%d,%d,%d);
\">%s</span>
"""
    return spanstr % (str(fontsize), fontcolor, fontcolor, fontcolor, body)

```

When *rgb* is given three equal parameters, it returns a value between black (0, 0, 0) and white (255, 255, 255). In this case, we start with a relatively light grey (200, 200, 200) and subtract larger and larger values from it as the scaling factor increases. When the scaling factor is 1, our font color is black. Recall that the *%d* formatting character allows us to interpolate an integer into a string.

It is also possible to adjust each of the red, green and blue color components independently. The following function creates a *heat map*, shading from a cool blue when the scaling factor is 0, to a hot red when it is 1. Make sure that you understand how the code works, then add it to the *dh.py* module.

```

# Given the body of a span and a scaling factor, return
# string containing HTML span with scaled font size and
# shading from cool blue to hot red.

```

```

def scaledFontHeatmapSpan(body, scalingfactor):
    import math
    minfont = 24
    maxfont = 54
    fontrange = maxfont - minfont
    fontsize = int(minfont + math.floor(fontrange * scalingfactor))
    fontcolor = int(250 - math.ceil(250 * scalingfactor))
    spanstr = """<span style=\"font-size:%spx;
color: rgb(%d,0,%d);
\">%s</span>
"""
    return spanstr % (str(fontsize), 250-fontcolor, fontcolor, body)

```

## Putting it all together

We can now write a program that makes use of these functions (and ones that we've written previously) to create a tag cloud for Dollard's biography. First add the following to `dh.py`:

```
# Given a dictionary of frequency-word pairs sorted
# in order of descending frequency, re-sort so it is
# in alphabetical order by word.
```

```
def reSortFreqDictAlpha(sorteddict):
    import operator
    aux = [pair for pair in sorteddict]
    aux.sort(key=operator.itemgetter(1))
    return aux
```

Now copy the following code to Komodo Edit, save it as `html-to-tag-cloud.py` and execute it.

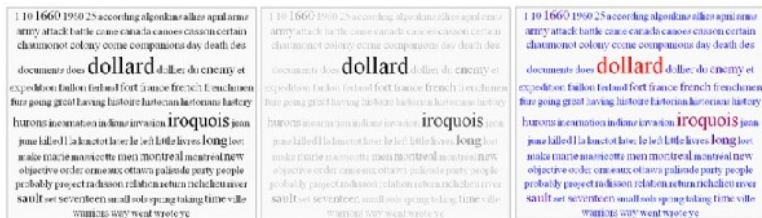
```
# html-to-tag-cloud.py

import dh

# create sorted dictionary of word-frequency pairs
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'
text = dh.webPageToText(url)
fullwordlist = dh.stripNonAlphaNum(text)
wordlist = dh.removeStopwords(fullwordlist, dh.stopwords)
dictionary = dh.wordListToFreqDict(wordlist)
sorteddict = dh.sortFreqDict(dictionary)

# create tag cloud and open in Firefox
cloudsize = 100
maxfreq = sorteddict[0][0]
minfreq = sorteddict[cloudsize][0]
freqrange = maxfreq - minfreq
outstring = ''
resorteddict = dh.reSortFreqDictAlpha(sorteddict[:cloudsize])
for k in resorteddict:
    kfreq = k[0]
    klabel = k[1]
    scalingfactor = (kfreq - minfreq) / float(freqrange)
    outstring += ' ' + dh.scaledFontSizeSpan(klabel, scalingfactor) + ' '
dh.wrapStringInHTML("html-to-tag-cloud", url, dh.defaultCSSDiv(outstring))
```

If you substitute `dh.scaledFontShadeSpan` or `dh.scaledFontHeatmapSpan` for `dh.scaledFontSizeSpan` you can create any of three different tag cloud visualizations like the ones show below. You can also modify the code to change the range of font sizes, the typeface, the background color of the `div`, the width of the border, and anything else you can think of.



Three Tag Cloud Visualizations

## Combining the tag cloud with KWIC

We're going to make one final refinement to this program, combining it with our ability to generate keyword in context (KWIC) displays. We want to create a web page with a tag cloud at the top and a number of KWIC displays in alphabetical order. When you click on a particular term, you're taken down the page to the KWIC display for that term. There is a link you can click to return to the tag cloud at the top. The code for this program is listed below. Much of it will be familiar, but it makes use of a few new techniques. Copy it to Komodo Edit, save as *html-to-tag-cloud-kwic.py* and execute it.

```
# html-to-tag-cloud-kwic.py

import dh

# create sorted dictionary of word-frequency pairs
url = 'http://niche-canada.org/files/dcb/dcb-34298.html'
text = dh.webPageToText(url)
fullwordlist = dh.stripNonAlphaNum(text)
wordlist = dh.removeStopwords(fullwordlist, dh.stopwords)
dictionary = dh.wordListToFreqDict(wordlist)
sorteddict = dh.sortFreqDict(dictionary)

# create dictionary of n-grams
n = 7
paddinglist = ('# ' * (n//2))
fullwordlist[:0] = paddinglist
fullwordlist.extend(paddinglist)
ngrams = dh.getNGrams(fullwordlist, n)
worddict = dh.nGramsToKWICDict(ngrams)

# create tag cloud
cloudsize = 40
maxfreq = sorteddict[0][0]
minfreq = sorteddict[cloudsize][0]
freqrange = maxfreq - minfreq
tempstring = ''
resorteddict = dh.reSortFreqDictAlpha(sorteddict[:cloudsize])
for k in resorteddict:
    kfreq = k[0]
    klabel = dh.undecoratedHyperlink('#'+k[1], k[1])
    scalingfactor = (kfreq - minfreq) / float(freqrange)
    tempstring += dh.scaledFontSizeSpan(klabel, scalingfactor)
outstring = dh.defaultCSSDiv(tempstring) + '<br />'

