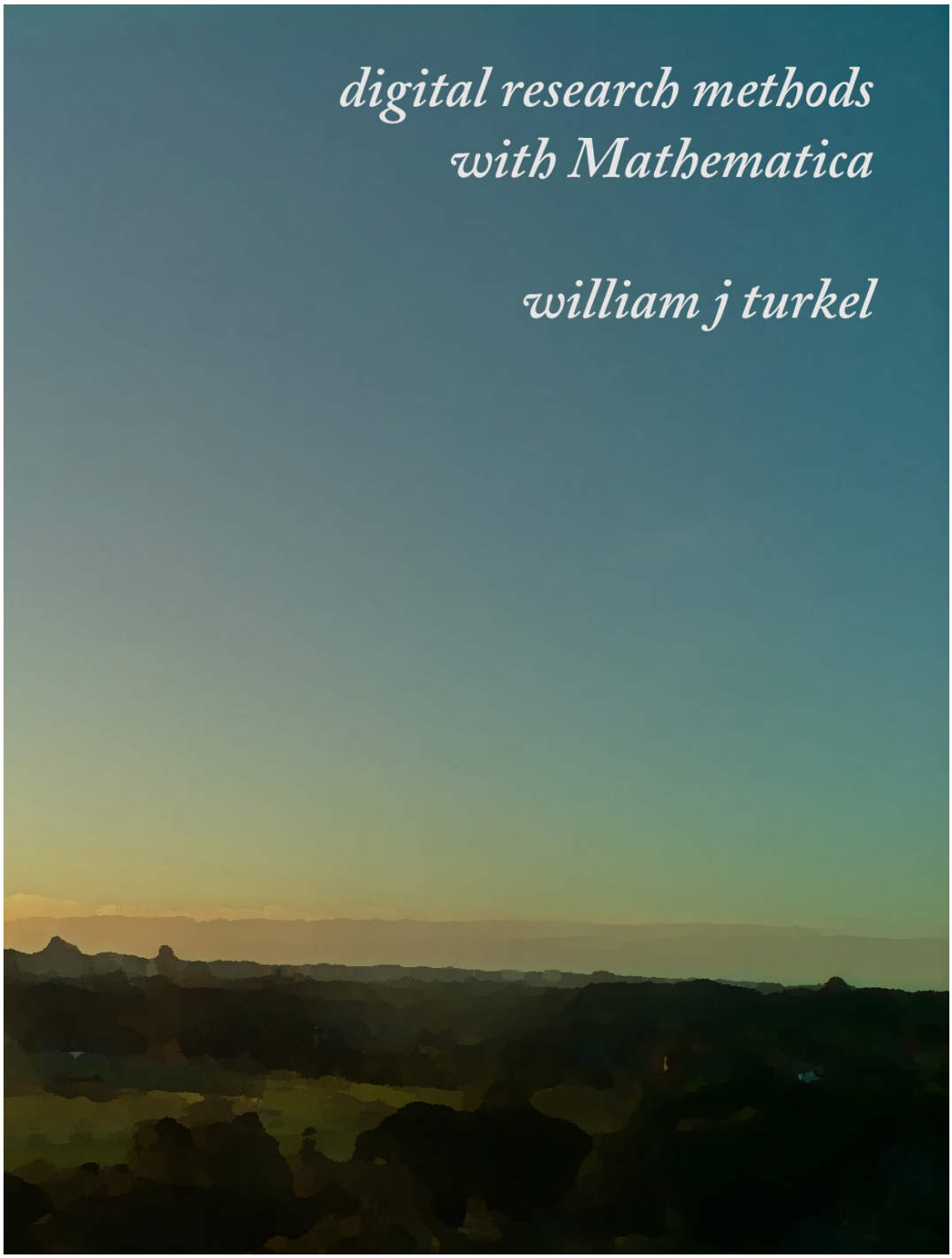
digital research methods
with Mathematica
william j turkel

Digital Research Methods with *Mathematica*®

William J. Turkel
University of Western Ontario
william.j.turkel@gmail.com

2015

This is a *Mathematica* notebook

If you have never used one before, you can expand and collapse sections by double clicking on the cell brackets to the right...

The cell beneath this one contains code. You can *evaluate* it by selecting its cell bracket and pressing SHIFT↵

```
2 + 2
```

about

About This Book

Version

This is version 1.0 (Summer 2015). There are six chapters that are more-or-less finished, and code snippets for about six more chapters that I would like to write. The sections that are unfinished are marked *(Under Development)*.

The latest version of this book is freely available at
http://williamjturkel.net/digital-research-methods-with-mathematica/

This book is compatible with *Mathematica* version 10.2 and later. Most of the code will work with versions later than 9.0.

Slides

You can find slides to complement the book at
http://williamjturkel.net/teaching/history-2816a-introduction-to-digital-history-fall-2015/
http://williamjturkel.net/teaching/history-9877a-digital-research-methods-fall-2015/

Feedback

Feedback is always welcome. You can email me at *william.j.turkel@gmail.com*

Licenses

Acknowledgements

Jack Bunce, my RA in the summer of 2015, provided feedback on chapters 01-06, put in a number of hyperlinks to the documentation and suggested some of the problems and exercises. Arno Bosse (University of Oxford) worked through chapters 01-02 and provided very extensive notes. Chris Roney (Wolfram Research) was an enthusiastic supporter of the project. Mathematica Stack Exchange is the first place I looked for answers to any questions that weren't covered by the documentation.

Cover Image

The cover image is based on a photograph of Sydney taken by the Climate Change Research Centre in August 2015. It has a CC-BY license. https://www.flickr.com/photos/ccrc_weather/20372731668/

Table of Contents

You can click on a heading to jump directly to that chapter.

intro

Introduction

This is a book about doing research with digital sources. It teaches you how to find, harvest, manage, excerpt, cluster and analyze digital materials throughout the research process, from initial exploratory forays through the production of an article, chapter or monograph which is ready to submit for publication. For more than a decade, 'doing research' has mostly meant doing research online, and yet there are few textbooks that bridge the gap between the basic stuff—using search engines, evaluating the quality of sources, verifying information, keeping track of citations with bibliographic software—and the kind of computational techniques that researchers need to thrive in a networked environment with superabundant source materials. For the key point about digital sources is that they can be read and processed by machines, in vast numbers and very quickly, often autonomously. The most scarce resource now is human care and attention. You want to be able to focus on reading, thinking and writing, and let computers do what they do best.

Why should you learn digital research methods? Aren't traditional research methods good enough?

Here are three scenarios.

1. You gain temporary access to a large collection of non-digital sources that are important for your research. These might be in an archive or private collection; they might be pamphlets or books or boxes of file folders. Given the ubiquity of camera phones, most people would now choose to photograph the pages if possible, and researchers who do extensive archival work often return home with thousands or tens of thousands of digital photographs of various documents. You could spend the next few years going through the pictures one at a time and typing notes into a word processor. Or you could write a small script to convert each image into readable text and drop the whole batch into a custom search engine. In less than an hour you could be searching for words and phrases anywhere in your primary sources.
2. You discover a collection of hundreds or thousands of online texts that are directly related to your research. You could look through the list of titles in your web browser and click on the links one at a time, scanning each to see if it is relevant. Even if you cut-and-paste notes from the sources to a word processor, it will still take you at least a few months to go through the collection. Or you could write a small script to download all of the sources to your own machine and run a clustering program on them. This sorts the texts into batches of closely related documents, then subdivides those by topic. In less than an hour, you would be able to visualize the contents of the whole collection and focus in on the topics that are of immediate interest to you.
3. You've been working with the written corpus of a historically significant figure. You have the books and essays that he or she wrote, their diary entries and their correspondence with a large number of other individuals. How do you make sense of a lifetime of writing? Can you chart important changes in someone's conceptual world? Spot the emergence of new ideas in the discourse of a community? Map the ever-changing social relations between a network of correspondents?

These examples are just the tip of the iceberg, however. *Digital Research Methods* introduces a wide variety of other powerful techniques: automatically extracting all of the images that appear in series of page images (say the run of a newspaper or journal) and classifying them into photos, drawings, charts, and so on; automatically identifying the people, places, institutions, dates, and other entities mentioned in texts; mapping and visualizing huge data sets; linking records to computable data; and many others. Computation won't magically do your research for you, but it will make you *much* more efficient. You can focus on close reading, interpretation and writing, and use machines to help you find, summarize, organize, analyze and visualize sources.

Digital Research Methods is suitable for self-study or for a one-term undergraduate or graduate course. Since it is intended for as wide an audience as possible, I've tried to keep mathematical prerequisites to a minimum. Not to say that there isn't some math in the book, just that it is included in sections that are designed to encourage further or deeper exploration, and can be skimmed on a first reading. Likewise, I don't assume that you already know how to program, although I do hope that you will learn some new techniques, whatever your level of previous programming experience.

I use this book to teach both undergraduate and graduate courses in the humanities and social sciences. For the undergraduate course, I schedule two blocks per week of two hours each. For the first hour of each block, I work through basic examples carefully, explaining how they work, answering any questions the students have, and asking them questions that I think will deepen their interest or understanding. I have been teaching programming for more than thirty years now, and I think that one of the best ways to learn how to program is to do it collaboratively with other people who have more experience programming. For the second hour of each block, the students have a chance to work on programming problems and exercises themselves while I walk around the room and answer questions, make suggestions, and so on. In the graduate course, I expect the students to try to work through some basic examples on their own. In class we cover the basics more quickly then discuss generalizations and applications. The students then try to apply the techniques to their

own research materials and write up their results in the form of reflective blog posts. For both classes, I have prepared a set of slides that complement this book. They can be found online at http://williamjturkel.net/teaching/

For self-study or for students in more technical disciplines, this book can be combined with a guidebook to using *Mathematica* for doing math (such as Torrence & Torrence 2009, which is dated but still good) and/or a book on *Mathematica* programming (such as Wellin 2013). Some of the undocumented code, project ideas and exercises require more substantial background in math or computer science. If you can't make sense of the code without an explanation, you're probably not ready to tackle that problem or technique yet.

Mathematica has a number of features which make it particularly useful for doing digital research. One of the main advantages is that the notebook model allows you to mix prose, data, executable code, visualizations, simulations, interface elements, hyperlinks and other elements. The computer scientist Donald Knuth called this 'literate programming': "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do" (Knuth 1984). This is crucial if you are collaborating with other people, which is something that most digital research projects require. Even if you're working by yourself, however, you'll find that your own code is much easier to understand a few months later if it is surrounded with some text that explains what the heck you were thinking when you wrote it. Notebooks are enhanced by the possibility of embedding code with interfaces that can be manipulated interactively or dynamically updated.

Mathematica has a vast number of very powerful commands, which makes it possible for an individual programmer to write very short programs to perform sophisticated tasks. The problem is usually finding the command that you need for a given problem. Whenever possible, I have provided links to the help system, and you should get in the habit of having a documentation window open on your screen and reading it continuously while you work. You can also enter commands in plain English (see the "What You Need to Know to Get Started" section below). Since every technical domain is described with a body of mathematics, another advantage to using *Mathematica* is that it already knows how to operate with the objects of these domains (e.g., matrices). And you have access to a huge amount of computable data via Wolfram Alpha, Wikipedia, and other online sources.

Finally a word about the examples used in the book, which are all taken from the history of natural history (and most of which focus on Darwin). The techniques presented here can be used for any kind of digital research and you don't need to be interested in Darwin to use them. I decided to focus on a subject that I am quite familiar with: I took graduate courses on the history of natural history, read many works by Darwin and about him and his milieu, and wrote a book of my own that overlaps with the subject (Turkel 2013). At the same time, this is a topic that is large enough that no one is familiar with even a significant portion of it, and thus one that amply rewards the use of digital research methods.

References and Further Reading

- Wolfram Language Principles
- Knuth, Donald E. "Literate Programming," 1984. http://www.literateprogramming.com/knuthweb.pdf
- Torrence, Bruce F. & Eve A. Torrence. 2009. *The Student's Introduction to Mathematica: A Handbook for Precalculus, Calculus, and Linear Algebra*. Cambridge: Cambridge University Press.
- Turkel, William J. 2013. *Spark from the Deep: How Shocking Experiments with Strongly Electric Fish Powered Scientific Discovery*. Baltimore: Johns Hopkins University Press.
- Wellin, Paul. 2013. *Programming with Mathematica: An Introduction*. Cambridge: Cambridge University Press.

- Wolfram, Stephen. 2010. “Making the World’s Data Computable.”

know

What You Need to Know to Get Started

Make sure you know your way around the *Mathematica* interface

Before you go too far, make sure you know how to do basic stuff like execute commands, collapse and expand notebook sections, and so on. There is a lot to learn, but don’t worry about getting it all at once. Practice a little bit each day.

Here are some useful links to introductory screencasts

Quick tour of *Mathematica*
First 10 Minutes with *Mathematica*
Hands-on Start to *Mathematica*

There are plenty more screencasts and videos available at the Wolfram Research site. There is also a collection of tutorials.

It is a good idea to learn the keystrokes for the things that you do all the time so you feel more comfortable using *Mathematica*.

Keyboard shortcut listing
Notebook shortcuts

You can add hyperlinks to your notebook by highlighting some text and choosing *Insert→Hyperlink* from the menu. When you click on a hyperlink it opens in your default browser.

The *Mathematica* documentation is amazing

If you come across a command that you haven’t encountered before, you can get help by typing a question mark and the name of the command and then evaluating it (i.e., pressing SHIFT↵)

```
? Import
```

If you need more help, you can click on the little double arrow (≫) to open the help browser on your own machine.

You can also open the help browser at any time by pressing the F1 key.

The documentation is also available online at

Wolfram *Mathematica* Documentation Center

Here are some screencasts about *Mathematica*’s help system

How to Search for Help
How to Use the Virtual Book
How to Use the Suggestions Bar
How to Use the Input Assistant
How to Find Available Options
How to Find Information about Functions

There is also a very useful list of Common “How Tos” at

How To Topics

How to input unusual characters

Mathematica includes a complete system for typesetting mathematical expressions and other

technical prose. If you need to input an unusual character, the first thing you should try is using one of the Palettes.

How to Use Palettes

When you hover over an entry in the palette, a tooltip will show you the characters you have to type to enter the same thing from the keyboard. If you need a particular character often, it is usually worth learning the keystrokes required to create it.

Commenting out code

If you don't want code to execute, you can select it and then use CMD/ to add or remove comment markers

```
(*Print["By default this is commented out"]*)
```

When all else fails try asking in plain English

Mathematica has direct access to the Wolfram Alpha computational knowledge engine. You can type a single equals sign at the beginning of an input line to use free-form linguistics to generate *Mathematica* output.

If you use a double equals sign at the beginning of an input line you can generate full Wolfram Alpha output in your notebook.

Voyage of the Beagle

As we will see, both of these options can be used to incorporate computable data into *Mathematica* programs.

If you already have some programming experience...

... start with the following links.

Fast Introduction for Programmers
Wolfram Language Principles and Concepts
Notes for Programming Language Experts
Mathematica Stack Exchange

Leonid Shifrin also has an open access (BY-NC-SA) textbook

Mathematica Programming: An Advanced Introduction (2008/9)

ch01

Chapter 01: Analyzing Text

Overview

We begin with text because almost every research project involves extensive reading and writing, even if your ultimate product will take a non-textual form. In some cases you will need to do a close reading of a text; in others you may be able to learn what you need to know simply by skimming or scanning it. Either way, human reading can be complemented by various kinds of machine reading. The simplest of these is a study of the frequency of words appearing in the text.

Basic Examples

A sample text: Darwin's *On the Origin of Species*

Digital sources come in many forms, both human-readible and machine-readible. One of the easiest to process is the raw text file. (Raw text is also known as ASCII text, for the method used in encoding the characters. More about this later). *Mathematica* includes a number of text files to experiment with. We will begin with Darwin's *On the Origin of Species* (1859). The following command retrieves the text and *assigns* it to be the value of the symbol *origin*. When we put a semicolon at the end of a line like this, we are telling *Mathematica* that we want it to supress any output that it might otherwise return to our notebook. In this case, if we were to omit the semicolon, the entire text of *Origin* would be pasted in.

All reserved keywords in *Mathematica* begin with a capital letter, so it is a good idea to name your own symbols with a lowercase letter (as we do with *origin*).

You need to evaluate each cell of code when you get to it. Use your mouse to select the bracket shaped cursor at the far right of the line below, hold down SHIFT and press ↵. The word 'origin' should turn from blue to black.

```
origin = ExampleData[{"Text", "OriginOfSpecies"}];
```

In *Mathematica*, we can use the **Short** command to look at something which may be quite large. In this case, *origin* is a whole book, so we just want to see a bit of it. Evaluate the cell below by selecting its cursor, holding down SHIFT and pressing ↵.

```
Short[origin]
```

That is pretty short. We can see a bit more of the text by giving the **Short** command an option which tells it about how many lines we want to see. Notice that the elipses (...) indicate the portion in the middle which has been removed.

```
Short[origin, 10]
```

The **Head** command tells us what something is. Our text is currently a *string*, an ordered sequence of characters.

```
Head[origin]
```

We can start our analysis of the text simply by counting the total number of words in it. Note that this is a count of word *tokens* rather than word *types*. In the example below, note how each instance of the word 'I' (that is, each token of the word) is counted separately.

```
WordCount["I came, I saw, I conquered."]
```

```
Head["I came, I saw, I conquered."]
```

Because strings may contain punctuation marks and whitespace (blank spaces, tabs, and so on), we need to enclose them in double quotation marks when we type them in so that *Mathematica* knows where they begin and end. So how many words are there in *Origin*? We can find out with the following command. We don't have to enclose the symbol *origin* in quotation marks, because *Mathematica* already knows it is a string.

```
WordCount[origin]
```

When you learn a new command, like **Short**, **Head** or **WordCount**, it is a good idea to spend a few moments looking at the documentation page for each.