# create KWIC listings for each item
for k in resorteddict:
    klabel = k[1]
    tempstring = ''
    tempstring += '<a name="%s">%s</a> ' % (klabel, klabel)
    tempstring += dh.undecoratedHyperlink('#', '[back]')
    outstring += dh.defaultCSSDiv(tempstring, opt='font-size : 24px;')
    outstring += '<p><pre>'
    for t in worddict[klabel]:
        outstring += dh.prettyPrintKWIC(t)
        outstring += '<br />'
    outstring += '</pre></p>'

# open in Firefox
dh.wrapStringInHTML("html-to-tag-cloud-kwic", url, outstring)
```

The first section of the code creates a sorted dictionary of word-frequency pairs, as you've done before. In the

second section, we need to add a list containing  $(n//2)$  padding characters (hash marks) to the beginning and end of our full list of words before creating a dictionary of n-grams. We use two new commands to do this. The first replaces the empty slice before the beginning of the list with the list of padding characters. The second extends the word list by adding the list of padding characters to the end of it. To experiment with this, open a Python shell and try the following:

```
testlist = 'this is a test'.split()
print testlist

-> ['this', 'is', 'a', 'test']

testlist[:0] = ['i', 'say']
print testlist

-> ['i', 'say', 'this', 'is', 'a', 'test']

testlist.extend(['is', 'it', 'not'])

-> None

print testlist

-> ['i', 'say', 'this', 'is', 'a', 'test', 'is', 'it', 'not']
```

Note that the extend function changes the list, but doesn't return a value (hence the 'None'). This is called *changing a list in place*.

In the third section we create a tag cloud. Each term in the cloud has a hyperlink that looks like this:

```
<a style="text-decoration:none" href="#1660">1660</a>
```

Putting a hash mark in front of a string creates a *relative link*, a link to a place within the current web page. (Don't confuse this with our use of the hash mark as a padding character above. The two are completely unrelated.) Each of these links has a corresponding *named anchor* farther down the page. The anchor looks like this:

```
<a name="1660">1660</a>
```

We also automatically create a number of links which take you back to the top of the page. Each of them looks like this:

```
<a style="text-decoration:none" href="#">[back]</a>
```

At this point it is probably a good idea to spend a few minutes studying the HTML file that this program created automatically. It is named *html-to-tag-cloud-kwic.html*.

## 9. Harvesting links and downloading pages

### The idea of text mining

The programs that we've written so far take a single web page or text as their input and do some processing that would be time-consuming to do by hand. Since historians are used to reading a lot, this may have seemed like a questionable exercise... wouldn't it be faster to read Dollard's biography, for example, than spend time



In Firefox, choose File->Save Page As and save this page as *dcb-v01-iroquois.html* using the "Web Page, HTML only" option. Now you should be able to use File->Open File in Firefox to make sure that you got a copy of the page. It should look like this:



Note that your local copy of the page is missing the images that the online version used for formatting. If you had saved a complete copy of the page, rather than HTML only, all of those images would have been downloaded to a directory on your machine. You don't need them, however.

## Extracting hyperlinks with Beautiful Soup

Our next task is to extract all of the hyperlinks from the saved copy of the web page, so we can then write a routine to download each biography automatically. Recall that HTML has a hierarchical structure. The problem of taking apart a structured representation in an orderly way is known as *parsing*. When the rules by which the structure was created are well-known and inflexible, parsing is easier. When there are a lot of exceptions or errors, parsing becomes more difficult and programmers sometimes turn to *scraping* instead. Rather than taking the structure apart, scraping relies on regular expression pattern matching to pull meaningful strings out of an undifferentiated mass. We're going to do a little of both.

Later you'll learn more about how to create your own parsers. For right now, we're going to use a Python library called [Beautiful Soup](#). If you haven't already installed it, download this package to your machine and save it in the `C:\Python25\Lib` directory.

To parse out all of the HTML tags, we first load our local copy of the web page into a string:

```
# load search results from saved file into string
searchresultfile = 'dcb-v01-iroquois.html'
f = open(searchresultfile, 'r')
searchresulthtml = f.read()
f.close()
```

We then call Beautiful Soup to extract all of the tags into a list. This version of the Python import statement

allows us to load the part of the library that we need to parse HTML, without loading the part that parses XML.

```
from BeautifulSoup import BeautifulSoup

# parse search results file to extract hyperlinks
searchresultsoup = BeautifulSoup(searchresulthtml)
linklist = searchresultsoup.findAll('a')

for link in linklist: print link
```

If you were to copy this code to Komodo and execute it, you would find that there are a number of links in the page that aren't of interest to us, in addition to ones that are:

```
<a name="TOP"></a>
<a href="../../../FR/index.html">
  </a>
<a href="mailto:WebServices@lac-bac.gc.ca?Subject=www.biographi.ca">
  </a>
...
<a href="ShowBio.asp?BioId=34298&query=iroquois">DOLLARD DES ORMEAUX, ADAM</a>
<a href="ShowBio.asp?BioId=34146&query=iroquois">ANNAOTAHA, Étienne</a>
<a href="ShowBio.asp?BioId=34590&query=iroquois">PIESKARET, Simon</a>
...
```

The links that we want to process all have the form

```
<a href="ShowBio.asp?BioId=      BIOID      &query=iroquois">      BIONAME      </a>
```

where BIOID is a five digit number and BIONAME is the person whose biography it is.

## Scraping with regular expressions

We now want to go through each link in *linklist* and use a regular expression to see if it matches the form shown above. Let's start by trying to match a five digit number. Open a Python shell so you can try the following expressions. Note that `\d` matches a single digit. Adding a number in curly braces after a pattern matches that many copies of it. So `\d{5}` matches a string of five digits in a row. The *search* method finds a match if one exists.

```
import re
digitpattern = re.compile(r'\d{5}')
print digitpattern.search('abc')

-> None

print digitpattern.search('123')

-> None

print digitpattern.search('123456')

-> <_sre.SRE_Match object at 0x0632F5D0>
```

Note the weird return value when it does find a match. What we really want to return is the match itself. For

that, we use the `group(0)` method. This will make more sense in a minute.

```
print digitpattern.search('123456').group(0)
```

```
-> 12345
```

Remember that regular expressions find matching patterns in a larger string. When you want to return the part that matches, but not any of the extraneous material, you put parentheses around the matching part. These matching parts are known as *groups*. Study the following examples, keeping in mind that `*` stands for zero or more copies of any single character. What would each of the two expressions return if `teststring` were 'junkjunkjunk'? Try this in the shell to make sure you understand what is going on.

```
digitpattern2 = re.compile(r'.*(\d{5}).*')
teststring = 'junk12345junk'
print digitpattern2.search(teststring).group(0)
```

```
-> junk12345junk
```

```
print digitpattern2.search(teststring).group(1)
```

```
-> 12345
```

Grouping a regular expression with parentheses like this allows us to indicate which part of a matching string is important to us. To match the whole thing, we use `group(0)`. The first part that is in parentheses is `group(1)`, the second part `group(2)`, and so on. Since we can have multiple groups, we are able to match both the BIOID and BIONAME. (Note that we have to escape the quotation marks within our test strings by preceding them with backslashes).

```
linkpattern = re.compile(r'(\d{5}).*>(.*?)<', re.UNICODE)
print linkpattern.search('<a name=\"TOP\"></a>')
```

```
-> None
```

```
testurl = '<a href=\"ShowBio.asp?BioId=34590&query=iroquois\">PIESKARET, Simon</a>'
print linkpattern.search(testurl).group(1)
```

```
-> 34590
```

```
print linkpattern.search(testurl).group(2)
```

```
-> PIESKARET, Simon
```

If you'd like to learn more about Python regular expressions, A. M. Kuchling has written a good [tutorial](#).

## Working with accented characters

If you only work with English-language sources, you usually don't have to deal with accented characters. People who work with sources in languages that use non-Latin alphabets or non-alphabetic writing systems will have to know more about how to represent these characters. Our sources include some French characters, so we need to make sure to represent them in a uniform way. You can generalize our routine to include other characters from the [latin-1](#) or [utf-8](#) character sets as necessary. If you will need to do this on a regular basis, you should spend some time now getting more familiar with Unicode and read the section on Unicode strings in the Python tutorial. There is also a very useful reference on [Computing with Accents, Symbols and Foreign Scripts](#) from Penn State.

Our task is complicated by the fact that both HTML and Unicode provide different ways to represent accented characters, and our source mixes and matches the two. The following routine converts a string to lowercase and then maps each accented character from the French language to its lowercase Unicode equivalent. Copy it to the *dh.py* module.

```
# Given a string containing French accented characters
# in Unicode or HTML, return normalized lowercase.

def normalizeFrenchAccents(str):
    newstr = unicode(str, 'utf-8').encode('latin-1', 'replace')
    newstr = newstr.lower()
    newstr = newstr.replace('&rsquo;', '\')
    newstr = newstr.replace('\xc0', '\xe0') # a grave
    newstr = newstr.replace('&agrave;', '\xe0') # a grave
    newstr = newstr.replace('\xc2', '\xe2') # a circumflex
    newstr = newstr.replace('&acirc;', '\xe2') # a circumflex
    newstr = newstr.replace('\xc4', '\xe4') # a diaeresis
    newstr = newstr.replace('&auml;', '\xe4') # a diaeresis
    newstr = newstr.replace('\xc6', '\xe6') # ae ligature
    newstr = newstr.replace('&aelig;', '\xe6') # ae ligature
    newstr = newstr.replace('\xc8', '\xe8') # e grave
    newstr = newstr.replace('&egrave;', '\xe8') # e grave
    newstr = newstr.replace('\xc9', '\xe9') # e acute
    newstr = newstr.replace('&eacute;', '\xe9') # e acute
    newstr = newstr.replace('\xca', '\xea') # e circumflex
    newstr = newstr.replace('&ecirc;', '\xea') # e circumflex
    newstr = newstr.replace('\xcb', '\xeb') # e diaeresis
    newstr = newstr.replace('&euml;', '\xeb') # e diaeresis
    newstr = newstr.replace('\xce', '\xee') # i circumflex
    newstr = newstr.replace('&icirc;', '\xee') # i circumflex
    newstr = newstr.replace('\xcf', '\xef') # i diaeresis
    newstr = newstr.replace('&iuml;', '\xef') # i diaeresis
    newstr = newstr.replace('\xd4', '\xf4') # o circumflex
    newstr = newstr.replace('&ocirc;', '\xf4') # o circumflex
    newstr = newstr.replace('&oeelig;', 'oe') # oe ligature
    newstr = newstr.replace('\xd9', '\xf9') # u grave
    newstr = newstr.replace('&ugrave;', '\xf9') # u grave
    newstr = newstr.replace('\xdb', '\xfb') # u circumflex
    newstr = newstr.replace('&ucirc;', '\xfb') # u circumflex
    newstr = newstr.replace('\xdc', '\xfc') # u diaeresis
    newstr = newstr.replace('&uuml;', '\xfc') # u diaeresis
    newstr = newstr.replace('\xc7', '\xe7') # c cedilla
    newstr = newstr.replace('&ccedil;', '\xe7') # c cedilla
    newstr = newstr.replace('&yuml;', '\xff') # y diaeresis
    return newstr
```

## Some helper functions

Given what we know, we can write some code to extract BIODID-BIONAME pairs to a dictionary.

```
# extract dictionary of bioid-name pairs
linkpattern = re.compile(r'(\d{5}).*\>(.*)\<', re.UNICODE)
bioidict = {}
for i in linklist:
    matchinglink = linkpattern.search(str(i))
    if matchinglink:
        bioid = matchinglink.group(1)
        bioname = matchinglink.group(2)
        bioidict[bioid] = dh.normalizeFrenchAccents(bioname)
```

We are also going to want to be able to do a few things with our local file system. We're going to need a separate directory to store our downloaded files in, and if it doesn't exist, we're going to have to create it. The file system is part of the operating system on your computer. Python includes an `os` library to access it.