Working with a piece of the text

Our whole text is almost 150 thousand words long. Let's take a smaller sample of it to explore. We can use the **StringTake** command to select a certain number of characters and assign them to a new symbol. These characters might be letters, numbers, punctuation or whitespace. When we take the first five thousand characters of our text, we are not sure where the cutoff point will be. We probably will end up splitting a word in two. Once again, the value returned will be quite long so we supress the output with a semicolon, then use **Short** to look at part of it.

```
origin5K = StringTake[origin, 5000];
```

```
Short[origin5K, 10]
```

This new text is also a string. It contains far fewer word tokens. We can also count the number of characters in the string by using the **StringLength** command.

```
Head[origin5K]
```

```
WordCount[origin5K]
```

```
StringLength[origin5K]
```

We can segment this text into separate sentences with the **TextSentences** command. We assign the output of the command to a new symbol. This, however, is not a string. It is a *list* of strings. In *Mathematica*, lists are enclosed in curly braces { } and list elements are separated by commas. The ≪*n*≫ indicates that *n* elements from the middle are not shown. In this case, each element is a separate sentence.

```
origin5KSentences = TextSentences[origin5K];
```

```
Short[origin5KSentences, 20]
```

```
Head[origin5KSentences]
```

We can count the number of elements in a list with the **Length** command.

```
Length[origin5KSentences]
```

The chapter title, 'INTRODUCTION', counts as a sentence here because it ends in a period. The final sentence, which has been interrupted by our use of the **StringTake** command, is also counted as a sentence. *Mathematica* has correctly recognized that the periods in 'H.M.S.' do not mark sentence boundaries. The method is not perfect, however, and occasionally it will be confused by punctuation.

Counting words per sentence

We can see that the first sentence is pretty long. In order to count the length of each sentence, we can make use of a very powerful operation, known as **Map**. There is more information about **Map** in the 'Programming with *Mathematica*' section below. What the following command says is 'apply the **WordCount** function to each element of the *origin5KSentences* list and return the results as a list.' For convenience, we will again assign this result to a symbol.

```
origin5KSentenceLengths = Map[WordCount, origin5KSentences]
```

```
Head[origin5KSentenceLengths]
```

```
Length[origin5KSentenceLengths]
```

We can visualize this list with the **ListPlot** command. The *Filling→Axis* option draws little lines from each point down to the x axis. The *AxesLabel* option provides labels for the *x* and *y* axes of the figure. In an input cell you can make the → character by typing in a dash followed by a greater than sign -> then pressing the space bar. Or you can type ESC->ESC.

```
ListPlot[origin5KSentenceLengths, Filling → Axis,
 AxesLabel → {"Sentence Number", "Length in Words"}]
```

Note that what we are counting here are word tokens. Even in a short text like this one, some word types (such as 'the') will be represented by many tokens. Others (such as 'absolutely') only appear once.

Parts and spans of lists

We might be interested in measuring the average sentence length for this part of the text, but if we include the chapter title it will skew our results, since it is very much shorter than the other sentences. We don't want to include the last sentence either, since it is only partial. We need a way of referring to and selecting parts of a list. The **Part** command does this.

```
Part[origin5KSentences, 1]
```

```
Part[origin5KSentences, 2]
```

The tenth sentence is particularly short. When we look at it, we can see that *Mathematica* accidentally split one sentence into two. We could repair this by joining the two sentence pieces back together but we won't bother right now.

```
Part[origin5KSentences, 10]
```

The 26th sentence seems particularly long. We can check to see that is indeed the case.

```
Part[origin5KSentences, 26]
```

Here is the last, interrupted, sentence...

```
Part[origin5KSentences, 27]
```

Another way of representing a part of a list is with a pair of square brackets. This is the same as the previous command.

```
origin5KSentences[[27]]
```

In *Mathematica*, these can also be represented with special characters that are created using esc[[esc and esc]]esc . This notation is equivalent to the two previous commands.

```
origin5KSentences⟦27⟧
```

Mathematica also has a special notation for counting from the end of list. You can use negative numbers like this:

```
origin5KSentences⟦-1⟧
```

```
origin5KSentences⟦-2⟧
```

We can refer to a range of elements by using a command called **Span**, which is represented by a pair of semicolons. Study the examples below. Here is **Part**, which you have already seen.

```
Part[{a, b, c, d}, 1]
```

Another way of writing **Part**

```
{a, b, c, d}[[1]]
```

Here is an example of **Span**

```
{a, b, c, d}[[1 ;; 2]]
```

This can also be written as

```
Part[{a, b, c, d}, Span[1, 2]]
```

Here is another example of **Span**, written with the more terse notation

```
{a, b, c, d}[[2 ;; 3]]
```

Using the special characters for the pairs of square brackets

```
{a, b, c, d}⟦2 ;; 3⟧
```

I have emphasized the different ways write **Part** and **Span** because you usually need to know the name of a command to look it up in the documentation, but almost all of the code examples that you see will use the abbreviated syntax.

Average sentence length

Now that we know how to refer to and select pieces of lists, we can return to the problem at hand. We are interested in averaging the lengths of all of the sentences in origin5KSentences except the first and last, so the span that we are interested in is *2;;-2*. We have already assigned the list of sentence lengths to *origin5KSentenceLengths*, so we can use the **Mean** command to find the average that we are interested in.

```
Mean[origin5KSentenceLengths⟦2 ;; -2⟧]
```

Unless directed otherwise, *Mathematica* does not convert rational numbers (fractions) to their decimal equivalents. In this case we want the decimal number, so we can convert the output of the **Mean** expression using the **N** command. Using two slashes like this, followed by a command, tells *Mathematica* to use the output of one expression as the input of the following one. There is more information about this in the 'Programming with *Mathematica*' section below.

```
Mean[origin5KSentenceLengths⟦2 ;; 26⟧] // N
```

The average sentence length for the first part of the introduction to *Origin* is 33.84 word tokens long (setting aside the problem of accidentally splitting the tenth sentence into two parts). In the section below on 'Generalizing the Examples' we look at sentence lengths for the whole book.

Determining word frequencies

We can learn a lot about a text by seeing which words occur more or less frequently. Some words, such as 'the', 'of', 'or' and 'she' occur frequently in most texts. They are crucial to the meaning of the text but they don't help to distinguish it from other texts. Other words, like 'geological', may occur relatively infrequently in normal use but very frequently in particular texts (like *Origin*).

We can use the **WordCounts** command to count the tokens in a string. The *IgnoreCase→True* option tells the following command to ignore capitalization, counting 'When' and 'when' as two different tokens of the same word type. This command returns its results in the form of an *association*.

```
origin5KWordFreqs = WordCounts[origin5K, IgnoreCase → True];
```

```
Head[origin5KWordFreqs]
```

In an association, *keys* are paired with *values*. Here the keys are word types that appear in the text, and the values are the number of times each appears (its frequency). If we know that a word appears in the text, we can look it up in the association like this:

```
origin5KWordFreqs["naturalist"]
```

So the word "naturalist" appears twice in the first part of the Introduction to *Origin*. Here is a word type that does not appear in that portion of the text.

```
origin5KWordFreqs["naturist"]
```

Let's get an idea of what an association looks like. We can see that it is delimited using a pair of

characters, an angle bracket and the vertical bar <|...|>. Elements are separated by commas, as with a list. As with lists, we can use **Length** to find the number of elements in an association.

```
Short[origin5KWordFreqs, 6]
```

```
Length[origin5KWordFreqs]
```

Since each word type occurs only once in the word frequency association, this shows us that there are about 380 word types in the first five thousand characters of *Origin*. Some of those, like the lowercase 'c' that resulted when we accidentally cut a word in half, aren't really word types. Earlier we discovered there are 862 word tokes in the first five thousand characters of *Origin* (and, again, some of those aren't real words either). In both cases, however, we now have good estimates that we could compare with, say, the number of word types and tokens appearing in the first 5000 characters of another 19th-century work.

We can see that the **WordCounts** command has sorted the entries in the association in order of descending frequency. This is usually what we want.

Each entry in the *origin5KWordFreqs* association consists of a **Rule** associating a key to a value. In the rule

```
the → 44
```

"the" is the key (a word type), and 44 is its value (the number of tokens). We can get a list of distinct words in this part of the text by requesting all of the keys in the association. We wrap the command in **Short** to abbreviate the list for display.

```
Short[Keys[origin5KWordFreqs], 5]
```

We can also get all of the values, too.

```
Short[Values[origin5KWordFreqs], 5]
```

If we visualize the distribution of frequencies, we find that a few words occur very frequently, and a lot of words occur very infrequently. This is known as *Zipf's Law*. Here we use the **ListLinePlot** command, which is like **ListPlot** except it draws a line from one point on the graph to the next. We use the *PlotRange→Full* option to tell *Mathematica* not to cut off any portion of the figure. Try copying the code cell and removing that option to see how it changes the resulting figure.

```
ListLinePlot[Values[origin5KWordFreqs], PlotRange → Full,
 AxesLabel → {"Word Number", "Number of Occurences"}]
```

In the figure above, "the" is word number 1, the most frequent word type. It occurs 44 times. The next most frequent word type is "of", which occurs 37 times. It is word number 2. Word number 3 is "to", which occurs 33 times, and so on.

Word clouds and stopwords

One common way to visualize word frequency information is in the form of a *word cloud*, where the font size of each word type is scaled to its frequency. These are quite easy to create in *Mathematica*. We can see that the most common words ('the', 'of', 'to') tell us little about what is distinct in this text. These words are known as *stopwords*.

```
WordCloud[origin5KWordFreqs]
```

To make the word cloud visualization more informative, we want to remove the stopwords before plotting it. We can use the **DeleteStopwords** command to remove the stopwords from our text string before counting words. This visualization gives us a much better idea of what this particular text is about.

```
origin5KWordFreqsNoStop =
  WordCounts[DeleteStopwords[origin5K], IgnoreCase → True];
```

```
WordCloud[origin5KWordFreqsNoStop]
```

The **DeleteStopwords** command only works for English, at least right now, but you will learn another method below which lets you remove arbitrary stopwords.

Searching through the text

Looking at the word cloud, we can see that the word 'species' plays an important role in this text. It would be nice to be able to quickly look at those places where it occurs. One way to do this is to first break our text into a list of words with the **TextWords** command.

```
origin5KWords = TextWords[origin5K];
```

```
Short[origin5KWords, 5]
```

It is often convenient to *normalize* the text: convert it to lowercase and remove all diacritics (accent characters). The following command shows how to do this.

```
origin5KWordsNormalized = ToLowerCase[RemoveDiacritics[origin5KWords]];
```

```
Short[origin5KWordsNormalized, 5]
```

Now we can locate instances of a particular word token in the list, like 'species', with the **Position** command. This shows us that the 53rd word in the normalized word list is 'species', as is the 261st, and so on. For the time being, ignore the extra set of curly braces around each of the positions in the result list. These will become important later on.

```
Position[origin5KWordsNormalized, "species"]
```

We can use the **Span** command to look at five words on either side of the first instance of 'species' (position 53) in the list.

```
origin5KWordsNormalized[[48 ;; 58]]
```

Here is the context in which the last token of 'species' appears in this part of the text (position 675).

```
origin5KWordsNormalized[[670 ;; 680]]
```

We don't want to have to do this by hand for each instance of the word, however. The process of looking at the context for a particular keyword is automated in the 'Generalizing the Examples' section below.

Summary

We began our study of digital research methods with text, since that is the most common kind of source for most research projects, and with raw text files, since they are the easiest to process. Whether we need to read a text closely or not, we can complement our understanding of the text with computational analyses. Some of the simplest of these methods include counting words in sentences and sections, and determining the frequency with which different words appear. We also began to consider ways of searching through texts and studying words in context, two problems that will become easier after we study *pattern matching* in Chapter 2.

In addition to numbers, we worked primarily with three kinds of fundamental constructs in the *Mathematica* language: strings, lists and associations. These will play a significant role in most of our digital research methods.

Generalizing the Examples

Studying sentence lengths across the whole book

In the 'Basic Examples' section above we measured sentence lengths for the sentences appearing in the first 5000 characters of the text. The code below plots sentence lengths for the whole of *Origin*. It takes a little while to run because it is doing a lot of analysis.

```
originSentences = TextSentences[origin];
originSentenceLengths = Map[WordCount, originSentences];

ListPlot[originSentenceLengths, Filling → Axis,
 AxesLabel → {None, "Length in Words"}, ImageSize → Full, PlotRange → Full]
```

Note that there appear to be vertical white lines at regular intervals. These indicate portions of the text where the sentences are shorter than fifteen or twenty words. Why might this be the case?

The first one of these gaps appears somewhere in the first two hundred sentences. Let's zoom in to plot that region by using **Part** to select part of the list of sentence lengths. Using the *PlotRange→15* option for **ListPlot** trims off the top of the graph. Looking at the next graph, it is pretty clear that there is a run of short sentences between positions 50 and 70.

```
ListPlot[originSentenceLengths[[1 ;; 200]], Filling → Axis,
 PlotRange → 15, AxesLabel → {None, "Length in Words"}, ImageSize → Full]
```

We can simply list these sentences and see if we can figure out what is going on. Sending the output through the **TableForm** command puts one sentence on each line. Right after the chapter title, we can see that there are a number of very short sentences that describe the content of the chapter.

```
originSentences[[50 ;; 70]] // TableForm
```

Now that we know that chapter titles appear in uppercase and that they might be followed by a number of short, summary phrases, it would be nice to search through the whole text and see if this pattern continues. One approach might be to use **StringPosition** to locate the second chapter title

```
StringPosition[origin, "CHAPTER 2"]
```

Then use **StringTake** to grab the next few hundred characters, **TextSentences** to break them into sentences, and **TableForm** to lay the results out with one sentence per line.

```
StringTake[origin, {79 817, 79 817 + 300}] // TextSentences // TableForm
```

We could continue in this vein. But in Chapter 2 we will learn about pattern matching, which makes this kind of searching and analysis much easier.

Defining a function for displaying the context of a word

In the 'Basic Examples' section above, we used **Position** to find the locations where the word 'species' appeared in the normalized word list, then looked at words to either side. Although we did this in a number of steps, it would be much more convenient if we could just type in one command. Programming languages like *Mathematica* allow you to define your own functions to *extend* the language. We will do that for word searching.

Our sequence of steps looked like this

```
origin5KWords = TextWords[origin5K];
origin5KWordsNormalized = ToLowerCase[RemoveDiacritics[origin5KWords]];
Position[origin5KWordsNormalized, "species"]
```

At each intermediate step, we saved the output by assigning it to a symbol like *origin5KWords*. When you are first developing a workflow, it is a good idea to do this so that you can check your work at each stage. (The process of creating a series of intermediate steps is visualized in the 'Further Exploration' section below). Once you know what steps you want to apply, however, you

don't have to save intermediate outputs unless you know you are going to want them for later analysis. We can *roll up* multiple steps into a single command by nesting them as follows.

```
Position[ToLowerCase[RemoveDiacritics[TextWords[origin5K]]], "species"]
```

This command does in one step what we did in three steps before. Note that it is still specific to the string we are processing (*origin5K*) and the keyword we are searching for ("species"). In order to make it more general-purpose, we want to substitute symbols. Let's use the symbol *textstring* for the text string that we are searching through, and the symbol *keyword* for the keyword we are interested in. That will look like this

```
Position[ToLowerCase[RemoveDiacritics[TextWords[textstring]]], keyword]
```

Note that we didn't put the symbol name *keyword* in quotes. Next, we have to create the body of the function and say something about what kind of inputs it is expecting. The function that we are defining is named *keywordSearch1*. It takes two *arguments*, a text to search through and a keyword to search for. It returns a list of positions. When we specify the arguments in the function definition, we have to use *named patterns* (more about these later) so each argument name is followed by a single underscore character.

```
keywordSearch1[textstring_, keyword_] :=
 Position[ToLowerCase[RemoveDiacritics[TextWords[textstring]]], keyword]
```

Now we can try using our function as follows

```
keywordSearch1[origin5K, "species"]
```

When we *call* the function, the symbol *textstring* inside the function is assigned the value of *origin5K*, and the symbol *keyword* inside the function is assigned the value of the string "species". The function *returns* a list of positions.

This is all very well and good, but it would be better if our function actually returned the context surrounding each token of the keyword. In order to get it to do this, we are going to have to make a few changes. First, for reasons that will soon become more clear, we will want our list of positions to be a simple list of numbers, and not have the extra curly braces. *Mathematica* has a command called **Flatten** which will do this. We create a second version of the function which outputs a flattened list.

```
Flatten[{{a}, {b}, {c}, {d}}]
```

```
keywordSearch2[textstring_, keyword_] :=
 Flatten[
  Position[ToLowerCase[RemoveDiacritics[TextWords[textstring]]], keyword]]
```

```
keywordSearch2[origin5K, "species"]
```

Next, it would be better if we assigned our normalized list of words to a temporary symbol inside of the function. We can use the **Module** command to do this. Our third version of the search function looks like this:

```
keywordSearch3[textstring_, keyword_] :=
 Module[{normalized},
  normalized = ToLowerCase[RemoveDiacritics[TextWords[textstring]]];
  Flatten[Position[normalized, keyword]]]
```

```
keywordSearch3[origin5K, "species"]
```

(You can learn more about why you might use a command like **Module** in this tutorial on modules and local variables.)

Note that the third version of the function returns exactly the same list as the second version. We haven't changed the output with this modification. Our function will be easier to understand (if a bit

less terse) if we also assign our position list to a different temporary symbol. Here is the fourth version, which also returns the same list as the second and third versions. (Changing the way a function works internally without changing its inputs or outputs is known as *refactoring*).

```
keywordSearch4[textstring_, keyword_] :=
 Module[{normalized, positions},
  normalized = ToLowerCase[RemoveDiacritics[TextWords[textstring]]];
  positions = Flatten[Position[normalized, keyword]];
  Return[positions]]
```

```
keywordSearch4[origin5K, "species"]
```

Note that we can use the **Return** command to be explicit about the value that the function should return. Otherwise the function will simply return the output of its last command. We still haven't returned the actual contexts for the keyword. The fifth version of our function accomplishes this.

```
keywordSearch5[textstring_, keyword_] :=
 Module[{normalized, positions},
  normalized = ToLowerCase[RemoveDiacritics[TextWords[textstring]]];
  positions = Flatten[Position[normalized, keyword]];
  Return[Map[normalized[[# - 5 ;; # + 5]] &, positions]]]
```

Understanding the **Map** expression in the fifth version of the function requires some background that is developed in the 'Programming with *Mathematica*' section below. In the meantime, however, we can see that our function now returns a list of contexts in which the keyword appears.

```
keywordSearch5[origin5K, "species"]
```

We can try it for a few different keywords

```
keywordSearch5[origin5K, "geological"]
```

```
keywordSearch5[origin5K, "conclusions"]
```

Being able to study keywords in context is such an important task in textual analysis that we will return to it again in more detail later.

Programming with *Mathematica*

Combining commands

In the discussion on defining functions for doing keyword searching, we noted that it is possible to combine multiple commands into one by nesting them. In fact, *Mathematica* gives the programmer a tremendous range of options for putting commands together to achieve a particular result.

Suppose, for example, we want a list of the letters from 'a' to 'g'. The **CharacterRange** command will generate such a list.

```
CharacterRange["a", "g"]
```

We can convert them **ToUpperCase**

```
ToUpperCase[CharacterRange["a", "g"]]
```

And put them in **Reverse** order

```
Reverse[ToUpperCase[CharacterRange["a", "g"]]]
```

Nested like this, the command above says 'first generate a list of characters from 'a' to 'g', then convert them to uppercase, then reverse the list.' In this chapter, however, we've also seen a couple of examples that make use of a double forward slash. Known as postfix notation, this allows us to write the same commands in different order to get the same result.

```
CharacterRange["a", "g"] // ToUpperCase // Reverse
```

This notation is particularly useful when you want to distinguish conceptually, say, between computing something and displaying it in a certain way (as we did above when we computed a mean then used **N** with postfix notation to output it as a decimal number.) Postfix notation can also be useful if we think of our input as flowing through a series of transformations on its way to becoming output.

There is one other common notational form, and that is *infix* notation. In *Mathematica* you can add three numbers by writing the **Plus** command as follows

```
Plus[2, 3, 4]
```

But it is more common simply to write

```
2 + 3 + 4
```

In this case, the plus sign is infix notation for the **Plus** command. We have already seen another example, the **Span** command, which uses a pair of semicolons between the beginning and ending elements.

```
{a, b, c, d, e}[[2 ;; 3]]
```

Infix notation can be quite useful for commands that are conceptualized as joining things together. *Mathematica* has an **Join** command for joining lists

```
Join[{a, b}, {c, d, e}]
```

If you wish, you can write this using infix notation as

```
{a, b} ~ Join ~ {c, d, e}
```

Later we will make use of a command called **StringJoin** which does the same thing for strings. It can be written in either of the forms shown below.

```
StringJoin["abc", "de"]
```

```
"abc" <> "de"
```

Bag of words

Here is another example of combining individual commands to transform an input into a desired output. Many methods used in the statistical analysis of text treat text as a *bag of words*. In the bag of words representation, text is normalized, word order is lost and only word types are kept. The following example converts a longish sentence from *Origin* into a bag of words.

```
origin5KSentences[[26]]
```

```
origin5KSentences[[26]] // ToLowerCase // TextWords // Union
```

If we wanted to delete stopwords, too, we could add that command to our chain.

```
origin5KSentences[[26]] // ToLowerCase // TextWords // DeleteStopwords // Union
```

Mathematica also has a *prefix* notation which effectively allows us to reverse the above order. Later we will see examples where this is particularly useful.

```
Union@TextWords@ToLowerCase@origin5KSentences[[26]]
```

```
Union@DeleteStopwords@TextWords@ToLowerCase@origin5KSentences[[26]]
```

Pure functions and more about Map

Sometimes it is convenient to be able to use a custom function without assigning it to a symbol. For example, suppose we wanted to make a list containing three copies of something, anything. (Not

sure why we'd want to do this, but stick with me.) We could define a function like this

```
threeCopies[x_] :=
 {x, x, x}
```

```
threeCopies[3]
```

```
threeCopies["dog"]
```

We have already seen that the **Map** command allows us to apply a function to each element of a list and return a list of the results. So we could do something like this, too

```
Map[threeCopies, {"badger", "cat", "dog"}]
```

But if we only want to use the *threeCopies* function inside of **Map** statements, we don't need to go through the trouble of defining it separately. Instead, we can use a *pure function* to do the same thing. In a pure function, we use a pound sign (#) to stand for the argument of the function, and we put an ampersand (&) at the end of it. Here is a pure function to replace *threeCopies*

```
{#, #, #} &
```

Here is the equivalent to the above command

```
Map[{#, #, #} &, {"badger", "cat", "dog"}]
```

If we change our mind and actually want lists with four copies instead of three, we just have to modify the pure function a little bit. We don't need to define another separate, standalone function.

```
Map[{#, #, #, #} &, {"badger", "cat", "dog"}]
```

The use of pure functions gets easier to understand with repeated exposure. The **Range** command returns a list of numbers

```
Range[10]
```

Here we use **Map** and a pure function to add 42 to each number between 1 and 10.

```
Map[# + 42 &, Range[10]]
```

We can test to see whether one number is less than another with **Less**

```
Less[7, 13]
```

```
Less[13, 7]
```

Here is how we use **Map** and a pure function to test a whole list of numbers to see if they are less than 7

```
Range[10]
```

```
Map[Less[#, 7] &, Range[10]]
```

The name of the ampersand when used in pure functions is **Function**, and the name of the pound sign is **Slot**.

Further Exploration

Getting more information about ExampleData

ExampleData in *Mathematica*, including texts, has metadata that can be retrieved by asking for *Properties*. Each of these properties can then be requested separately. What properties are available for *Origin*?

```
ExampleData[{"Text", "OriginOfSpecies"}, "Properties"]
```

Who wrote the text?

```
ExampleData[{"Text", "OriginOfSpecies"}, "Author"]
```

What is the full title for the publication?

```
ExampleData[{"Text", "OriginOfSpecies"}, "FullTitle"]
```

What language is the text written in?

```
ExampleData[{"Text", "OriginOfSpecies"}, "Language"]
```

Visualizing workflows as networks

If you give *Mathematica* a list of **Rules** of the form *a→b*, you can use **LayeredGraphPlot** to visualize the resulting network. The *VertexLabeling→True* option associates each node (or vertex) of the network with its name. *The ImageSize→Small* option scales the output so the graph is relatively small.

```
LayeredGraphPlot[{a → b , a → c, b → c, c → d, c → e},
 VertexLabeling → True, ImageSize → Small]
```

We can also associate an edge (i.e., a connection between two vertices) with a label as follows...

```
LayeredGraphPlot[
 {{a → b, "1st"} , {a → c, "3rd"}, {b → c, "2nd"}, {c → d, "4th"}, {c → e, "5th"}},
 DirectedEdges → True, VertexLabeling → True, ImageSize → Small]
```

Let's make use of **LayeredGraphPlot** to visualize the sequence of transformations and intermediate steps by which we analyzed part of *Origin of Species* and ultimately visualized our results. We started by assigning the full text of *Origin* to the symbol *origin* with **ExampleData**, then took the first 5000 characters of the text. In our network we will use ORIG to represent the original text.

```
LayeredGraphPlot[
 {{"ORIG" → "origin", "ExampleData"}, {"origin" → "origin5K", "StringTake"}},
 VertexLabeling → True, ImageSize → Small]
```

Note that we are using strings to label our edges and vertices. We don't want *Mathematica* to use the actual symbols or functions. Note also that **LayeredGraphPlot** takes care of figuring out the layout of the network for us. We do have the ability to alter the layout if we don't like it, but we will mostly stick to defaults for now.

Next we used **TextSentences** to generate a list of sentences, then used **Map** and **WordCount** to determine how long each sentence was. We visualized the results with **ListPlot**. Let's use *VIS* to represent a visualization in our workflow. As our figure starts to become more crowded, we increase the *ImageSize*.

```
LayeredGraphPlot[
 {{"ORIG" → "origin", "ExampleData"}, {"origin" → "origin5K", "StringTake"},
  {"origin5K" → "origin5KSentences", "TextSentences"},
  {"origin5KSentences" → "origin5KSentenceLengths", "Map WordCount"},
  {"origin5KSentenceLengths" → "VIS", "ListPlot"}},
 DirectedEdges → True, VertexLabeling → True, ImageSize → Medium]
```

Next we used **WordCounts** on *origin5K* to generate an association list called *origin5KWordFreqs*, then we used **WordCloud** to visualize this. As you add more vertices and edges (especially labeled ones) figures become crowded. Here we use the *VertexCoordinateRules* option to place each vertex in a specific position relative to the others.

```
LayeredGraphPlot[
 {{"ORIG" → "origin", "ExampleData"}, {"origin" → "origin5K", "StringTake"},
  {"origin5K" → "origin5KSentences", "TextSentences"},
  {"origin5KSentences" → "origin5KSentenceLengths", "Map WordCount"},
  {"origin5KSentenceLengths" → "VIS", "ListPlot"},
  {"origin5K" → "origin5KWordFreqs", "WordCounts"},
  {"origin5KWordFreqs" → "VIS", "WordCloud"}}, DirectedEdges → True,
 VertexLabeling → True, ImageSize → Medium, VertexCoordinateRules →
  {{0, 2}, {0, 1}, {0, 0}, {-1, -1}, {-1, -2}, {0, -3}, {2, -1}}]
```

But the stopwords were dominating the visualization, so we started again with *origin5K* and applied some different transformations. Our overall workflow looked like this:

```
LayeredGraphPlot[
 {{"ORIG" → "origin", "ExampleData"}, {"origin" → "origin5K", "StringTake"},
  {"origin5K" → "origin5KSentences", "TextSentences"},
  {"origin5KSentences" → "origin5KSentenceLengths", "Map WordCount"},
  {"origin5KSentenceLengths" → "VIS", "ListPlot"},
  {"origin5K" → "origin5KWordFreqs", "WordCounts"},
  {"origin5KWordFreqs" → "VIS", "WordCloud"},
  {"origin5K" → "origin5KWordFreqsNoStop", "WordCounts DeleteStopwords"},
  {"origin5KWordFreqsNoStop" → "VIS", "WordCloud"}}, DirectedEdges → True,
 VertexLabeling → True, ImageSize → Large, VertexCoordinateRules →
  {{1, 2}, {1, 1}, {1, 0}, {-1, -1}, {-1, -2}, {2, -3}, {2, -1}, {5, -1}}]
```

When you are first developing a workflow, you often go down a number of blind alleys, do things inefficiently, and save intermediate steps that you end up not needing. Taking a moment to visualize your workflow can help you think of ways to make it more efficient or suggest operations you haven't yet explored. It can also serve as a useful form of documentation if you put a project aside for a while. Imagine coming back to the Darwin analysis in six months or a year and trying to figure out what *origin5KSentenceLengths* was and where it came from. Being able to look at an image like this helps you to reorient yourself quickly.

Mathematica Commands to Review

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration
- Association (BE)
- CharacterRange (PM)
- DeleteStopwords (BE)
- ExampleData (BE)
- Flatten (GE)
- Function (PM)
- GraphPlot (FE)
- Head (BE)
- Join (PM)
- Keys (BE)
- Length (BE)
- Less (PM)
- List (BE)
- ListLinePlot (BE)

- ListPlot (BE)
- Map (BE)
- Mean (BE)
- Module (GE)
- Part (BE)
- Position (BE)
- Postfix (PM)
- Prefix (PM)
- Range (PM)
- RemoveDiacritics (BE)
- Return (GE)
- Reverse (PM)
- Rule (FE)
- Short (BE)
- Slot (PM)
- Span (BE)
- String (BE)
- StringJoin (PM)
- StringLength (BE)
- StringPosition (GE)
- StringTake (BE)
- TableForm (GE)
- TextSentences (BE)
- ToLowerCase (BE)
- ToUpperCase (PM)
- Union (PM)
- Values (BE)
- WordCloud (BE)
- WordCount (BE)

Problems

```
Problem 1.1
Input: "This is a TEST"
Output: {This, is, a, TEST}
```

```
TextWords["This is a TEST"]
```

```
Problem 1.2
Input: {g,a,a,e,c,f,b,d,h}
Output: {a,b,c,d,e,f,g,h}
```

```
Union[{g, a, a, e, c, f, b, d, h}]
```

```
Problem 1.3
Input: "This is a TEST"
Output: {THIS,IS,A,TEST}
```

```
TextWords[ToUpperCase["This is a TEST"]]

TextWords@ToUpperCase@"This is a TEST"

"This is a TEST" // ToUpperCase // TextWords
```

```
Problem 1.4
Input: {a,b,c,d,e,f,g,h}
Output: {c,d}
```

```
{a, b, c, d, e, f, g, h}[[3 ;; 4]]

CharacterRange["a", "g"][[3 ;; 4]]
```

```
Problem 1.5
Input: {{a,b},{c,d},{e,f},{g,h}}
Output: {b,d,f,h}
```

```
{{a, b}, {c, d}, {e, f}, {g, h}}[[All, 2]]

{{a, b}, {c, d}, {e, f}, {g, h}}[[1 ;; 4, 2]]

{{a, b}, {c, d}, {e, f}, {g, h}}[[{1, 2, 3, 4}, 2]]
```

```
Problem 1.6
Input: {{a,b},{c,d},{e,f},{g,h}}
Output: {h,g,f,e,d,c,b,a}
```

```
Reverse[Flatten[{{a, b}, {c, d}, {e, f}, {g, h}}]]
```

```
Problem 1.7
Input: {{a,b},{c,d},{e,f},{g,h}}
Output: {g,h,e,f,c,d,a,b}
```

```
Flatten[Reverse[{{a, b}, {c, d}, {e, f}, {g, h}}]]
```

```
Problem 1.8
Input: {{a},{b,c},{d,e,f},{g},{h}}
Output: {1,2,3,1,1}
```

```
Map[Length, {{a}, {b, c}, {d, e, f}, {g}, {h}}]
```

```
Problem 1.9
Input: "This is a test"
Output: {4,2,1,4}
```

```
Map[StringLength, TextWords["This is a test"]]
```

```
Problem 1.10
Input: "This is a test"
Output: "test"
```

```
StringTake["This is a test", -4]

StringTake["This is a test", {11, 14}]
```

```
Problem 1.11
Input: "The quick brown fox jumped over the lazy dog"
Output: {{1,3},{33,35}}
```

```
StringPosition["The quick brown fox jumped over the lazy dog",
 "the", IgnoreCase → True]
```

```
StringPosition[
 ToLowerCase["The quick brown fox jumped over the lazy dog"], "the"]
```

```
Problem 1.12
Input: {d,a,d,a,g,a,g,a}
Output: {{2},{4},{6},{8}}
```

```
Position[{d, a, d, a, g, a, g, a}, a]
```

```
Problem 1.13
Input: {{d,a,d,a},{g,a},{g,a}}
        d   a   d   a
Output: g   a
        g   a
```

```
TableForm[{{d, a, d, a}, {g, a}, {g, a}}]
```

```
Problem 1.14
Input: 12
Output: {1,2,3,4,5,6,7,8,9,10,11,12}
```

```
Range[12]
```

```
Problem 1.15
Input: {a,b,c},{d,e},{f}
Output: {a,b,c,d,e,f}
```

```
Join[{a, b, c}, {d, e}, {f}]
```

```
{a, b, c}~Join~{d, e}~Join~{f}
```

Exercises

1. (*Down the rabbit hole*). *Mathematica* includes a number of other sample texts to experiment with. Using the Darwin example above as a guide, explore Lewis Carroll's *Alice in Wonderland* (1865). You can load the text with the command shown below. Choose a portion of the text and analyze the number of words, sentences, and words per sentence. Explore word frequencies, and look at a few of the contexts in which some interesting keywords appear.

```
carroll = ExampleData[{"Text", "AliceInWonderland"}];
```

---

2. (*Sentence lengths*). You can get a list of sample texts that are included in *Mathematica* with the command below. Choose a different text and try studying the sentence lengths across the whole work. Are there any interesting patterns?

```
ExampleData["Text"]
```

---

3. (*Another way to compute word frequencies*). The **WordCounts** command was introduced in *Mathematica* in 2015. Prior to that time, if you wanted to determine word frequencies, you had to do it a different way. Try converting one of the sample texts to a word list then using the **Tally** command. Compare your results with the output of **WordCounts** on the same text. You will want to read the *Mathematica* documentation for the **Tally** command.

ch02

Chapter 02: Pattern Matching

Overview

In the first chapter, we learned that we can use word frequencies to get some idea of what a text is about, whether we are already familiar with it or not. Starting with a raw text file in the form of a string, we learned to count the number of words and sentences, to generate an *association* containing word frequencies and to visualize this information in the form of a word cloud. Once we have identified some words and phrases that seem to play an important role in the text, the next step is to study the contexts in which those expressions appear. To do this effectively, we need to know how to match patterns. *Mathematica* has a very powerful set of tools for matching patterns and for transforming them with *rewrite* rules. Facility with pattern matching will also allow us to create a *concordance* for our text, a display that shows us keywords in context.