```
import os

# make directory to store downloaded pages if one doesn't exist
if os.path.exists('iroquois') == 0: os.mkdir('iroquois')
```

Before downloading a `file`, we will also want to be able to see `if` it already exists...

```
outfile = 'iroquois/dcb-' + str(b) + '.html'
if os.path.isfile(outfile) == 0:
    # outfile doesn't already exist
```

We will want to introduce a time delay into our program, so that it waits for a while before trying to download the next file. Since programs can request pages from a web server much faster than human users, it is considered polite to separate automatic requests with a time delay. Python includes a `time` library which gives access to a number of different timing functions.

```
import time

# pause for two seconds
time.sleep(2)
```

Finally, we will be keeping track of our program's progress by sending messages to the "Command Output" pane of Komodo. Instead of sending one character at a time to the output, the Python interpreter uses a *buffering* strategy. It puts the characters in a temporary holding space until the space is full, then sends them all to the output at once. This is more efficient, but it means that you don't have immediate feedback about what your program is up to. Fortunately, the Python `sys` module gives us the ability to *flush a buffer* whenever we want.

```
import sys

# send feedback to the "Command Output" pane immediately
print "File already downloaded"
sys.stdout.flush()
```

## Putting it all together

Our program will perform the following tasks

1. Load a number of Python modules
2. Load the search results from a local saved file into a string
3. Parse the search results to extract all HTML a tags
4. Scrape the BIOID-BIONAME pairs from the a tags and put them in a dictionary
5. Make a directory to store the local copies of the files if one doesn't already exist
6. For each BIOID, check to see if the file already exists
  1. If not, download it and save a local copy then wait two seconds
  2. Otherwise print a message that the file has already been downloaded
  3. Flush the "Command Output" buffer so the user gets immediate feedback
7. Create a page of links to local copies of the biographies and open in Firefox

Here is the code that accomplishes these tasks.

```

# get-iroquois-bios.py

import dh
import re, os, sys, time, urllib2
from BeautifulSoup import BeautifulSoup

# load search results from saved file into string
searchresultfile = 'dcb-v01-iroquois.html'
f = open(searchresultfile, 'r')
searchresulthtml = f.read()
f.close()

# parse search results file to extract hyperlinks
searchresultsoup = BeautifulSoup(searchresulthtml)
linklist = searchresultsoup.findAll('a')

# extract dictionary of bioid-name pairs
linkpattern = re.compile(r'(\d{5}).*\>(.*)\<', re.UNICODE)
biodict = {}
for i in linklist:
    matchinglink = linkpattern.search(str(i))
    if matchinglink:
        bioid = matchinglink.group(1)
        bioname = matchinglink.group(2)
        biodict[bioid] = dh.normalizeFrenchAccents(bioname)

# make directory to store downloaded pages if one doesn't exist
if os.path.exists('iroquois') == 0: os.mkdir('iroquois')

# download a local copy of each bio
urlprefix = 'http://www.biographi.ca/EN/ShowBioPrintable.asp?BioId='
for b in biodict:
    print "Processing bioid: " + str(b)
    url = urlprefix + str(b)
    outfile = 'iroquois/dcb-' + str(b) + '.html'
    if os.path.isfile(outfile) == 0:
        response = urllib2.urlopen(url)
        html = response.read()
        f = open(outfile, 'w')
        f.write(html)
        f.close()
        time.sleep(2)
    else:
        print "File already downloaded"
        sys.stdout.flush()

# create a page of links to local copies
outstring = ''
for b in biodict:
    outfile = 'dcb-' + str(b) + '.html'
    outstring += dh.undecoratedHyperlink('iroquois/'+outfile, str(b))
    outstring += '&nbsp;' * 4
    outstring += biodict[b]
    outstring += "<br />"
dh.wrapStringInHTML("get-iroquois-bios", searchresultfile, outstring)

```

Copy the code to Komodo, save it as *get-iroquois-bios.py* and execute it. Sometimes the server will choke for some reason and your program will halt with an error message. You can simply re-run it. It will skip over the biographies that it has already downloaded and carry on where it stopped. If a downloaded file is garbled for some reason, you can delete the file and rerun this program. We've designed it to take into account the fact that things sometimes go wrong when you're automatically harvesting online data.

If all goes well, the program should download the 167 biographies from volume 1 of the *DCB* to a directory called *iroquois* on your local disk. It will also open a page in Firefox with links to each of the downloaded files. You're now ready to learn how to index a collection of documents.

## 10. Indexing a document collection

### An overview

In an earlier lesson, we wrote code that harvested hyperlinks from a web page and then followed each link to download a local copy of the file pointed to. As a result we now have a directory called *iroquois* which contains 167 biographies from volume 1 of the *Dictionary of Canadian Biography*. The next thing we would like to do is create a combined index for these files. Ideally, the index should enable us to find any instance of a given word in all of the files in which it appears. So we need to keep track of files which contain a particular term, and we also want to know all of the locations in a file where that term can be found. Since we don't have page numbers to use for indexing purposes, we will use a word's position in the file instead.

Suppose a file (e.g., *dcb-34125.html*) begins like this

```
Dictionary of Canadian Biography AGARIATA (Agoriata), Mohawk chief; fl. 1666.
Agariata's career can only be reconstructed, without too much certainty, from
fragmentary and conflicting accounts (Bacqueville de La Potherie [Le Roy*]
alone gives him a name). He steps into history ...
```

We start by cleaning up the file a little bit to regularize capitalization, punctuation, and so on. Then we can number each word in the file, beginning with the first.

```
1:dictionary 2:of 3:canadian 4:biography 5:agariata 6:agoriata 7:mohawk
8:chief 9:fl 10:1666 11:agariata 12:s 13:career 14:can 15:only 16:be
17:reconstructed 18:without 19:too 20:much 21:certainty 22:from
23:fragmentary 24:and 25:conflicting 26:accounts 27:bacqueville 28:de
29:la 30:potherie 31:le 32:roy 33:alone 34:gives 35:him 36:a 37:name
38:he 39:steps 40:into 41:history ...
```

We don't want to bother indexing stopwords, but we can't simply remove them as we have in the past, because that would throw off our positional indexing system. Instead we will replace each stopword with a special placeholder that we aren't going to index. Compare the following version with the one given above.