Basic Examples

Matching string patterns

Once again, we will be working with Darwin's *Origin of Species*. For the time being, we will leave the text in the form of one long string. For this example, let's use the first 20,000 characters.

```
origin = ExampleData[{"Text", "OriginOfSpecies"}];
origin20K = StringTake[origin, 20 000];
```

```
Head[origin20K]
```

```
StringLength[origin20K]
```

The **StringCases** command allows us to search a string for a particular pattern. In order to build a pattern, we use the **StringExpression** command, which allows us to mix explicit strings with more general objects that match more than one string.

Suppose we want to find instances of the word 'domestic'. The **StringExpression** that we will need looks like this

```
StringExpression["domestic", WordBoundary]
```

It says match the string 'domestic' followed by any character that represents a boundary between words. Word boundaries include the beginnings and endings of lines, whitespace, and punctuation marks. The value what was returned here shows a shorthand notation for **StringExpression**, a pair of tilde characters. (This is an example of *infix* notation. There is more information about this in the 'Programming with *Mathematica*' section of Chapter 1.) The following command shows how we use the pattern with **StringCases**. The word 'domestic' appears four times in lowercase in the first 20,000 characters of *Origin*.

```
StringCases[origin20K, StringExpression["domestic", WordBoundary]]
```

The next command is equivalent to the previous command, using the shorthand (infix) notation for **StringExpression.**

```
StringCases[origin20K, "domestic" ~~ WordBoundary]
```

The word 'domestic' also occurs four times with the first character in uppercase.

```
StringCases[origin20K, "Domestic" ~~ WordBoundary]
```

Are there any other occurrences? If we don't want to miss anything, we can use the *IgnoreCase*→

True option for **StringCases**.

```
StringCases[origin20K, "Domestic" ~~ WordBoundary, IgnoreCase → True]
```

If we would like to search for words that begin with a sequence of characters, we can build a more general pattern using the **StringExpression** command. In the example below, three underscores stand for a sequence of zero or more characters. (This command is named **BlankNullSequence**). Note that we have had to wrap that pattern in a command called **Shortest**. Think for a moment about what "zero or more characters" means: if not constrained it could potentially match all of the characters to the end of the *origin20K* string. What we want to say is "match zero or more characters until you hit a word boundary."

```
StringCases[origin20K,
 "domestic" ~~ Shortest[___] ~~ WordBoundary, IgnoreCase → True]
```

We see that in addition to the word 'domestic,' Darwin also uses 'domestication' and 'domesticated' in this part of *Origin*. If we only want to see those words (and not instances of 'domestic') we can use a different pattern. In the following example, **WordCharacter** followed by two dots stands for a sequence of one or more word characters (letters or digits). The two dots are shorthand for the **Repeated** command. The following pattern says 'find the string "domestic" followed by one or more word characters followed by a word boundary.' Since we say there has to be at least one word character following "domestic", we don't match the word "domestic". We would match the word "domestics" if it occurred in the text.

```
StringCases[origin20K,
 "domestic" ~~ WordCharacter .. ~~ WordBoundary, IgnoreCase → True]
```

We can also search for words with a particular ending. Note the increasingly general patterns.

```
StringCases[origin20K,
 WordCharacter .. ~~ "ological" ~~ WordBoundary, IgnoreCase → True]
```

```
StringCases[origin20K,
 WordCharacter .. ~~ "ical" ~~ WordBoundary, IgnoreCase → True]
```

```
StringCases[origin20K, WordCharacter .. ~~ "al" ~~ WordBoundary, IgnoreCase → True]
```

Making use of WordData

In the case of a pair of related words like 'domestic' and 'domesticate', the root word is a proper *substring* of the derived form. Not all English words have such clean morphology. Suppose you want to look for expressions that contain words derived from or related to the verb 'decide'. You can use *Mathematica*'s built-in **WordData** to facilitate your search. The following command shows us that there are (at least) four different senses of 'decide'.

```
WordData["decide"]
```

We can request the *InflectedForms* property to get other forms of the verb. The inflected forms of the verb 'decide' are the same for each of the four different senses.

```
WordData["decide", "InflectedForms"]
```

If we wanted to search for all of these possibilities, we can see that they all begin with the substring 'decid'. So we could use a command like the following. Only one of the inflected forms appears in this section of *Origin*.

```
StringCases[origin20K,
 "decid" ~~ Shortest[___] ~~ WordBoundary, IgnoreCase → True]
```

The part of the word that we are searching for in the command above is known as the *stem*. In fact, there is an algorithm called *Porter Stemming* which automatically removes common endings, and the *Mathematica* **WordStem** command does this. So we could also search for inflected forms as

follows.

```
StringCases[origin20K,
 WordStem["decide"] ~~ Shortest[___] ~~ WordBoundary, IgnoreCase → True]
```

Here is another example, starting with a different verb.

```
StringCases[origin20K,
 WordStem["admire"] ~~ Shortest[___] ~~ WordBoundary, IgnoreCase → True]
```

If we want to find words that are related to 'decide', we can request the *MorphologicalSource* property. Note that this turns up some terms that we wouldn't have caught when we searched using the word stem, like 'decision' and 'decisive'.

```
WordData["decide", "MorphologicalSource"]
```

We see that 'decision' does occur in our text (but 'decisive' does not).

```
StringCases[origin20K,
 "decis" ~~ Shortest[___] ~~ WordBoundary, IgnoreCase → True]
```

Generating a concordance

We can use string patterns to generate a listing that shows a particular keyword in context. The set of all such listings is known as a concordance, or a *keyword in context* (KWIC) listing. We begin by trying to match a pattern which includes the keyword and one following word. In the **StringExpression** below, **Whitespace** matches one or more whitespace characters and **WordCharacter** followed by two dots matches one or more word characters. We send the output through **TableForm** to print one match per line. The following pattern says 'match the string "domestic" followed by any amount of whitespace (blank spaces, tabs, etc.) followed by one or more word characters.'

```
StringCases[origin20K,
  "domestic" ~~ Whitespace ~~ WordCharacter .., IgnoreCase → True] // TableForm
```

Now we can add a similar sequence on the left to match the preceding word. In doing so, we lose one of our instances (domestic pigeons), which we will fix below.

```
StringCases[origin20K, WordCharacter .. ~~ Whitespace ~~ "domestic" ~~
   Whitespace ~~ WordCharacter .., IgnoreCase → True] // TableForm
```

The problem with the previous expression is that the pattern of word characters interspersed with whitespace is too specific to match cases where one sentence ends and another begins. Instead we should be looking for sequences of word characters interspersed with sequences of non-word characters. The **Except** command allows us to specify a pattern consisting of anything except what we don't want to match.

```
StringCases[origin20K,
  WordCharacter .. ~~ Except[WordCharacter] .. ~~ "domestic" ~~
   Except[WordCharacter] .. ~~ WordCharacter .., IgnoreCase → True] // TableForm
```

Now suppose we want to see a window of three words to either side of our keyword. In *Mathematica*, the **With** command allows us to temporarily assign a value to a symbol. That can save us a lot of typing in a situation like this where we are repeating a pattern over and over. We set the temporary symbol *w* to match one or more word characters followed by one or more non-word characters, then we use multiple copies of it in our **StringExpression**.

```
With[{w = WordCharacter .. ~~ Except[WordCharacter] ..},
  StringCases[origin20K, w ~~ w ~~ w ~~ "domestic" ~~
    Except[WordCharacter] .. ~~ w ~~ w ~~ w, IgnoreCase → True]] // TableForm
```

We will clean up the output and turn this into a standalone function in the 'Programming with *Mathematica*' section below.

Capitalized words and phrases

The first word of a sentence in English is usually capitalized. Elsewhere in the sentence capitalized words and phrases are often of special interest. Among other things, they may refer to people, places, institutions, dates, works of art, acronyms, vessels, brand names, deities or personifications. In later chapters we will focus on doing interesting things with these entities, but for now we concentrate on the problem of finding them in text.

We start by breaking our long string into a list of sentences, using techniques we learned in Chapter 1.

```
origin20KSentences = TextSentences[origin20K];
```

```
Head[origin20KSentences]
```

```
Length[origin20KSentences]
```

Next we want to split each of those sentences into a list of words. The **Map** command makes a task like this easy.

```
origin20KWordLists = Map[TextWords, origin20KSentences];
```

```
Head[origin20KWordLists]
```

The result is still a list, but instead of being a list of strings, it is a list of lists. This kind of structure is known as a *nested list*. If we use the **Head** command on the first element of the list of word lists, we find that it is also a list.

```
origin20KWordLists[[1]]
```

```
Head[origin20KWordLists[[1]]]
```

Let's look at the second element in our list of word lists.

```
origin20KWordLists[[2]]
```

As expected, the first word is capitalized, but so are "H.M.S. Beagle," "I" and "South America." These are the kind of phrases we want to find. The **Rest** command returns the list with the first element removed.

```
Rest[origin20KWordLists[[2]]]
```

We can take the first character of a string with **StringTake**, and test to see if it is uppercase with **UpperCaseQ**. In *Mathematica*, functions that ask a yes-no question and return either *True* or *False* are named so they end with a capital Q. We will see more examples later on.

```
StringTake["Beagle", 1]
```

```
UpperCaseQ[StringTake["Beagle", 1]]
```

Let's create a small function that tests to see if a word is capitalized. In keeping with the *Mathematica* convention, we name this function so it ends in a capital Q.

```
capitalizedQ[w_] := UpperCaseQ[StringTake[w, 1]]
```

```
capitalizedQ["Beagle"]
```

```
capitalizedQ["naturalist"]
```

Now we use the **Select** command and the function that we just defined to pull out all of the capitalized words that occur within the second sentence (note that we don't include the first word, which is always capitalized by convention).

```
Select[Rest[origin20KWordLists[[2]]], capitalizedQ]
```

If we want to find all of the capitalized words in this section of *Origin*, we can build up the desired command one step at a time. To begin with, we need to exclude the first word of each sentence. We can do this by mapping **Rest** across our list of word lists. The output will be quite long, so we use **Short** to have a look at it.

```
Short[Map[Rest, origin20KWordLists], 20]
```

Now that we've gotten rid of the first word in each sentence, we don't care where the sentence boundaries are anymore. So we can **Flatten** the list.

```
Short[Flatten[Map[Rest, origin20KWordLists]], 20]
```

We can now **Select** all of the capitalized words from this list. The **Select** command pulls out elements for which a given condition is *True*. We have few enough results that we don't have to use **Short** to view them.

```
Select[Flatten[Map[Rest, origin20KWordLists]], capitalizedQ]
```

Finally, we want to get rid of duplicates, so we use the **Union** command to create a *bag of words* representation. (There is more information about this in the 'Programming with *Mathematica*' section of Chapter 1.)

```
Union[Select[Flatten[Map[Rest, origin20KWordLists]], capitalizedQ]]
```

In a number of cases it would be more useful to have a list of capitalized phrases rather than capitalized words. We want to know, in other words, that "South" goes with "America" and "Linnean" goes with "Society." We will return to this problem.

N-gram analysis

A sequence of *n* words in a text is known as an *n-gram*. If we provide the **WordCounts** command with a parameter, we can use it to count n-grams rather than words. The results are returned in an association

```
origin20KBigrams = WordCounts[origin20K, 2, IgnoreCase → True];
```

```
Head[origin20KBigrams]
```

```
Length[origin20KBigrams]
```

If we look the twenty most common bigrams (i.e., 2-grams) in this part of *Origin*, we see that they almost all contain one or more stopwords. The only exception is 'organic beings'.

```
origin20KBigrams[[1 ;; 20]]
```

Does Darwin describe anything else as 'organic' in this part of Origin? In order to answer this question, first we use the **Keys** command to pull all of the bigrams out of the association, then we use the **Cases** command to choose those that have "organic" as the first word and any expression (the single underscore) as the second word. (The single underscore is shorthand for a command called **Blank**). We find that in this part of the text, the word "organic" is always followed by "beings".

```
Short[Keys[origin20KBigrams], 5]
```

```
Cases[Keys[origin20KBigrams], {"organic", _}]
```

Given this bigram association, we have a different way of searching for places where Darwin used the word "domestic" in this part of the text.

```
Cases[Keys[origin20KBigrams], {"domestic", _}]
```

We can count the number of times each appears with the following command.

```
Lookup[origin20KBigrams, Cases[Keys[origin20KBigrams], {"domestic", _}]]
```

Does Darwin refer to other kinds of ducks or pigeons here than domestic ones? No.

```
Cases[Keys[origin20KBigrams], {_, "duck"}]
```

```
Cases[Keys[origin20KBigrams], {_, "pigeons"}]
```

Does he refer to wild animals? Only in the phrase “wild parent”.

```
Cases[Keys[origin20KBigrams], {"wild", _}]
```

Bigrams that do not contain stopwords

Earlier we noted that the twenty most common bigrams in this part of *Origin* almost all contain stopwords. It would be nice to find the twenty most common bigrams that do not. First we have to grab a list of stopwords using the **WordData** command.

```
stopwords = WordData[All, "Stopwords"];
```

```
Length[stopwords]
```

Next we make a small function that tests whether a word is in the list of stopwords or not. If the word is a member of the stopword list, we want the function to return *False* and if it is not we want it to return *True*.

```
nonStopwordQ[w_] := Not[MemberQ[stopwords, w]]
```

```
nonStopwordQ["the"]
```

```
nonStopwordQ["duck"]
```

Finally we use the **Cases** command on the keys of our bigram association. The pattern we are searching for is a list containing two items. Each of these can be anything (single underscore) as long as it is not a stopword. The *pattern?test* notation matches an expression only if the test is true when applied to the expression. The last step is to use **Part** to select the first 20 results. Since they are ordered by descending frequency, these are the most common bigrams that do not contain stopwords.

```
Cases[Keys[origin20KBigrams], {_?nonStopwordQ, _?nonStopwordQ}][[1 ;; 20]]
```

In this part of *Origin*, Darwin uses the phrase “natural selection” three times.

```
origin20KBigrams[{"natural", "selection"}]
```

Note how much information these bigrams provide about the nature of this text.

Capitalized bigrams

In the example above, we used the *Ignorecase->True* option when creating an association of bigrams with the **WordCounts** command. If we don’t use that option, however, we can find capitalized phrases. As before, we need to remove the first word from each sentence, since it will automatically be capitalized. We did this by mapping **Rest** across our sentences once they had been turned into word lists, then flattening the result.

```
Short[Flatten[Map[Rest, origin20KWordLists]], 20]
```

The **WordCounts** command takes a string as input, not a list, so we have to turn this list of words back into a string. We do this with the **StringRiffle** command, which puts whitespace between each word.

```
StringRiffle[{"a", "b", "c", "d"}]
```

```
Short[StringRiffle[Flatten[Map[Rest, origin20KWordLists]]], 20]
```

Now we can compute all the bigrams and use **Cases** to pull out those with two capitalized words in a row. Note in cases where there is a longer capitalized n-gram it is broken into bigrams, as with 'VARIATION UNDER DOMESTICATION.'

```
origin20KCapBigrams = Cases[
  Keys[WordCounts[StringRiffle[Flatten[Map[Rest, origin20KWordLists]]], 2]],
  {_?capitalizedQ, _?capitalizedQ}]
```

Summary

Mathematica provides a very rich set of pattern matching tools that allow us to find examples of particular words or phases, to find classes of related words, to see words or phrases in context, and to analyze n-gram sequences. This is only the tip of the iceberg, however, and further examples of pattern matching are demonstrated in later sections of this chapter. Methods of pattern matching will also play an important role in later chapters, when we turn our attention to *computable data* and to some of the techniques of *information retrieval*.

Some of the pattern-matching commands that we learned were specific to strings, others work on lists, associations, or other expressions. We also made use of some of *Mathematica*'s built-in knowledge about the English language via the **WordData** command.

Generalizing the Examples

Extracting more sensible parts of a text to study

So far we have simply taken the first few thousand characters of *Origin* when we didn't want to work with the whole text. The problem with this method is that it does not respect the natural boundaries within the work. Now that we know how to match string patterns, however, we can select more meaningful units to study. One way to do this is to look for *internal evidence* from text itself. The word 'introduction' appears at the beginning of the Introduction to *Origin*, the word 'chapter' appears at the beginning of each Chapter, and so on. So even though we have been dealing with this book as a single long string, we have some evidence about how the author (and/or publisher) wanted the book to be understood as a sequence of contained units. Note that, at this point at least, we don't have evidence about how the book was broken into smaller units, especially numbered pages. We will deal with that in a later chapter.

Here is a command that pulls out the Introduction. The two underscores are shorthand for a command called **BlankSequence**. This pattern stands for any sequence of one or more expressions.

```
originIntroduction =
  StringCases[origin, "INTRODUCTION. " ~~ Shortest[__] ~~ "CHAPTER"][[1]];
Short[originIntroduction, 10]
```

Note that this leaves the word "CHAPTER" at the end of the text string and "INTRODUCTION" at the beginning. If we only want the stuff between those two words, we can use a named pattern as follows.

```
Clear[originIntroduction];
originIntroduction =
  StringCases[origin, "INTRODUCTION. " ~~ Shortest[x__] ~~ "CHAPTER" → x][[1]];
Short[originIntroduction, 10]
```

Above we use the **Clear** command to clear the value that has already been assigned to the symbol *originIntroduction*, then we assign a new value to it.

Investigating summary sentences at the beginnings of chapters

While plotting sentence lengths for *Origin* in Chapter 1, we noted that there were bursts of short sentences at regular intervals. These turned out to be summary sentences printed right after the

chapter title. Using our new pattern matching skills, we can investigate this characteristic of the book more easily. The command below pulls out the chapter titles.

```
StringCases[origin,
  "CHAPTER " ~~ DigitCharacter .. ~~ Shortest[__] ~~ "."] // TableForm
```

If we want to see five sentences after each chapter title, we can do it with the following command. Note that passing the output through **TabView** gives us an interface control which puts each chapter summary on its own page. Click the chapter numbers to see the corresponding summary.

```
With[{s = Shortest[__] ~~ "."}, StringCases[origin,
   "CHAPTER " ~~ DigitCharacter .. ~~ s ~~ s ~~ s ~~ s ~~ s ~~ s]] // TabView
```

Finding one word near another

Sometimes you want to know whether two words occur within a certain distance of one another. For small enough distances you can use n-grams, but you probably don't want to compute all the 50-grams for a text just to see if the word 'geological' occurs somewhere near 'geographical'. Here is a function that solves this problem. It takes a text, a pair of strings to search for, and an integer that determines the width (in characters) for the search window. A value of 100, for example, means that the positions of the first character in each search term should be within 100 characters of one another. The details of how this function works are covered in the 'Further Exploration' section below.

```
stringFindNear[txt_, str1_, str2_, within_] :=
 TabView[Map[StringTake[txt, {#[[1]] - 100, #[[2]] + 100}] &,
   Select[DeleteCases[Partition[Sort[Join[Map[List[str1, #] &, StringPosition[
             txt, str1 ~~ WordBoundary][[All, 1]]], Map[List[str2, #] &,
          StringPosition[txt, str2 ~~ WordBoundary][[All, 1]]]], #1[[2]] < #2[[2]] &],
       2, 1], {{x_, _}, {x_, _}}][[All, 1 ;; 2, 2]], (#[[2]] - #[[1]]) ≤ within &]]]
```

And here are some examples of its use.

```
stringFindNear[origin20K, "geological", "geographical", 100]
```

```
stringFindNear[origin20K, "embryo", "monstrosities", 200]
```

```
stringFindNear[origin20K, "cats", "dogs", 400]
```

Here is what the output looks like if a match is not found within the specified distance.

```
stringFindNear[origin20K, "cats", "dogs", 100]
```

We would also like to be able simply to search for a term and see some surrounding context. The function below does that.

```
textSearch[txt_, str_] :=
 TabView[Map[StringTake[txt, {#[[1]] - 100, #[[2]] + 100}] &,
   StringPosition[txt, str ~~ WordBoundary]]]
```

```
textSearch[origin, "aphides"]
```

Programming with *Mathematica*

A function to display long sources with a scrollbar

We have been using the **Short** command to abbreviate any output that would be unnecessarily long if dumped into our notebook. It would be more convenient, however, to display long sources inside a little window with a scrollbar. *Mathematica* has an extensive set of commands that make it easy to build interfaces within notebooks.

We will start with **Short** and give it a parameter that limits the number of lines to display.

```
Short[originIntroduction, 5]
```

We put a box around something with the **Framed** command. As usual, we put commands together by *nesting* them (or, more technically, by *composing* the functions).

```
Framed[Short[originIntroduction, 5]]
```

We don't just want to put a box around the string, however. First we want to put the string onto a background that we can resize, then we will put a box around that. The resizable background is called a **Pane**.

```
Framed[Pane[Short[originIntroduction, 5]]]
```

It still looks the same until we tell *Mathematica* how big we want the **Pane** to be. In this case, we are saying 'make the panel take up the whole width of the notebook (i.e., it is Automatic), but make the height be 200 pixels.'

```
Framed[Pane[Short[originIntroduction, 5], {Automatic, 200}]]
```

Once you start nesting commands, you will often have many layers of brackets nested within one another. If you are not sure how the brackets pair up, try triple-clicking on the leftmost one. The whole expression will be highlighted. If you do that on the left bracket after **Pane** in the above statement, you will see that *{Automatic, 200}* is inside the **Pane** command. Try triple-clicking on the bracket after the **Short** command.

Now let's get rid of the **Short** command (so we can see our whole string) and add some scrollbars to our pane.

```
Framed[Pane[originIntroduction, {Automatic, 200}, Scrollbars → True]]
```

This is useful, so we are going to turn it into a *function* that we can use anytime we need it. We define a function as shown below. The name of our function is **viewData** (best to start with a lower-case letter so our user-defined functions don't get confused with built-in functions). Instead of making the function specific to the variable that we used to test it, we create a new variable called *x*. The x followed by an underscore in the definition is an example of a named pattern.

```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Now we can use our new command to look at any source that might be too long to display without it.

```
viewData[originIntroduction]
```

Testing for matches

The **MatchQ** command tests to see whether a pattern is matched by an expression, returning *True* if so and *False* if not. It is very useful for testing and debugging patterns. Let's use it to learn more about the **Blank** (_) command.

Blank matches a single expression. This might be a symbol, integer, rational number, string, list or other kind of expression.

```
MatchQ[a, _]
```

```
MatchQ[123, _]
```

```
MatchQ[1/2, _]
```

```
MatchQ["Beagle", _]
```

```
MatchQ[{a, b, c}, _]
```

If we want to specify what kind of element to match, we can put the **Head** after the underscore.

```
MatchQ[{a, b, c}, _List]
```

```
MatchQ[{a, b, c}, _Symbol]
```

```
MatchQ[a, _Symbol]
```

When we define a function, we usually use a named **Blank** to stand for each argument. In the function below, the symbol *x* could be assigned the value of any single list.

```
f[x_List] := Length[x]
```

```
f[{a, b, c}]
```

If we try to give this function something that is not a list, or more than one list, the pattern doesn't match.

```
f[a]
```

```
f[{a, b, c}, {d, e}]
```

If we wanted a function definition that could match an arbitrary number of arguments, we could use a named **BlankSequence** (__) instead of a named **Blank**. Instead of measuring the length of a list, this function counts the number of arguments you give it.

```
g[x__List] := Length[{x}]
```

```
g[{a, b, c}]
```

```
g[{a, b, c}, {d, e}, {p, q, r}]
```

Since we are finished with this example, we can **Clear** the function definitions.

```
Clear[f, g]
```

Sorting and pure functions

The **Sort** command puts elements of a list into a canonical order. If the elements are integers they are sorted from least to greatest. If they are symbols, they are sorted into alphabetical order.

```
Sort[{3, 3, 6, 3, 4, 8, 2}]
```

```
Sort[{d, y, w, b, a, u}]
```

If you want the elements in a different order, you have to give **Sort** an *ordering function*. This is usually a pure function. If you want to sort numbers from greatest to least, the ordering function is

```
#1 > #2 &
```

and the command looks like this

```
Sort[{3, 3, 6, 3, 4, 8, 2}, #1 > #2 &]
```

The default order for strings is alphabetical. You can sort them by StringLength as shown in the second of these examples.

```
Sort[{"this", "could", "be", "a", "test", "of", "sorts"}]
```

```
Sort[{"this", "could", "be", "a", "test", "of", "sorts"},
 StringLength[#1] < StringLength[#2] &]
```

If you are sorting nested lists, associations, or something else with internal structure, your ordering function can make use of **Part**. The following examples show how to sort a nested list based on (a)

the first element of each sublist, (b) the second element, and (c) the second element from greatest to least.

```
Sort[{{a, 23}, {f, 11}, {c, 9}, {w, 1}, {s, 2}, {b, 42}}]
```

```
Sort[{{a, 23}, {f, 11}, {c, 9}, {w, 1}, {s, 2}, {b, 42}}, #1[[2]] < #2[[2]] &]
```

```
Sort[{{a, 23}, {f, 11}, {c, 9}, {w, 1}, {s, 2}, {b, 42}}, #1[[2]] > #2[[2]] &]
```

Cleaning up the concordance (KWIC) code

At the point that we finished developing our concordance (keyword in context) code above, it looked like the following.

```
With[{w = WordCharacter .. ~~ Except[WordCharacter] ..},
  StringCases[origin20K, w ~~ w ~~ w ~~ "domestic" ~~ Except[WordCharacter] .. ~~ w
~~ w ~~ w, IgnoreCase -> True]] // TableForm
```

We *hardcoded* the size of the window (three words to either side) and the keyword ('domestic') and we left the output more-or-less unformatted. This code will be more useful, however, if we allow the hardcoded values to vary and wrap everything up in the form of a function.

The function will have three parameters, one each for the text we are searching, the keyword and the window size. The first two parameters should be strings and the third should be an integer. The function should return nicely formatted output. This is what we have got so far

```
kwic[text_String, keyword_String, win_Integer] :=
 Module[{...},
  ...
  Return[...]]
```

The **Module** command inside our function definition allows us to temporarily define symbols. We will use one symbol for the pattern that matches an individual word, one for our window (sequence of word patterns) and one for the results we want the function to return. We can just copy our word pattern code from above. Our function definition now looks like this

```
kwic[text_String, keyword_String, win_Integer] :=
  Module[{wordpattern, window, resultlist},
   wordpattern = WordCharacter .. ~~ Except[WordCharacter] ..;
   ...
   Return[resultlist]]
```

Whenever we have used two dots in a **StringExpression**, we have actually been using shorthand notation for the **Repeated** command. The two expressions below mean the same thing.

```
WordCharacter ..
Repeated[WordCharacter]
```

If we want to specify a pattern where something is repeated *n* times, we can use the following expression.

```
Repeated[pattern, {n}]
```

This makes it easy to specify the size of our window, as follows.

```
kwic[text_String, keyword_String, win_Integer] :=
 Module[{wordpattern, window, resultlist},
  wordpattern = WordCharacter .. ~~ Except[WordCharacter] ..;
  window = Repeated[wordpattern, {win}];
  ...
  Return[resultlist]]
```

At this point we can copy the **StringCases** code from above. Note that we replace the three copies

of our old word pattern with a new one based on **Repeated**. Our function definition now looks like this:

```
kwic[text_String, keyword_String, win_Integer] :=
 Module[{wordpattern, window, resultlist},
  wordpattern = WordCharacter .. ~~ Except[WordCharacter] ..;
  window = Repeated[wordpattern, {win}];
  resultlist = StringCases[text,
    window ~~ keyword ~~ Except[WordCharacter] .. ~~ window, IgnoreCase -> True];
  Return[resultlist]]
```

We want to add a little more code to format the results so they are easier to read. We break each of our result strings into a list of words using **StringSplit**, then use the **TableForm** and **Style** commands to output the words in columns of medium-sized text. The formatted output is returned in a framed pane with scrollbars.

```
kwic[text_String, keyword_String, win_Integer] :=
 Module[{wordpattern, window, resultlist, formatted},
  wordpattern = WordCharacter .. ~~ Except[WordCharacter] ..;
  window = Repeated[wordpattern, {win}];
  resultlist = StringCases[text,
    window ~~ keyword ~~ Except[WordCharacter] .. ~~ window, IgnoreCase → True];
  formatted = Style[TableForm[StringSplit[resultlist]], Medium];
  Return[Framed[Pane[formatted, {Full, Automatic}, Scrollbars → True]]]]
```

```
kwic[origin20K, "domestic", 3]
```

Here is another example of our function in action. Note that because the **StringSplit** command uses whitespace by default to separate words, the hyphens and dashes screw up our formatting a little bit.

```
kwic[origin20K, "species", 3]
```

A problem with overlapping windows

Beware of increasing the window size too much when using this function. If the words in the window to the right of the first instance of a keyword overlap with those in the window to the left of a second instance, **StringCases** will miss the second instance. If you increase the window size to 4, and do the search for “domestic”, the “domestic pigeons” instance will drop out because of this overlap, as illustrated below.

```
kwic[origin20K, "domestic", 4]
```

This is the passage that has the overlapping KWIC windows when the window size is four

```
StringTake[origin20K, {10 390, 10 497}]
```

The “one or” on the right edge of the first window overlaps with the “one or” on the left edge of the second window.

```
With[{t = TextWords[StringTake[origin20K, {10 390, 10 497}]]},
 {t[[1 ;; 9]], t[[8 ;; 16]]}]
```

Further Exploration

A function for finding one word near another

In the ‘Generalizing the Examples’ section above, we used a function that can find two words near one another in a text string. Here we work through the logic of solving that problem.

We start by using **StringPosition** to find locations of each word in the text string. It will return the starting and ending character positions of all matches in order from the beginning of the text to the

end. We'll use *origin20K* for our text string.

```
StringPosition[origin20K, "geological" ~~ WordBoundary]
```

```
StringPosition[origin20K, "geographical" ~~ WordBoundary]
```

In both cases we only need to keep the starting character positions, so we use **Part** to pull them out.

```
StringPosition[origin20K, "geological" ~~ WordBoundary][[All, 1]]
```

```
StringPosition[origin20K, "geographical" ~~ WordBoundary][[All, 1]]
```

If we merge the two location lists together, we only need to see if adjacent words are within our specified distance, and we only need to check if the two words are different from one another. That is to say that we don't care if one instance of 'geological' is closely followed by another, only if an instance of 'geological' is followed by one of 'geographical', or vice versa. So we pair each word with its locations before merging, like shown below. (The use of pure functions with the **Map** command was introduced in the 'Programming with *Mathematica*' section of Chapter 1).

```
Map[List["geological", #] &,
 StringPosition[origin20K, "geological" ~~ WordBoundary][[All, 1]]]
```

```
Map[List["geographical", #] &,
 StringPosition[origin20K, "geographical" ~~ WordBoundary][[All, 1]]]
```

Join the two lists together and **Sort** on the second member of each pair. (Sorting is explained in more detail in the 'Programming with *Mathematica'* section above.)

```
Sort[Join[Map[List["geological", #] &,
    StringPosition[origin20K, "geological" ~~ WordBoundary][[All, 1]]],
  Map[List["geographical", #] &, StringPosition[origin20K,
     "geographical" ~~ WordBoundary][[All, 1]]]], #1[[2]] < #2[[2]] &]
```

The **Partition** command lets us group adjacent elements of the list into pairs to check.

```
Partition[{a, b, c, d}, 2, 1]
```

```
Partition[Sort[Join[Map[List["geological", #] &,
     StringPosition[origin20K, "geological" ~~ WordBoundary][[All, 1]]],
   Map[List["geographical", #] &, StringPosition[origin20K,
      "geographical" ~~ WordBoundary][[All, 1]]]], #1[[2]] < #2[[2]] &], 2, 1]
```

Now we can use **DeleteCases** to get rid of any pairs where the two words are the same.

```
DeleteCases[
 Partition[Sort[Join[Map[List["geological", #] &, StringPosition[origin20K,
       "geological" ~~ WordBoundary][[All, 1]]], Map[List["geographical", #] &,
     StringPosition[origin20K, "geographical" ~~ WordBoundary][[All, 1]]],
   #1[[2]] < #2[[2]] &], 2, 1], {{x_, _}, {x_, _}}]
```

We are finished now with the word labels, so we can get rid of them with a mix of **Part** and **Span** commands.

```
DeleteCases[
  Partition[Sort[Join[Map[List["geological", #] &, StringPosition[origin20K,
        "geological" ~~ WordBoundary][[All, 1]]], Map[List["geographical", #] &,
      StringPosition[origin20K, "geographical" ~~ WordBoundary][[All, 1]]],
    #1[[2]] < #2[[2]] &], 2, 1], {{x_, _}, {x_, _}}][[All, 1 ;; 2, 2]]
```

We want to **Select** any pair of positions that are within some distance (here we use 100 characters).

```
Select[DeleteCases[
   Partition[Sort[Join[Map[List["geological", #] &, StringPosition[origin20K,
         "geological" ~~ WordBoundary][[All, 1]], Map[List["geographical", #] &,
       StringPosition[origin20K, "geographical" ~~ WordBoundary][[All, 1]]],
     #1[[2]] < #2[[2]] &], 2, 1], {{x_, _}, {x_, _}}][[
  All, 1 ;; 2, 2]], (#[[2]] - #[[1]]) < 100 &]
```

Finally, we can display the matching parts of the text by mapping the **StringTake** function across the character position pairs that are within the desired distance. We show 100 characters before the beginning of the first match and after the end of the second.

```
Map[StringTake[origin20K, {#[[1]] - 100, #[[2]] + 100}] &, {{3526, 3553}, {8420, 8512}}]
```

Now we can bundle everything up into a function. The inputs will be the text string we are searching, the two strings we are searching for, and the number of characters that we want to use as a window. The output will be a **TabView** display.