```
1:dictionary 2:# 3:canadian 4:biography 5:agariata 6:agoriata 7:mohawk
8:chief 9:fl 10:1666 11:agariata 12:# 13:career 14:# 15:# 16:#
17:reconstructed 18:# 19:# 20:# 21:certainty 22:#
23:fragmentary 24:# 25:conflicting 26:accounts 27:bacqueville 28:#
29:la 30:potherie 31:le 32:roy 33:# 34:gives 35:# 36:# 37:#
38:# 39:steps 40:# 41:history ...
```

We can see that the word "certainty" occurs at position 21 in file *dcb-34125.html*. If we were to store that information in a list, it would look like this

```
['certainty', 'iroquois/dcb-34125.html', 21]
```

Our goal is to find, and index, every instance of every significant word in our document collection, so we want each word to be associated with a list of tuples, as follows

```
'certainty'
```

```
[('iroquois/dcb-34125.html', 21), ('iroquois/dcb-34231.html', 8764),
('iroquois/dcb-34298.html', 4047), ('iroquois/dcb-34427.html', 1997),
('iroquois/dcb-34427.html', 2523), ('iroquois/dcb-34427.html', 2662),
('iroquois/dcb-34663.html', 10049)]
```

A collection of such lists will make up our index.

## Getting a list of filenames from a directory

Whenever we have a collection of documents that we want to process, we are going to have to know the name of each file. One possibility would be to code these names into our programs, as we have done in the past.

```
f = open('dcb-34125.html', 'r')
```

A better strategy is to store the names of each of the files that need to be processed in a list, and then work through them in a loop. So we need a function that uses the operating system to open a directory, get the names of all files in that directory and return them in a Python list. Copy the following into *dh.py*. **Note that operating system calls tend to be quite specific. This version works on Windows XP but will probably need to be modified to work on Mac or Linux.**

```
# Given a string containing the name of a directory
# return a list of files in that directory.
```

```
def getFileNames(dirname):
    import os
    dircommand = 'dir ' + dirname + ' /B'
    filelist = os.popen(dircommand).readlines()
    filelist = [x.rstrip() for x in filelist]
    return filelist
```

## Normalizing the files

Before we create our index, we want to get rid of all HTML markup, convert the text to lowercase, and remove all punctuation. We make use of the *stripTags* routine that we've already written to create a new function that converts an HTML page into lowercase text and replaces any non-breaking spaces (&nbsp;) with regular spaces. Copy the following and put it into *dh.py*.

```
# Given a local copy of a webpage, return string
# of lowercase text from page.
```

```
def localWebPageToText(webpage):
    f = open(webpage, 'r')
    html = f.read()
    f.close()
    text = stripTags(html).replace('&nbsp;', ' ')
    return text.lower()
```

Assuming that *fname* is a string which holds the file name that we're processing, we can call the function that we just wrote, followed by *stripNonAlphaNum*.

```
ftext = dh.localWebPageToText(fname)
flist = dh.stripNonAlphaNum(ftext)
```

This will give us a list of the words in a file, but we still need to replace stopwords with a special placeholder. The code that we'll use to do this introduces some powerful new programming techniques.

## Mapping an anonymous function over a list

So far when we've wanted to create a function, we've given our function a name and used the Python *def* statement to define it. This allowed us to reuse our routines in different programs. Python also allows us to create unnamed functions using a *lambda expression*. We will usually do this when we'd like to wrap up some code for convenience, but won't necessarily want to reuse it later on. Some examples will hopefully make the idea a bit more clear.

Suppose we want to double every number in a list. One possibility would be to create a special function to do this:

```
def doubleListElements(numlist):
    doubledlist = []
    for n in numlist:
        doubledlist.append(n+n)
    return doubledlist
```

```
inlist = [1, 2, 3, 7, 8, 9]
print doubleListElements(inlist)
```

```
-> [2, 4, 6, 14, 16, 18]
```

Alternately, we could write a function that doubles a single element, and use a loop to apply that function to each item in a list:

```
def doubleElement(x):
    return x+x
```

```
inlist = [1, 2, 3, 7, 8, 9]
```

```
doubledlist = []
for n in inlist:
    doubledlist.append(doubleElement(n))
print doubledlist
```

```
-> [2, 4, 6, 14, 16, 18]
```

Python includes a special function called *map* which makes it easy to apply another function to each element in a list and collect the results in a new list. We can rewrite the previous example as follows.

```
def doubleElement(x):
    return x+x
```

```
inlist = [1, 2, 3, 7, 8, 9]
```

```
doubledlist = map(doubleElement, inlist)
print doubledlist
```

```
-> [2, 4, 6, 14, 16, 18]
```

If we're simply going to use the *doubleElement* function for this purpose, it doesn't make sense to go to the trouble of giving it its own name. Instead, we can use *lambda* to create an anonymous function expression and pop it into the statement where we need it. The same example using *map* and *lambda* looks like this:

```
inlist = [1, 2, 3, 7, 8, 9]
print map((lambda x: x+x), inlist)
-> [2, 4, 6, 14, 16, 18]
```

The style of programming that relies on applying functions to sequences like this is known as *functional programming*. Before you go on, how would you modify the lambda expression above so that it squares each item rather than doubling it? Check your answer in a Python shell.

## Replacing stopwords with a placeholder

Let's make use of functional programming to replace the stopwords in our list with a placeholder. Suppose we have a list of the first 41 words in the file *dcB-34125.html*. Note that we don't have to explicitly number the words, because Python lists are already indexed (starting from 0, so `wordlist[20] = 'certainty'`).

```
wordlist = ['dictionary', 'of', 'canadian', 'biography',
'agariata', 'agoriata', 'mohawk', 'chief', 'fl', '1666',
'agariata', 's', 'career', 'can', 'only', 'be', 'reconstructed',
'without', 'too', 'much', 'certainty', 'from', 'fragmentary',
'and', 'conflicting', 'accounts', 'bacqueville', 'de', 'la',
'potherie', 'le', 'roy', 'alone', 'gives', 'him', 'a', 'name',
'he', 'steps', 'into', 'history']
```

Of these, the following are our stopwords:

```
stopwordlist = ['of', 's', 'can', 'only', 'be', 'without',
'too', 'much', 'from', 'and', 'de', 'alone', 'him', 'a',
'name', 'he', 'into']
```

Now for each word in our word list, we want to return the placeholder if the word is one of the stopwords, otherwise we want to return the word itself. We can translate this directly into a lambda expression:

```
lambda x: placeholder if x in stopwords else x
```

Then we can build this anonymous function into one which replaces all of the stopwords in a list with the placeholder. Copy the following and put it into *dh.py*.

```
# Given a list of words, replace any that are
# stop words with a placeholder.
def replaceStopwords(wordlist, stopwords, placeholder = '#'):
    replacefunction = (lambda x: placeholder if x in stopwords else x)
    return map(replacefunction, wordlist)
```

If you try it in a shell with the two lists given above, you should get the following:

```
print replaceStopwords(wordlist, stopwordlist)
-> ['dictionary', '#', 'canadian', 'biography', 'agariata',
'agoriata', 'mohawk', 'chief', 'fl', '1666', 'agariata', '#',
'career', '#', '#', '#', 'reconstructed', '#', '#', '#',
'certainty', '#', 'fragmentary', '#', 'conflicting',
'accounts', 'bacqueville', '#', 'la', 'potherie', 'le', 'roy',
'#', 'gives', '#', '#', '#', '#', 'steps', '#', 'history']
```

Note that the `replaceStopwords` function expects three parameters, but we only gave it two. The placeholder = '#' syntax in the function definition provides a default value for the third parameter if we don't include it in our function call. How would you override the default parameter value to make the placeholder equal to '% %'? Test your answer in a Python shell.

## Zip and tuples

We know how to replace stopwords with a placeholder, so the next step is to associate each word in the file with the position(s) in which it occurs. Once again we will use a functional programming technique, this time making use of the Python `zip` function. Let's build up to it slowly.

Suppose we have the list of words with placeholders. We can use `len` to find out how many elements there are in the list. Python also has a `range` function which can be used to generate numeric sequences. Study the examples below.

```
replacedlist = ['dictionary', '#', 'canadian', 'biography',
'agariata', 'agoriata', 'mohawk', 'chief', 'fl', '1666',
'agariata', '#', 'career', '#', '#', '#', 'reconstructed',
'#', '#', '#', 'certainty', '#', 'fragmentary', '#',
'conflicting', 'accounts', 'bacqueville', '#', 'la',
'potherie', 'le', 'roy', '#', 'gives', '#', '#', '#', '#',
'steps', '#', 'history']
```

```
print len(replacedlist)
```

```
-> 41
```

```
print range(0, len(replacedlist))
```

```
-> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
```

`Zip` allows us to create a list of pairs from a pair of lists, like so.

```
print zip(replacedlist, range(0, len(replacedlist)))
```

```
-> [('dictionary', 0), ('#', 1), ('canadian', 2), ('biography', 3),
('agariata', 4), ('agoriata', 5), ('mohawk', 6), ('chief', 7),
('fl', 8), ('1666', 9), ('agariata', 10), ('#', 11), ('career', 12),
('#', 13), ('#', 14), ('#', 15), ('reconstructed', 16), ('#', 17),
('#', 18), ('#', 19), ('certainty', 20), ('#', 21),
('fragmentary', 22), ('#', 23), ('conflicting', 24),
('accounts', 25), ('bacqueville', 26), ('#', 27), ('la', 28),
('potherie', 29), ('le', 30), ('roy', 31), ('#', 32), ('gives', 33),
('#', 34), ('#', 35), ('#', 36), ('#', 37), ('steps', 38),
('#', 39), ('history', 40)]
```

Note that each of the elements of this list is a pair of items in parentheses, rather than square brackets. The parentheses tell us that these are Python *tuples* rather than lists. A tuple is like a list in that it is a sequence. Unlike a list, however, the elements of a tuple cannot be modified. For present purposes we can gloss over the difference.

## Putting it all together

We now have enough background to create a complete index of our document collection. Copy the following code and save it as *index-document-collection.py*.

```
# index-document-collection.py

import dh
import os

# load a list of filenames to process

collectiondir = 'iroquois'
filelist = dh.getFileNames(collectiondir)

# For each file in the list, index the words in it
completeindex = []
for i in filelist:

    # convert file to list of words
    fname = collectiondir + '/' + i
    ftext = dh.localWebPageToText(fname)
    flist = dh.stripNonAlphaNum(ftext)

    # replace stopwords with placeholder
    flist = dh.replaceStopwords(flist, dh.stopwords)

    # create a list of (word, filename, offset) tuples
    ftuplelist = zip(flist, (fname,)*len(flist), range(0, len(flist)))

    # add tuples to complete index list
    completeindex += ftuplelist

# remove stop words from complete index list
completeindex = [x for x in completeindex if x[0] != '#']

# print the first few members of complete index list
print completeindex[0:16]
```

When you execute the program, it should print the following to the output pane