```
stringFindNear[txt_, str1_, str2_, within_] :=
 TabView[Map[StringTake[txt, {#[[1]] - 100, #[[2]] + 100}] &,
   Select[DeleteCases[Partition[Sort[Join[Map[List[str1, #] &, StringPosition[
            txt, str1 ~~ WordBoundary][[All, 1]], Map[List[str2, #] &,
          StringPosition[txt, str2 ~~ WordBoundary][[All, 1]]], #1[[2]] < #2[[2]] &],
       2, 1], {{x_, _}, {x_, _}}][[All, 1 ;; 2, 2]], (#[[2]] - #[[1]]) ≤ within &]]]

stringFindNear[origin20K, "geological", "geographical", 100]
```

Mathematica Commands to Review

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration
- Blank (BE)
- BlankNullSequence (BE)
- BlankSequence (GE)
- Cases (BE)
- Clear (GE)
- DeleteCases (FE)
- DigitCharacter (GE)
- ExampleData (BE)
- Except (BE)
- Flatten (BE)
- Framed (PM)
- Head (BE)
- Join (FE)
- Keys (BE)
- Length (BE)
- List (FE)
- Lookup (BE)
- Map (BE)
- MatchQ (PM)

- MemberQ (BE)
- Module (PM)
- Pane (PM)
- Part (BE)
- Partition (FE)
- PatternTest (BE)
- Repeated (BE)
- Rest (BE)
- Select (BE)
- Short (BE)
- Shortest (BE)
- Sort (FE)
- StringCases (BE)
- StringExpression (BE)
- StringJoin (FE)
- StringLength (BE)
- StringPosition (FE)
- StringRiffle (BE)
- StringSplit (PM)
- StringTake (BE)
- Style (PM)
- TableForm (BE)
- TabView (GE)
- TextSentences (BE)
- TextWords (BE)
- Union (BE)
- UpperCaseQ (BE)
- Whitespace (BE)
- With (BE)
- WordBoundary (BE)
- WordCharacter (BE)
- WordCounts (BE)
- WordData (BE)
- WordStem (BE)

Problems

```
Problem 2.1
Input: {g,a,g,a,d,a,d,a}
Output: {g,g,d,d}
```

```
Cases[{g, a, g, a, d, a, d, a}, Except[a]]
```

```
Problem 2.2
Input: {{a,1},{b,1},{c,2},{d,2},{e,1}}
Output: {{a,1},{b,1},{e,1}}
```

```
Cases[{{a, 1}, {b, 1}, {c, 2}, {d, 2}, {e, 1}}, {_, 1}]
```

```
Problem 2.3
Input: {{a,1},{b,1},{c,2},{d,2},{e,1}}
Output: {a,b,e}
```

```
Cases[{{a, 1}, {b, 1}, {c, 2}, {d, 2}, {e, 1}}, {x_, 1} → x]

Cases[{{a, 1}, {b, 1}, {c, 2}, {d, 2}, {e, 1}}, {_, 1}][[All, 1]]
```

```
Problem 2.4
Input: {{a,1},{b,1},{c,2},{d,2},{e,1}}
Output: {1,2,a,b,c,d,e}
```

```
Union[Flatten[{{a, 1}, {b, 1}, {c, 2}, {d, 2}, {e, 1}}]]
```

```
Problem 2.5
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: {QUICK,LAZY}
```

```
Select[TextWords["The QUICK brown fox jumped over the LAZY dog"], UpperCaseQ]
```

```
Problem 2.6
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: ⟨|the→2,quick→1,over→1,lazy→1,jumped→1,fox→1,dog→1,brown→1|⟩
```

```
WordCounts["The QUICK brown fox jumped over the LAZY dog", IgnoreCase → True]
```

```
Problem 2.7
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: ⟨|{the,quick}→1,{the,lazy}→1,{quick,brown}→1,{over,the}→1,{lazy,dog}→
1,{jumped,over}→1,{fox,jumped}→1,{brown,fox}→1|⟩
```

```
WordCounts["The QUICK brown fox jumped over the LAZY dog", 2, IgnoreCase → True]
```

```
Problem 2.8
Input: {{a,b,c},{d,e,f},{g,h,i}}
Output: {{b,c},{e,f},{h,i}}
```

```
Map[Rest, {{a, b, c}, {d, e, f}, {g, h, i}}]

{{a, b, c}, {d, e, f}, {g, h, i}}[[All, 2 ;; 3]]
```

```
Problem 2.9
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: {fox,dog}
```

```
StringCases["The QUICK brown fox jumped over the LAZY dog",
 WordBoundary ~~ WordCharacter ~~ "o" ~~ WordCharacter ..]
```

```
Problem 2.10
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: {The,QUI,bro,fox,jum,ove,the,LAZ,dog}
```

```
StringCases["The QUICK brown fox jumped over the LAZY dog",
 WordBoundary ~~ Repeated[WordCharacter, {3}]]
```

```
Map[StringTake[#, 3] &,
 TextWords["The QUICK brown fox jumped over the LAZY dog"]]
```

```
Problem 2.11
Input: "quick"
Output: {Noun,Adjective,Adverb,Interjection}
```

```
WordData["quick", "PartsOfSpeech"]
```

```
Problem 2.12
Input: "quick"
Output: {Agile,Warm,Prompt,Speedy,Fast,Ready}
```

```
Cases[WordData["quick"], {_, "Adjective", x_} → x]
```

```
Problem 2.13
Input: "The QUICK brown fox jumped over the LAZY dog"
Output: {quick,brown,fox,jumped,lazy,dog}
```

```
DeleteStopwords[
 TextWords[ToLowerCase["The QUICK brown fox jumped over the LAZY dog"]]]
```

```
Problem 2.14
Input: {a,b,123,c,45,d,e,67}
Output: {123,45,67}
```

```
Cases[{a, b, 123, c, 45, d, e, 67}, _Integer]
```

```
Problem 2.15
Input: {a,b,123,c,45,d,e,67}
Output: {a,b,c,d,e}
```

```
Cases[{a, b, 123, c, 45, d, e, 67}, _Symbol]
```

```
Problem 2.16
Input: {1,2,3,4,5}
Output: {{1,2,3},{2,3,4},{3,4,5}}
```

```
Partition[{1, 2, 3, 4, 5}, 3, 1]
```

```
Problem 2.17
Input: {1,2,3,4,5}
Output: {{1},{2},{3},{4},{5}}
```

```
Partition[{1, 2, 3, 4, 5}, 1, 1]

Map[List, {1, 2, 3, 4, 5}]
```

Exercises

1. *(Gendered language)*. Choose one of the sample texts and do an analysis of the author's use of gendered language. One of the easiest ways to start is to look at the contexts in which he or she uses pronouns (like "he" or "she"). For example, do the same kinds of verbs follow both pronouns? Is the reader assumed to be masculine? You can use the **LanguageData** command to get a list of basic pronouns for a particular language as shown in the command below. You will probably also want to consider possessives such as "his" and "hers".

```
LanguageData["English", "Pronouns"]
```

2. (*Trigrams*). Try analyzing the 3-grams in the Introduction to *Origin of Species*. What kinds of patterns become more evident with longer phrases? What tradeoffs do you encounter as you increase *n* in your n-gram analyses?
3. *(Roll your own n-gram analysis)*. Prior to the introduction of the **WordCounts** command in 2015, *Mathematica* programmers had to use different methods to extract and count n-grams. Using **StringSplit**, **Sort**, **Partition** and **Tally**, write your own n-gram frequency code. Compare your results to those output by **WordCounts**.
4. *(Visualizing workflow)*. In the 'Further Exploration' section of Chapter 1, there is a step-by-step description of using **LayeredGraphPlot** to visualize your workflow. Create a similar image for the 'Basic Examples' section of Chapter 2. What are the advantages and disadvantages of including more or less detail in your network visualization?

ch03

Chapter 03: Who and What

Overview

In the first two chapters, we began to learn how to use the statistical analysis of text to identify important words and phrases in a document or corpus of documents. We also started working with commands that allow us to describe and match *patterns*: generalizations that apply to more than one word or phrase. And in the case of the **WordData** command we saw that *Mathematica* has access to a lot of information about the English language, although we only scratched the surface of what that command can do. In fact, one of the main advantages of using *Mathematica* for digital research is that it has direct, computational access to information about millions of entities. In this chapter and the next, we will explore some of this *computable data* and see how it allows us to rapidly prototype tools that can be customized to our research sources and questions.

N.B. Because sources of computable data are constantly being updated, the results that I obtained when I wrote the text could well be different from the ones that you get when you are using these commands. Keep that in mind if your results are a bit different from mine.

Basic Examples

Computable data about people

In exploring the meaning of a text, *named entities* (such as people, organizations, locations, dates and times) play a special role, and being able to automatically identify these entities allows you to link data from multiple sources. We begin with the problem of identifying people and retrieving information about them.

Mathematica has direct access to information about millions of entities via the **WolframAlpha** command. If we want to search by hand, we can type CTRL= in our Notebook, and then type our query into the text box. Let's do this for Darwin himself. (This command, and many others we use in this chapter, require Internet access).

Charles Darwin (person) ••• ✓

Mathematica responds by giving us an input box where we can either accept the default interpretation (which happens to be correct) or choose an alternate interpretation (if it is not). The formatting of the output shows us that "Charles Darwin" is an *entity*, a symbolic representation that is meaningfully attached to many other kinds of information. Looking at the input box, we can also see that this entity is of the type "Person".

We can get the same functionality by using the **WolframAlpha** command and asking for the "WolframResult" format.

```
WolframAlpha["Darwin", "WolframResult"]
```

Another way to retrieve an entity is to use an **Interpreter**. This family of commands generates entites of a desired type. Here is how we generate an entity corresponding to Darwin.

```
Interpreter["ComputedPerson"]["Charles Darwin"]
```

Once we discover that there is an entity corresponding to one of our words or phrases, we can ask what kind of entity it is with the **EntityTypeName** command.

```
EntityTypeName[ Charles Darwin (person) ]
```

Once we have identified an entity, we can request other information, or *Properties*, of that entity. Here is how we retrieve Darwin's birth and death dates. The formatting of the output shows that each of these commands is returning other kinds of entites (locations in this case).

```
PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "BirthPlace"]
```

```
PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "DeathPlace"]
```

Computable data is returned in a form that allows us to use it for computation. How far away from his birthplace did Darwin die?

```
GeoDistance[ Shrewsbury (city) , London (city) ]
```

We can use the **PersonData** command with "Properties" to find out what kind of information is available for that person. Some of the categories of information probably won't be useful in a case like this (e.g., Chinese zodiac sign) and some of it won't apply at all (e.g., space missions, notable film direction credits). But some of the information that we can request will definitely be useful.

```
PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "Properties"]
```

If we just have part of a name, we can retrieve a full name.

```
PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "FullName"]
```

The following command retrieves Darwin's birth date. Note that the returned value is also an entity. In the next chapter we will make more use of entities representing dates and places.

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "BirthDate"]
```

We can request a picture of the person if it is available.

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "Image"]
```

And a very short statement of their importance.

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "NotableFacts"]
```

Other books he wrote.

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "NotableBooks"] // TableForm
```

Some of the information that we might be curious about may not be available from the **PersonData** command.

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "Weight"]
```

Storing information in an association

Each time that we request external information, it takes a while to contact the server and retrieve the results. So it is a good idea to store a local copy of any information that we retrieve, so that we don't

have to keep asking for it. There are a number of different ways that we might do this, all involving what are called *data structures*. Some examples of data structures in *Mathematica* that we have already seen are the string, list and association. We will be using an association for this task.

A given person, like Darwin, might be associated with a lot of different kinds of information: birth date, birth place, death date, place of death, spouse, children, etc. It is straightforward to imagine keeping all of this information in an association. In fact, a version of the **PersonData** command allows us to create such an association in one step, as shown below.

```
darwinAssoc = PersonData[Interpreter["ComputedPerson"]["Darwin"],
  {"Entity", "FullName", "Image", "BirthDate", "DeathDate", "NotableFacts"},
  "PropertyAssociation"]
```

Our association consists of a collection of rules of the form *key→value*. We can request the value associated with a particular **Key** by using a notation similar to **Part** :

```
darwinAssoc[[Key[EntityProperty["Person", "Entity"]]]]
```

```
darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]]
```

We are going to want to retrieve **PersonData** a number of times, so we create a function for it.

```
getWAPersonPropertyAssoc[person_] :=
 PersonData[Interpreter["ComputedPerson"][person], {"Entity", "FullName",
   "Image", "BirthDate", "DeathDate", "NotableFacts"}, "PropertyAssociation"]
```

Formatting retrieved information for display

Later in the chapter we will see that it is relatively easy to go through a text, automatically identify some of the people mentioned in it, and retrieve information about them. In a traditional research workflow, it was common to write notes on 3x5 inch index cards. We can lay out the information in our association as if it were written on such a card.

Column displays a list in a column.

```
Column[{a, b, c}]
```

Here we use **Column** to put some information in a column.

```
Column[{darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]],
  darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]],
  darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]}]
```

We use the **DateString** command to turn date entities into strings. The single at sign (@) is infix notation for **Apply**. It lets us apply one command to another without building up a lot of nested brackets.

```
Column[{darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]],
  DateString@darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]],
  DateString@darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]}]
```

The **Style** and **Text** commands change formatting.

```
Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold]
```

```
Text@Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold]
```

When we put them in our column of information it looks like this. We have joined birth and death dates into a long string on one line using the infix notation for **StringJoin** (<>), and then formatted the whole thing as **Text**.

```
Column[
 {Text@Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold],
  Text["b. " <> DateString@
      darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]] <> ", d. " <>
    DateString@darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]]}]
```

The notable facts are themselves a list of strings, so they go inside another **Column** command. We want the facts to be formatted in a smaller font (**Style** Medium), so we **Map** a formatting command across the list. The slash at-sign (/@) is infix shorthand for the **Map** command.

```
Column[Text[Style[#, Medium]] & /@
  darwinAssoc[[Key[EntityProperty["Person", "NotableFacts"]]]]]
```

If we put all of our formatted information together, it now looks like this.

```
Column[
 {Text@Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold],
  Text["b. " <> DateString@
      darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]] <> ", d. " <>
    DateString@darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]],
  Column[Text[Style[#, Medium]] & /@
    darwinAssoc[[Key[EntityProperty["Person", "NotableFacts"]]]]]}]
```

The **Grid** command displays a nested list as a grid.

```
Grid[{{a, b}, {c, d}}]
```

We can use **Grid** to put our picture beside the column of information. The *Frame→All* option draws boxes around each cell in the grid.

```
Grid[{{darwinAssoc[[Key[EntityProperty["Person", "Image"]]]], Column[
    {Text@Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold],
     Text["b. " <> DateString@
         darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]] <> ", d. " <>
       DateString@darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]],
     Column[Text[Style[#, Medium]] & /@ darwinAssoc[
         Key[EntityProperty["Person", "NotableFacts"]]]]}]}},
 Frame →
  All]
```

Now we use the *ItemSize* scaling option for **Grid** so that twenty-five percent of the width of the notebook goes to the image and sixty-five percent to the column of information. We use the *Alignment* option so that the picture is aligned to the center of its cell, the information is aligned to the left of its cell, and both picture and information are aligned to the tops of their respective cells.

```
Grid[{{darwinAssoc[[Key[EntityProperty["Person", "Image"]]]], Column[
    {Text@Style[darwinAssoc[[Key[EntityProperty["Person", "FullName"]]]], Bold],
     Text["b. " <> DateString@
         darwinAssoc[[Key[EntityProperty["Person", "BirthDate"]]]] <> ", d. " <>
       DateString@darwinAssoc[[Key[EntityProperty["Person", "DeathDate"]]]]],
     Column[Text[Style[#, Medium]] & /@ darwinAssoc[
         Key[EntityProperty["Person", "NotableFacts"]]]]}]}},
 Frame → All, ItemSize → {{Scaled[.25], Scaled[.65]}},
 Alignment → {{Center, Left}, {Top, Top}}]
```

We can wrap this up into a little function.

```
formatWAPersonData[assoc_] :=
 Grid[{{assoc⟦Key[EntityProperty["Person", "Image"]]⟧,
    Column[{Text@Style[assoc⟦Key[EntityProperty["Person", "FullName"]]⟧, Bold],
      Text["b. " <> DateString@
          assoc⟦Key[EntityProperty["Person", "BirthDate"]]⟧ <> ", d. " <>
        DateString@assoc⟦Key[EntityProperty["Person", "DeathDate"]]⟧],
      Column[Text[Style[#, Medium]] & /@
        assoc⟦Key[EntityProperty["Person", "NotableFacts"]]⟧]}]},
  Frame → All, ItemSize → {{Scaled[.25], Scaled[.65]}},
  Alignment → {{Center, Left}, {Top, Top}}]

formatWAPersonData[darwinAssoc]
```

An association of associations

Since we will be automatically retrieving information for people and other named entities mentioned in a text, we don't want to have a separate data structure for each entity. We actually need to create a *nested* association, an association of associations. For the keys at the outside level, we are going to use a string that denotes the entity. For people, we will use the string *Lastname, Firstname*. Each of those keys will be linked to an association that keeps track of the information for that person.

```
<|"Darwin, Charles" ->
   ⟨|EntityProperty[Person, Entity] → Charles Darwin, ..., EntityProperty[Person,
BirthDate] -> "Sun 12 Feb 1809", ...|>,
  "Lyell, Charles" ->
   ⟨|EntityProperty[Person, Entity] → Charles Lyell, ..., EntityProperty[Person,
BirthDate] -> "Tue 14 Nov 1797", ...|>,
  ...
 |>
```

Let's start building our association by creating an entry for Darwin.

```
darwinPeopleAssoc =
 Association[{"Darwin, Charles" → getWAPersonPropertyAssoc["Charles Darwin"]}]
```

Next we can add an entry for Charles Lyell, who is mentioned a number of times in *Origin*.

```
textSearch[origin, "Charles Lyell"]
```

First we make sure that *Wolfram Alpha* has information about Lyell.

```
Interpreter["ComputedPerson"]["Charles Lyell"]
```

Then we add him to our association of associations with the **AssociateTo** command, which adds a rule to an existing association.

```
AssociateTo[darwinPeopleAssoc,
  "Lyell, Charles" → getWAPersonPropertyAssoc["Charles Lyell"]];
```

We can view the formatted data for Charles Lyell with the following command.

```
formatWAPersonData[darwinPeopleAssoc⟦"Lyell, Charles"⟧]
```

Another person mentioned in *Origin* is George Bentham. Let's add him to *darwinPeopleAssoc*, too.

```
textSearch[origin, "Bentham"]
```

```
Interpreter["ComputedPerson"]["George Bentham"]
```

```
AssociateTo[darwinPeopleAssoc,
  "Bentham, George" → getWAPersonPropertyAssoc["George Bentham"]];
```

```
formatWAPersonData[darwinPeopleAssoc⟦"Bentham, George"⟧]
```

Who's Who: A stack of cards

If we imagine the formatted display of information as something akin to a traditional file card, then we will want to have some way of flipping through them. The command that we are going to use for this is called **MenuView**. It is similar to **TabView**, but instead of clicking buttons to change panes, we change them with a pull-down menu.