```
[('', 'iroquois/dcb-34125.html', 0), ('dictionary', 'iroquois/dcb-34125.html', 1),
('canadian', 'iroquois/dcb-34125.html', 3), ('biography', 'iroquois/dcb-34125.html',
4),
('agariata', 'iroquois/dcb-34125.html', 5), ('agoriata', 'iroquois/dcb-34125.html', 6),
('mohawk', 'iroquois/dcb-34125.html', 7), ('chief', 'iroquois/dcb-34125.html', 8),
('fl', 'iroquois/dcb-34125.html', 9), ('1666', 'iroquois/dcb-34125.html', 10),
('agariata', 'iroquois/dcb-34125.html', 11), ('career', 'iroquois/dcb-34125.html', 13),
('reconstructed', 'iroquois/dcb-34125.html', 17), ('certainty', 'iroquois/dcb-
34125.html', 21),
('fragmentary', 'iroquois/dcb-34125.html', 23), ('conflicting', 'iroquois/dcb-
34125.html', 25)]
```

Note that we've used *zip* to create triples that include the file name as well as the position of a given word. In the next lesson we'll rework this list so that the information is organized by word and then store our index in a relational database.

## Suggested Readings

Lutz, *Learning Python*

Ch. 17: Advanced Function Topics

## *Discussion of The Programming Historian, 1st ed.*

### Do you need to learn how to program?

16 Apr 2008. Nick Poyntz writes:

The section on techniques that don't involve programming is helpful, but might be pitched at an even simpler level. People coming to this fresh, perhaps only using EndNote and IE, or maybe not even EndNote, will need to know that Firefox is an alternative browser that allows you to do other helpful things, and will need to have it explained in more detail what Zotero is other than just a citation manager.

14 Apr 2008. Gavin Robinson writes:

As you say, pretty much all historians have already gone digital to a certain extent. You could also point out that even where sources are not online and are only available in archives, some archives allow digital photography and increasing numbers of historians are taking advantage of this.

"If you don't program, your research process will always be at the mercy of those who do." - And what's worse, you'll be at the mercy of people who do it badly. A big incentive for learning to program is that some of the most useful sites for historians are badly designed, Google unfriendly, and don't have good search features built in.

## Getting started

### Stuff that needs more explanation

16 Apr 2008. Nick Poyntz writes:

On the HTML page, it is implicit but not explicit that this is the basic language that web pages are written in - again for beginners it might just be worth making that 100% clear.

7 Apr 2008. We should probably include a bit more discussion about the similarities and differences between terminals, shells and interpreters.

## Macintosh

29 Mar 2008. We haven't had a chance yet to run through all of the exercises on the Mac platform. David Jones writes:

I've made it through the getting started page, with eventual success, though some Mac related glitches.

1. The developer's extension link didn't work consistently, but I got it eventually.
2. For html, the tools->extension developer does have an html editor (like the ones for python and java) which works fine: I can enter code, save it as an html file, and then open it with firefox.

## Firefox Plugins

18 Feb 2008. Ben Brumfield writes:

You should really consider switching your first plugin recommendation from the Web Developer toolbar to FireBug. It incorporates the best of both the Web Developer CSS tools and the Venkman JavaScript debugger, and it seems there's a lot more ongoing support and development on it. A better recommendation: both developers who were expert users of Venkman and designers who were expert users of Web Developer have recommended Firebug enthusiastically to me.

14 Apr 2008. Gavin Robinson writes:

On Firebug I'd say it's not really for beginners. It's immensely powerful but might also be intimidatingly complicated to people with no experience of coding. Web Developer is good to start with, then people can move onto Firebug when they need something extra.

One of us (WJT) found that Firebug crashed one of his Windows systems, so was hesitant to recommend it until that was figured out. Pending.

22 Jun 2008. Sharon Howard writes:

I upgraded to Firefox 3 on my backup laptop the other day as a test run, and it took exception to the Extension Developer's Extension for not providing 'secure updates'. Presumably as a result of this, the author has now put the extension on the official Mozilla site and this version seems to install OK:

<https://addons.mozilla.org/en-US/firefox/addon/7434>

12 Aug 2008. Adam Crymble writes:

A lot of extensions don't meet the security standards of Firefox 3 and therefore it refuse to allow the user to enable them. The specific problem for most of these add-ons is that the program "does not provide secure updates."

There is a free and easy way to override this called "Nightly Tester Tools" which can be downloaded freely from <https://addons.mozilla.org/en-US/firefox/addon/6543> which, once installed should help solve the problem.

However, this gives your readers the opportunity to install programs Firefox specifically tells them not to, which obviously can lead to problems. So you'll have to be careful advising this.

The Extensions Developer's Extension is the one I had problems with.

## Revision Control Systems

19 Feb, 2, 20 Mar 2008. Ben Brumfield writes:

At the end of the page, you mention making backups. I'm not sure this goes quite far enough. Based on my own experience as a developer and two years of work in a university helpdesk, I'm convinced that researchers need to be comfortable with source control tools.

Any website can expound on the joys of SCM for programming, but I think there are some side-effects worth mentioning.

1. Think of the non-programming use cases: I can see the differences between the four revisions to my THATcamp application. And since the differences live in the SC repository, my letter is a text file -- no worries that the eventual recipient will turn on "track changes" and see the embarrassing typos in the first rev.
2. Checking everything into an SCM gives you portability. The Zotero FAQ you link to suggests "an external storage device (such as a removable/USB drive, portable hard drive, or iPod) and use Zotero within that". Why fool with key fobs you might run through the laundry? I can get at my source code from our laptop, our desktop, AND from the server I'm deploying it on. That's a hard task for a USB drive.
3. Any centrally managed SCM gives you a much higher chance of having your worked backed up correctly. If the SCM is on a university system, it's likely that system is backed up more frequently than your development machine -- especially if the system is shared campus-wide, rather than a departmental box. If it's an external host, the same is true. Better yet, even if you don't trust the box the repository is on, the portability I mentioned in 2 gives you a chance to script something that checks out and zips up your whole tree every night, somewhere you do trust.
4. SCR often allows for easy distribution. Provided your repository is on a machine accessible to the Internet, it's really pretty simple to allow people to browse or download items in SVN via the svn client or the web interfaces.

Some references:

- \* Vitamin has just posted an introduction to Subversion and SCR, aimed at another non-developer group: designers. It does a good job covering client installation on Mac and Windows.
- \* For a technical -- but possibly skeptical -- readership, I recommend the relevant chapter from Cal Henderson's Building Scalable Websites.

One of us (WJT) is just starting to use Subversion to synchronize all of his own files across multiple computers and operating systems. Check *Digital History Hacks* for more information.

### **Lutz, *Learning Python***

29 Mar 2008. WJT used the 2nd edition of this book a lot, but hasn't used the 3rd ed much so wasn't too sure about it. Adam Crymble writes:

When I first went through the Programming Historian, I did use Lutz's "Learning Python" and found it quite helpful.

### **Working with files and web pages**

16 Apr 2008. Nick Poyntz writes:

The advice on close reading is one of the most useful central tenets of the book - so much so that I wonder if it is worth repeating in the section on "Why program". It reassured me that actually, I probably do have the skills to go through this tutorial and come out of it at the other end with

some skills.

8 Aug 2008. Guilherme Ferreira reported a trailing blank (%20) appended after the extension of the saved file when using file-output and file-input. We haven't recreated the bug, but it may have something to do with a difference between Win and Linux. Please let us know if you have a similar problem. It can probably be easily fixed in code by using Python `rstrip`.

## From HTML to a list of words

### Stuff that needs more explanation

16 Apr 2008. Nick Poyntz writes:

By the time I got to the section on moving from HTML to a list of words, it was getting rather dense - I had to read through a number of times to make sure I'd understood it. This probably says more about me than the text, but I wonder if there is some way of trying to break it up.

It's not just you, Nick :) A number of people--including AM--think that the new material is too dense here. We've talked it over and agree that we need to provide more explanation about why we choose to save some code in separate Python programs and some in the `dh.py` module. We also need to include a downloadable copy of the `dh.py` file at the end of each lesson, so readers can check to make sure that they haven't accidentally introduced any mistakes or forgotten to add routines that will lead to strange bugs in future lessons. Pending.

18 Mar 2008. David Jones writes:

1. Operators / operations seem to include everything from `print` to `append`, `split`, etc. What exactly are these -- functions built into Python itself? Is `char` an operation? A type or an object? Why are some built in, where as others require importing `urllib2`?
2. `[ ]` vs. `( )`: clearly have different meanings, like with your explanation of `=` vs. `==`
3. In `html-to-list-1`, what is `text`? Is it a variable you introduce? Do you have to define variables in python before you use them? For instance, when `string-to-list` says `charlist=[]`, is that defining `charlist` as a list, or is that clearing whatever prior values `charlist` (not requiring definition) might have had? `Text` seems to be both the product of `dh / striptags(html)`, and then something that operates on `split`? I guess I can follow, but don't quite understand python syntax.

14 Apr 2008. Gavin Robinson writes:

Maybe you could explain more explicitly that an integer is a whole number that you can do mathematical operations with but can't store fractions. It might not be obvious to people with no programming/mathematical background, and it's such an important and basic concept that I wouldn't rely on readers to follow the link you've provided. There's probably no harm in spelling it out.

## Computing frequencies

### Possible Glitch

19 Mar 2008. David Jones writes:

The program stopped working the first time I tried the delete stopword bit, but when I redid everything it did work -- don't know what was up there.

## Wrapping output in HTML

### Possible Glitch

19 Mar 2008. David Jones writes:

I got an error on the bit about having Komodo automatically open the file in firefox. As best as I can tell, the browser can't find the file, so I wonder if I'm saving my python scripts and their output in a bad place. Or maybe it's something else.

Happily it looks like I can proceed regardless of this error, and just open up the browser on my own.