Let's begin with the **Table** command, which makes a structured list. Here we make a list with four elements. Each is a **Rule** which connects a number to the string "something".

```
Table[i → "something", {i, 3}]
```

If we modify the command a little bit, we can get it to create a list of rules where the left hand side of each is a person in our association of associations.

```
Keys[darwinPeopleAssoc]
```

```
Keys[darwinPeopleAssoc][[2]]
```

```
Table[Keys[darwinPeopleAssoc][[i]] → "something", {i, 3}]
```

Let's sort the list so it is alphabetical by last name.

```
Sort[Table[Keys[darwinPeopleAssoc][[i]] → "something", {i, 3}]]
```

The number 3 here is just the number of elements in our list of keys. We can use **Length** to get that information. Then when we add more people, our command will still work. When we wrap the output in the **MenuView** command, we get our basic interface. The final display has to fit nicely in the notebook, so we use the *ImageSize → Automatic* property for **MenuView**.

```
MenuView[Sort[Table[Keys[darwinPeopleAssoc][[i]] → "something",
    {i, Length[Keys[darwinPeopleAssoc]]}]], ImageSize → Automatic]
```

Now that is working, we replace "something" with the nicely formatted display of information about the person, then put the whole thing in a function.

```
whosWho[assoc_] :=
 MenuView[
  Sort[Table[Keys[assoc][[i]] → formatWAPersonData[assoc[[Keys[assoc][[i]]]],
    {i, Length[Keys[assoc]]}]], ImageSize → Automatic]

whosWho[darwinPeopleAssoc]
```

Linking people to passages

Let's make one more refinement to the idea of automatically generating a *Who's Who* of people mentioned in a text. It would be nice to include a browser that shows us passages in the text where the person in question was mentioned. We do this by adapting our *textSearch* code from Chapter 2. Instead of using the **TabView** command, however, we are going to use a command called **SlideView**, which lets us navigate through panes as if we were clicking through a slide show. We want the pane to take up the same amount of space as our formatted display (which was 90% of the notebook width) so we specify that with the *ImageSize→Scaled[0.9]* property.

```
SlideView[Map[StringTake[origin, {#[[1]] - 100, #[[2]] + 100}] &,
  StringPosition[origin, "Bentham" ~~ WordBoundary]], ImageSize → Scaled[0.9]]
```

Setting a few other properties allows us to move the controls to the bottom of the interface, to add some indication of how many slides there are, and to show which slide is currently being displayed. In addition, we format as **Text**.

```
Text@SlideView[Map[StringTake[origin, {#[[1]] - 100, #[[2]] + 100}] &,
    StringPosition[origin, "Bentham" ~~ WordBoundary]], ImageSize → Scaled[0.9],
  ControlPlacement → Bottom, AppearanceElements → {"FirstSlide",
    "PreviousSlide", "NextSlide", "LastSlide", "SlideNumber", "SlideTotal"}]
```

We can't be sure that there will be a match, so we want our **SlideView** to do something graceful if not. Here is how it behaves without any error checking.

```
Text@SlideView[Map[StringTake[origin, {#[[1]] - 100, #[[2]] + 100}] &,
    StringPosition[origin, "Darwin" ~~ WordBoundary]], ImageSize → Scaled[0.9],
  ControlPlacement → Bottom, AppearanceElements → {"FirstSlide",
    "PreviousSlide", "NextSlide", "LastSlide", "SlideNumber", "SlideTotal"}]
```

So we add a **With** command that tries to do the matching first, and an **If** command that displays an error message if the text search returns no matches. We also change the size from 100 to 200 characters on either side of the match.

```
Text@SlideView[With[{srch = Map[StringTake[origin, {#[[1]] - 200, #[[2]] + 200}] &,
      StringPosition[origin, "Darwin" ~~ WordBoundary]]},
   If[srch ≠ {}, srch, {Style["No match found", Italic]}]],
  ImageSize → Scaled[0.9], ControlPlacement → Bottom,
  AppearanceElements → {"FirstSlide", "PreviousSlide",
    "NextSlide", "LastSlide", "SlideNumber", "SlideTotal"}]
```

The last thing we need to do is get the search term. Recall that our keys are listed as "Lastname, Firstname". If we use **StringSplit**, we can pull out the first word, like this:

```
Keys[darwinPeopleAssoc][[3]]
```

```
StringSplit[Keys[darwinPeopleAssoc][[3]], Except[WordCharacter]][[1]]
```

Here is a new version of the *whosWho* function that shows text matches for each person.

```
whosWho2[assoc_, txt_] :=
 MenuView[Sort[Table[Keys[assoc][[i]] → Column[{
        formatWAPersonData[assoc[[Keys[assoc][[i]]]]],
        Text@
         SlideView[With[{srch = Map[StringTake[txt, {#[[1]] - 200, #[[2]] + 200}] &,
               StringPosition[txt, StringSplit[Keys[assoc][[i]],
                    Except[WordCharacter]][[1]] ~~ WordBoundary]]},
           If[srch ≠ {}, srch, {Style["No match found", Italic]}]],
          ImageSize → Scaled[0.9], ControlPlacement → Bottom,
          AppearanceElements → {"FirstSlide", "PreviousSlide",
            "NextSlide", "LastSlide", "SlideNumber", "SlideTotal"}]}],
    {i, Length[Keys[assoc]]}]], ImageSize → Automatic]
```

```
whosWho2[darwinPeopleAssoc, origin]
```

Summary

The deep integration of computable data into *Mathematica* makes it possible to resolve named entities of many different kinds and automatically retrieve information about them. This, in turn, means that you can write computational tools that do some kinds of simple research tasks that you (or maybe a research assistant) used to do by hand. (Who was George Bentham? Was his *Handbook of British Flora* published before *Origin*? Where in the text of *Origin* does Darwin mention him?)

So far we have focused on people, but in the rest of this chapter and in the next we will also see examples of named institutions, books, places, dates, ships, and other entities. Because each *Wolfram Alpha* or *Wikipedia* query takes a significant amount of time, we *cached* data that we retrieved in an association, so that future requests for it would be much faster.

Generalizing the Examples

Identifying probable names

Anything that we can do once or twice, we can do over and over. In Chapter 2 we developed methods for finding capitalized words that were not at the beginnings of sentences. Now we will pull out all of the capitalized words from *Origin of Species*, try to figure out which ones are probably names, see which of those we can identify, and then retrieve computable information for each.

```
capitalizedWords[textstr_] :=
 Union[Select[
   Flatten[Map[Rest, Map[TextWords, TextSentences[textstr]]]], capitalizedQ]]
```

```
originCapWords = capitalizedWords[origin];
```

```
viewData[originCapWords]
```

Looking through these capitalized words, we see some that we know (or suspect) are people's names, and others that we know (or suspect) are not. We can start by removing any word from the list if it also appears in the text in lowercase. This should get rid of things like 'absence' and 'abstract'. (It may also get rid of names like "Little", "Brown", "Gray", "Smith", "Banks", "Burns" and so on. It is usually a good idea to study your sources carefully before and after each operation, to make sure you aren't losing critical information.)

This is a list of each of the words in the whole book.

```
originTerms = Union[TextWords[origin]];
```

Each word in our list of capitalized words is in *originTerms*.

```
MemberQ[originTerms, "Abstract"]
```

We can test to see if it also appears in the text in lowercase like this.

```
MemberQ[originTerms, ToLowerCase["abstract"]]
```

Here are all the words we will remove if we pull out ones that appear both capitalized and in lowercase. We look through the list to make sure we're not getting rid of any probable names. (There are a few, like Brown and Gray, but we won't worry about this right now).

```
viewData[Select[originCapWords, MemberQ[originTerms, ToLowerCase[#]] &]]
```

We will use the **Complement** command to find the ones that remain. This list is much shorter but still contains many words that are not proper names.

```
Complement[{a, b, c, d, e}, {a, b}]
```

```
viewData[Complement[originCapWords,
  Select[originCapWords, MemberQ[originTerms, ToLowerCase[#]] &]]]
```

We will use the **WordData** command to try to narrow down the list. It recognizes some capitalized words as people's names.

```
WordData["Agassiz"]
```

If we request the "BroaderTerms" property, it will give us a career description. The first time you use this functionality, *Mathematica* contacts Wolfram servers to download additional information. Unlike the **WolframAlpha** or **WikipediaSearch** commands, however, subsequent calls are quite fast.

```
WordData["Agassiz", "BroaderTerms"]
```

And if we ask for "BroaderTerms" for those terms, we eventually hit a description that includes the word "person".

```
WordData["naturalist", "BroaderTerms"]
```

```
WordData["biologist", "BroaderTerms"]
```

```
WordData["scientist", "BroaderTerms"]
```

Some of the words that we are trying to filter out of this list are both nouns that refer to people and adjectives (e.g., 'the American', 'American scientist'). A proper name is a noun but not an adjective. Let's create small functions that checks to see if a word is a noun or an adjective.

```
nounQ[str_] :=
 Cases[WordData[str], {_, "Noun", ___}] ≠ {}
```

```
nounQ["scientist"]
```

```
nounQ["scientific"]
```

```
adjectiveQ[str_] :=
 Cases[WordData[str], {_, "Adjective", ___}] ≠ {}
```

```
adjectiveQ["scientist"]
```

```
adjectiveQ["scientific"]
```

The function below will start with a word that is a noun and not an adjective, and follow broader terms for three jumps to see if it hits "person". The details of how it works are explained in the 'Programming with *Mathematica*' section below.

```
personTermQ[str_] :=
 If[nounQ[str] && Not[adjectiveQ[str]], MemberQ[
   Union[Flatten[NestList[Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &,
       {str}, 3]]], "person"], False]
```

```
personTermQ["scientist"]
```

```
personTermQ["biologist"]
```

```
personTermQ["naturalist"]
```

```
personTermQ["Agassiz"]
```

Now that we have this function, let's test each of the capitalized words with it, and keep the ones that seem to refer to people. It definitely won't be perfect, because we will lose any capitalized surnames that don't appear in the **WordData** dictionary. And it won't manage to get rid of some of the place names or other noise.

```
originCapWordsPerson =
  Select[Complement[originCapWords, Union[CharacterRange["A", "Z"], Select[
      originCapWords, MemberQ[originTerms, ToLowerCase[#]] &]]], personTermQ];
```

```
originCapWordsPerson
```

This list is short enough that we can test each entry with **WolframAlpha** and return any entities are associated with a person.

```
getWAPerson[str_] :=
 Module[{entity},
  entity = WolframAlpha[str, "WolframResult"];
  If[EntityTypeName[entity] == "Person", Return[entity]]]
```

```
getWAPerson["Agassiz"]
```

```
getWAPerson["America"]
```

Now we can **Map** this function across the list of capitalized words we want to test, and collect any matching person entites. (While I was revising this text, the WolframAlpha server returned a number

of unrelated warning messages. By wrapping the **Cases** command in **Quiet**, I am telling *Mathematica* to suppress warnings just for this example.)

```
originCapWordsPersonEntities =
 Quiet[Cases[Map[getWAPerson, originCapWordsPerson], _Entity]]
```

The wrong Wallace

When we look through the list above, we can see that we've got the wrong "Wallace". (Joke about the wrong trousers left as an exercise for the reader). We've also obviously got the wrong King and, not so obviously, the wrong Bentham.

In a case like this, we might try to make use of **WordData** and/or **WikipediaSearch** to try to find the correct entities. The former is faster than the latter, so we should always start with it.

```
WordData["Wallace", "Definitions"]
```

```
WikipediaSearch["Content" → "Wallace naturalist 1823 1913", "MaxItems" → 3]
```

We make sure that Wolfram Alpha has an identity for the correct Wallace.

```
Interpreter["ComputedPerson"]["Alfred Russel Wallace"]
```

Now we can use the **ReplacePart** command to replace the wrong Wallace with the correct one.

```
originCapWordsPersonEntities = ReplacePart[originCapWordsPersonEntities,
  -1 -> Interpreter["ComputedPerson"]["Alfred Russel Wallace"]]
```

Next we try to figure out which "King" Darwin was referring to.

```
textSearch[origin, "King"]
```

So we can safely delete MLK from our list of person entities.

```
originCapWordsPersonEntities = Delete[originCapWordsPersonEntities, 4]
```

Finally we can delete this Bentham, too. We already added the correct Bentham, George, above.

```
originCapWordsPersonEntities = Delete[originCapWordsPersonEntities, 2]
```

Let's add information about these six people to our association of associations and then rerun the *whosWho2* function.

```
AssociateTo[darwinPeopleAssoc,
  "Agassiz, Louis" → getWAPersonPropertyAssoc["Louis Agassiz"]];
```

```
AssociateTo[darwinPeopleAssoc,
  "Cuvier, Georges" → getWAPersonPropertyAssoc["Georges Cuvier"]];
```

```
AssociateTo[darwinPeopleAssoc, "Lamarck, Jean-Baptiste" →
   getWAPersonPropertyAssoc["Jean-Baptiste Lamarck"]];
```

```
AssociateTo[darwinPeopleAssoc,
  "Magellan, Ferdinand" → getWAPersonPropertyAssoc["Ferdinand Magellan"]];
```

```
AssociateTo[darwinPeopleAssoc,
  "Pliny the Elder" → getWAPersonPropertyAssoc["Pliny the Elder"]];
```

```
AssociateTo[darwinPeopleAssoc, "Wallace, Alfred Russel" →
   getWAPersonPropertyAssoc["Alfred Russel Wallace"]];
```

```
whosWho2[darwinPeopleAssoc, origin]
```

Working with capitalized bigrams

In Chapter 2 we also developed methods for finding capitalized bigrams, and some of these will be

names that we missed. Following a similar process to the one we used for capitalized words, we will see if we can find any person entities among the capitalized bigrams.

```
capitalizedBigrams[txtstr_] :=
 Cases[Keys[WordCounts[
    StringRiffle[Flatten[Map[Rest, Map[TextWords, TextSentences[txtstr]]]]],
    2]], {_?capitalizedQ, _?capitalizedQ}]
```

```
originCapBigrams = capitalizedBigrams[origin];
```

```
viewData[originCapBigrams]
```

Looking through these capitalized bigrams, we see some that we know (or suspect) are people's names, and others that we know (or suspect) are not. We can start by removing any bigram from the list if both words also appear in the text in lowercase. This should get rid of things like {THE, SAME} and {Geological, Record}.

Here is how we test both words at the same time. The double ampersand (&&) is infix notation for **And**.

```
MemberQ[originTerms, ToLowerCase["Geological"]] &&
 MemberQ[originTerms, ToLowerCase["Record"]]
```

Here are all the bigrams that we would remove if we pulled out the ones that appear both capitalized and in lowercase. We look through the list to make sure we're not getting rid of any probable names.

```
viewData[Select[originCapBigrams, MemberQ[originTerms, ToLowerCase[#[[1]]] &&
    MemberQ[originTerms, ToLowerCase[#[[2]]] &]]
```

As we can see, none of them are names. Again we will use the **Complement** command to find the ones that remain. This list is much shorter but still contains a lot of bigrams that don't seem to refer to people.

```
viewData[Complement[originCapBigrams,
  Select[originCapBigrams, MemberQ[originTerms, ToLowerCase[#[[1]]] &&
     MemberQ[originTerms, ToLowerCase[#[[2]]] &]]]
```

Part of the trick to cleaning up a list like this is to remove entities that you can identify as something else. Most of the nouns that appear in **WordData**, for example, are actually places, as you can see in the **Select** statement below.

```
Select[Complement[originCapBigrams,
  Select[originCapBigrams, MemberQ[originTerms, ToLowerCase[#[[1]]] &&
     MemberQ[originTerms, ToLowerCase[#[[2]]] &]], nounQ[StringRiffle[#]] &]
```

```
WordData["Asa Gray", "Definitions"]
```

```
WordData["Robert Brown", "Definitions"]
```

Other names are more easily retrieved by using the **Cases** command to pull out titles.

```
Cases[Complement[originCapBigrams,
  Select[originCapBigrams, MemberQ[originTerms, ToLowerCase[#[[1]]] &&
     MemberQ[originTerms, ToLowerCase[#[[2]]] &]], {"Colonel" | "Professor", _}]
```

```
WikipediaSearch["Content" → "Dana Darwin Origin", "MaxItems" → 2]
```

```
WikipediaSearch["Content" → "Huxley Darwin Origin", "MaxItems" → 2]
```

If we wished, we could continue to track down named person entities from the text and add them to our association of associations, but we will leave it for now. The amount of work that you do in a situation like this depends on your goals. If you are just trying to get the gist of a source quickly, you can leave things messy. If you are writing a dissertation or book about Darwin or the history of natural history, you want to get every detail right. And that, of course, would also involve a critical

reading of the information returned by *Wolfram Alpha*, *Wikipedia*, and other sources of computable data.