```
0:36: execution error: An error of type -2110 has occurred. (-2110)
Traceback (most recent call last):
  File "write-html-2.py", line 15, in
    webbrowser.open_new_tab('helloworld.html')
  File
"/System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/webbrowser.py",
line 68, in open_new_tab
  File
"/System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/webbrowser.py",
line 60, in open
  File
"/System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/webbrowser.py",
line 539, in open
  File "/System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/plat-
mac/ic.py", line 235, in launchurl
  File "/System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/plat-
mac/ic.py", line 202, in launchurl
MacOS.Error: (-673, 'no URL found')
```

18 May 2008. Ted Lawless writes:

When I was working on the Sending Output to HTML section, I received the error message that is listed above. On my Mac I was able to correct the problem by adding the filepath to the filename variable. Here is an example from my machine:

```
filename = 'file:///Users/-username/Desktop/python-digitalhistorian/' + filename
open_new_tab(filename)
```

## Keyword in context (KWIC)

14 Apr 2008. Gavin Robinson writes:

I wonder if the pretty printing is really necessary here. Padding strings with spaces seems to be a bit tricky and perhaps not very useful in the long term. Would it be better to present the KWIC in an HTML table instead? That might tie in better with your aim of working in the browser as much as possible, and would also be better semantics than preformatted text.

## Tag clouds

20 Mar 2008. In an earlier draft we forgot to include the code for reSortFreqDictAlpha. David Jones pointed out the error, so we included the routine but haven't had a chance to explain it yet. So for now, "You are not meant to understand this". Sorry!

## *Peer Reviewers*

Like all open works, this project continually benefits from the review of our peers. We would like to thank the following people for questions, comments, corrections, and suggestions for improvement:

- [Boggs, Jeremy](#)
- [Brumfield, Ben](#)
- [Cohen, Dan](#)
- [Crombez, Thomas](#)
- [Dalziel, Karin](#)
- [Day, Shawn](#)
- [Elliott, Devon](#)
- Ferreira, Guilherme
- Fortin, Marcel
- [Garrigus, John D.](#)
- [Goodger, David](#)
- Gracy, Benjamin
- [Heppler, Jason](#)
- [Howard, Sharon](#)
- [Jones, David](#)
- Jovanović, Neven
- [Krishnan, Shekhar](#)
- Lawless, Ted
- [Lester, Dave](#)
- [Poyntz, Nick](#)
- [Nicolas Quiroga](#)
- [Ramsay, Stephen](#)
- [Robinson, Gavin](#)
- [Sadler, Bess](#)
- [Sinclair, Stéfan](#)