The author of *Vestiges*

Sometimes a person isn't mentioned by their full name but is referred to by another phrase instead. To human readers the meaning may be obvious in context. For example, in the passage shown below we can safely conclude that "Sir Charles Lyell" and "Sir C. Lyell" refer to the same person. Getting the computer to recognize that fact is more difficult.

```
stringFindNear[origin, "C", "Charles", 200]
```

In *Origin*, Joseph Dalton Hooker is always referred to as "Dr. Hooker". In this case—as with "Mr. Wallace"—we need to consult evidence outside the text (e.g., *Wikipedia*) to disambiguate the referent.

```
textSearch[origin, "Hooker"]
```

```
WikipediaSearch["Joseph Dalton Hooker"]
```

Sometimes, however, people are referred to by a phrase that doesn't contain a name at all, and the Introduction to *Origin* contains one (in)famous example of this. Darwin refers to "the author of the 'Vestiges of Creation'" because at the time *Origin* was published (1859), the person who wrote *Vestiges* wasn't known. Published anonymously in 1844, the book was a sensation and the list of suspected authors was quite long. In fact Darwin himself was among them (Secord 2003).

```
stringFindNear[origin, "author", "Vestiges", 200]
```

We can use the **WikipediaSearch** and **PersonData** commands to quickly resolve who this phrase refers to.

```
WikipediaSearch["vestiges of creation"]
```

```
PersonData[ Robert Chambers (person) , "FullName"]
```

```
PersonData[ Robert Chambers (person) , "Image"]
```

```
PersonData[ Robert Chambers (person) , "BirthDate"]
```

We will return to *Vestiges of the Natural History of Creation* in a future chapter.

Other kinds of named entity

In the next chapter we will consider events, times and places, but here are a few other examples of named entities that appear in *Origin*. In many cases we can get further information with calls to **WolframAlpha** or **WikipediaSearch**, just as we did in the case of named persons.

Institutions:

```
textSearch[origin, "Linnean Society"]
```

```
WikipediaSearch["Linnean Society of London"]
```

Books:

```
WikipediaSearch["Principles of Geology"]
```

```
EntityValue[ The Voyage of the Beagle (book) , "Properties"]
```

```
EntityValue[ The Voyage of the Beagle (book) , "Author"]
```

```
EntityValue[ The Voyage of the Beagle (book) , "OriginalTitle"]
```

Vessels:

```
WikipediaSearch["H.M.S. Beagle"]
```

Taxonomic categories:

```
textSearch[origin, "Columbidae"]
```

```
WikipediaSearch["Columbidae"]
```

pigeon (species specification)

```
EntityValue[ pigeon (species specification) ,
 {"Kingdom", "Phylum", "Class", "Order", "Family"}]
```

Programming with *Mathematica*

Nondestructive versus destructive operations

In *Mathematica*, operations tend to be *nondestructive* by default. Suppose you assign a list to a symbol.

```
testList = {a, b, c, d, e}
```

If you apply a command like **Reverse**, the output will be a reordering of the list.

```
Reverse[testList]
```

But the original list has not changed. That is what is meant by a nondestructive operation.

```
testList
```

If you want the original list to change, you have to assign a new value to the symbol.

```
testList = Reverse[testList]
```

Now our original list has been replaced with the reversed version. That is a destructive operation.

```
testList
```

We use the **Clear** command to clear the assignment.

```
Clear[testList]
```

In the case of the association that we created to keep track of information retrieved from *Wolfram Alpha*, we need to make destructive changes. The **AssociateTo** command modifies associations whenever we use it.

When we used the **Delete** or **ReplacePart** commands to destructively modify lists, we assigned the new list to the original symbol, as shown below

```
originCapWordsPersonEntities = Delete[originCapWordsPersonEntities, 4]
```

The benefits of reading the *Mathematica* documentation

Often when you are working on a problem you will find that one of the examples in the *Mathematica* documentation comes close to doing what you need. The *personTermQ* function was adapted from some code in the help file for **WordData**.

```
personTermQ[str_] :=
 If[nounQ[str] && Not[adjectiveQ[str]],
```

```
   MemberQ[
     Union[
       Flatten[NestList[Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &,
         {str}, 3]]], "person"], False]
```

```
personTermQ[str_] :=
 If[nounQ[str] && Not[adjectiveQ[str]], MemberQ[
   Union[Flatten[NestList[Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &,
      {str}, 3]]], "person"], False]
```

The key to understanding this code is the **NestList** command, which applies a function to an expression a number of times, collecting the output in a list as it goes. To demonstrate, recall the *threeCopies* function we introduced in Chapter 1, which makes three copies of something.

```
threeCopies[x_] := {x, x, x}
```

```
threeCopies[a]
```

If you use **NestList** with this function and the symbol **a** twice, you get the following result.

```
NestList[threeCopies, a, 2]
```

The first item in the list is just the symbol **a**. It is the result of not applying the function at all. The second item in the list is a list with three copies of the symbol. It is the result of applying the function once. The third item in the list is a list of three copies of the list with three copies of the symbol. If we use **Column** to display the output of **NestList** it is a bit easier to see how it works.

```
NestList[threeCopies, a, 2] // Column
```

Here is what happens if we do it three times. Here we use the **Text** command to fit the fourth item in the list on one line. At each step, the thing that is being copied three times is the line above it.

```
NestList[threeCopies, a, 3] // Column // Text
```

Now let's try this with the **WordData** command. Here is what happens when we call the function once.

```
Union[Flatten[WordData["naturalist", "BroaderTerms", "List"]]]
```

This is the same thing, written as a pure function mapped across a list of terms. Our initial list just contains one term.

```
Union[(Flatten[WordData[#, "BroaderTerms", "List"]] &) /@ {"naturalist"}]
```

Now if we use **NestList** zero times we just have the original term.

```
Union@Flatten[NestList[
   Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &, {"naturalist"}, 0]]
```

If we use it once, we have a list with the original term and the broader terms.

```
Union@Flatten[NestList[
   Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &, {"naturalist"}, 1]]
```

If we use it twice, we get the broader terms of the broader terms added to our list.

```
Union@Flatten[NestList[
   Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &, {"naturalist"}, 2]]
```

If we use it three times, we have travelled quite a distance from the original term. If this list contains the word "person" (it does), then *personTermQ* returns *True*.

```
Union@Flatten[NestList[
   Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &, {"naturalist"}, 3]]
```

Here is an example of a noun that doesn't include "person" among its broader terms, or the broader

terms of those.

```
Union@Flatten[
  NestList[Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &, {"tapir"}, 3]]
```

References and Further Reading

- Secord, James A. *Victorian Sensation: The Extraordinary Publication, Reception, and Secret Authorship of* Vestiges of the Natural History of Creation. Chicago: University of Chicago Press, 2003.

Mathematica Commands to Review

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration
- And (GE)
- Apply (BE)
- AssociateTo (BE)
- Column (BE)
- Complement (GE)
- DateString (BE)
- Entity (GE)
- EntityTypeName (BE)
- EntityValue (GE)
- GeoDistance (BE)
- Grid (BE)
- If (BE)
- Interpreter (BE)
- MenuView (BE)
- NestList (PM)
- PersonData (BE)
- ReplacePart (GE)
- SlideView (BE)
- TabView (BE)
- Text (BE)
- WikipediaSearch (GE)
- With (BE)
- WolframAlpha (BE)

Exercises

1. *(Modifying whosWho)*. Try writing a version of the *whosWho* function that displays a KWIC listing rather than the text search one.

ch04

Chapter 04: When and Where

Overview

In the last chapter, we saw that we can use **WolframAlpha** and related commands like **PersonData** and **EntityValue** to associate strings in our text (like “Charles Lyell”, “Voyage of the Beagle” and “South America”) with *entities* (like Charles Lyell, The Voyage of the Beagle, and South America). Having access to entities opens up a world of meaning, because we can retrieve properties associated with each entity, and use those both for computation and for further querying. In this chapter, we continue our exploration of entities, focusing now on times and places.

Basic Examples

Historical periods and events

One of the most basic ways to contextualize a research project is to situate it with respect to time and place. Let’s begin with time. *Mathematica* has access to entities that represent both broad historical periods and particular events. We can find out how many historical periods have associated entites with the following command.

```
EntityValue["HistoricalPeriod", "EntityCount"]
```

Each historical period entity can be queried for the following properties (although there is no guarantee that there will always be associated data).

```
EntityValue["HistoricalPeriod", "Properties"]
```

Origin of Species was a product of the Victorian era. We check to see that there is an entity for that, and then request some of its associated properties.

```
Interpreter["HistoricalPeriod"]["Victorian era"]
```

```
EntityValue[Interpreter["HistoricalPeriod"]["Victorian era"], "StartDate"]
```

```
EntityValue[Interpreter["HistoricalPeriod"]["Victorian era"], "EndDate"]
```

```
EntityValue[Interpreter["HistoricalPeriod"]["Victorian era"], "PeopleInvolved"]
```

If we are not sure how the Victorian era fits in with other historical periods, we can try retrieving entities for some other periods. We know Darwin was a scientist (that word began its rise to prominence after *Origin* was published), an explorer, a ‘Renaissance’ man, etc. To graphically display historical periods, we can use **TimelinePlot** with a list of elements. If you hover over the bubbles with your mouse, start and end dates are shown. These are entities, too.

```
TimelinePlot[{Interpreter["HistoricalPeriod"]["age of exploration"],
  Interpreter["HistoricalPeriod"]["renaissance"],
  Interpreter["HistoricalPeriod"]["scientific revolution"],
  Interpreter["HistoricalPeriod"]["age of enlightenment"],
  Interpreter["HistoricalPeriod"]["industrial revolution"],
  Interpreter["HistoricalPeriod"]["second industrial revolution"],
  Interpreter["HistoricalPeriod"]["victorian era"]}]
```

Historical events occur on a much shorter time frame than periods, and *Mathematica* has access to entities representing far more of them. Each historical event entity has a number of associated properties.

```
EntityValue["HistoricalEvent", "EntityCount"]
```

```
EntityValue["HistoricalEvent", "Properties"]
```

In order to figure out which of these event entities might provide context for *Origin of Species* and Darwin's other works, we will download the entity, name and start date for each of them in the form of an association. In this case, the entity is the key, and the associated value is a list containing the name and start date. (This takes a little while).

```
histevents = EntityValue[EntityValue["HistoricalEvent", "Entities"],
    {"Name", "StartDate"}, "EntityAssociation"];
```

```
Length[histevents]
```

```
Head[histevents]
```

Next we can use a **Select** command to pull out events that have to do with Darwin and plot them on a timeline. We limit our search to things that happened before his death in 1882. The **DateList** command allows us to extract the year from a date entity.

```
DateList[ Tue 27 Dec 1831 ]
```

```
DateList[ Tue 27 Dec 1831 ][[1]]
```

```
darwinEventsAssoc = Select[histevents,
  StringContainsQ[#[[1]], "Darwin" | "Beagle"] && DateList[#[[2]]][[1]] ≤ 1882 &]
```

Recall that in the last chapter, we also retrieved a set of entities representing some of the books that Darwin wrote.

```
darwinBookEntities =
  PersonData[Interpreter["ComputedPerson"]["Darwin"], "NotableBooks"];
```

```
darwinBookEntities // TableForm
```

We can visualize both Darwin's publications (in blue) and some of the significant events in his life (in gold) on the same timeline, using the following command. The *PlotLayout→"Overlapped"* and *AxesOrigin→Center* properties prevent the timeline from becoming too cluttered to read.

```
TimelinePlot[{darwinBookEntities → "FirstPublished", Keys[darwinEventsAssoc]},
 PlotLayout → "Overlapped", AxesOrigin → Center]
```

Lifespans

Another way to contextualize Darwin's life is to visualize it in the context of some of his predecessors and interlocutors. In the previous chapter we retrieved birth and death dates for some of the people mentioned in *Origin*. The function and timeline plot below show Darwin's lifetime compared with some of the people who influenced him (his grandfather Erasmus, the explorer Captain Cook, earlier natural historians like Lamarck and Cuvier) and some of his younger contemporaries (Wallace and Huxley). We use the **Interval** command to represent all of the time between the birth and death dates.

```
lifespan[person_] :=
 Labeled[
  Interval[{PersonData[Interpreter["ComputedPerson"][person], "BirthDate"],
    PersonData[Interpreter["ComputedPerson"][person], "DeathDate"]}], person]
```

```
TimelinePlot[{{lifespan["James Cook"]}, {lifespan["Erasmus Darwin"]},
  {lifespan["Jean-Baptiste Lamarck"]}, {lifespan["Alexander Humboldt"]},
  {lifespan["Georges Cuvier"]}, {lifespan["Charles Darwin"]},
  {lifespan["Thomas Huxley"]}, {lifespan["Alfred Russel Wallace"]}}]
```

We can also combine lives and works, as in the following figure. Here each author's works are given the same color as their lifespan.

```
TimelinePlot[{
  {lifespan["Charles Darwin"], Labeled["1839", Style["Beagle", Italic]],
   Labeled["1859", Style["Origin", Italic]]},
  {lifespan["Charles Lyell"], Labeled[Interval[{"1830", "1833"}],
    Style["Principles of Geology", Italic]]},
  {lifespan["Robert Chambers"], Labeled["1844", Style["Vestiges", Italic]]}
 }]
```

The Voyage of the *Beagle*

One of the most important events in Darwin's life was the voyage that he made on the H.M.S. *Beagle* as a young man. (In fact, the first sentence of *Origin* refers to it). Using the entity representing Darwin and the one representing his voyage, we can do some quick date calculations. How long did the voyage last? We can use the **DateDifference** command to find out.

Darwin's Voyage of the H.M.S. Beagle (historical event)

```
DateDifference[
 EntityValue[Interpreter["HistoricalEvent"]["darwin voyage beagle"],
  "StartDate"], EntityValue[
  Interpreter["HistoricalEvent"]["darwin voyage beagle"], "EndDate"]]
```

If we would rather have the information in a more traditional form, we can use **UnitConvert** to convert it.

```
UnitConvert[Quantity[1741, "Days"],
 MixedRadix["Years", "Months", "Days", "Hours"]]
```

How old was Darwin when he set out? When he returned?

```
DateDifference[
 PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "BirthDate"],
 EntityValue[Interpreter["HistoricalEvent"]["darwin voyage beagle"],
  "StartDate"], {"Year", "Month", "Day"}]
```

```
DateDifference[
 PersonData[Interpreter["ComputedPerson"]["Charles Darwin"], "BirthDate"],
 EntityValue[Interpreter["HistoricalEvent"]["darwin voyage beagle"],
  "EndDate"], {"Year", "Month", "Day"}]
```

Note that we can also do the unit conversion as part of a call to the **DateDifference** command.

It would be nice to be able to retrieve the *Beagle*'s itinerary with a call to *Wolfram Alpha*, but that is not available, at least not yet. In a later chapter we will retrieve a detailed itinerary from a website about Darwin. For now, I have simply created a list of a few of the places that the ship visited on its voyage around the world. Since each of these are entities, we can use them to do various kinds of geospatial computation and visualization.

```
beagleBriefItinerary = {{{1831, 12, 28}, Plymouth (city)},
    {{1832, 1, 4}, Madeira Island (island)}, {{1832, 1, 16}, Praia (city)},
    {{1832, 2, 20}, Fernando de Noronha (island)}, {{1832, 2, 28}, Salvador (city)},
    {{1832, 4, 4}, Rio de Janeiro (city)}, {{1832, 7, 26}, Montevideo (city)},
    {{1832, 9, 6}, Bahia Blanca (city)}, {{1832, 10, 26}, Montevideo (city)},
    {{1832, 11, 2}, Buenos Aires (city)}, {{1832, 11, 14}, Montevideo (city)},
    {{1832, 12, 16}, Tierra del Fuego, Argentina (administrative division)},
    {{1833, 2, 26}, Falkland Islands (country)},
    {{1833, 4, 28}, Maldonado (city)}, {{1833, 8, 3}, Viedma (city)},
    {{1833, 8, 17}, Bahia Blanca (city)}, {{1833, 9, 20}, Buenos Aires (city)},
    {{1833, 11, 4}, Montevideo (city)}, {{1834, 2, 24}, Wollaston Island (island)},
    {{1834, 3, 10}, Falkland Islands (country)}, {{1834, 6, 28},
      Chiloé, Los Lagos, Chile (administrative division)}, {{1834, 7, 23}, Valparaiso (city)},
    {{1834, 11, 21}, Chiloé, Los Lagos, Chile (administrative division)},
    {{1834, 12, 13}, Fitzroy Island (island)}, {{1835, 2, 9}, Valdivia (city)},
    {{1835, 3, 4}, Concepción (city)}, {{1835, 3, 11}, Valparaiso (city)},
    {{1835, 5, 14}, Coquimbo (city)}, {{1835, 7, 19}, Lima (city)},
    {{1835, 9, 16}, San Cristóbal (island)}, {{1835, 11, 15}, Tahiti Island (island)},
    {{1835, 12, 21}, Paihia (city)}, {{1836, 1, 12}, Sydney (city)},
    {{1836, 2, 5}, Hobart (city)}, {{1836, 3, 6}, Albany (city)},
    {{1836, 4, 1}, Cocos Keeling Islands (country)},
    {{1836, 4, 29}, Mauritius (country)}, {{1836, 6, 1}, Cape Town (city)},
    {{1836, 7, 8}, Saint Helena, Ascension and Tristan da Cunha (country)},
    {{1836, 8, 1}, Salvador (city)}, {{1836, 8, 31}, Praia (city)}, {{1836, 9, 13},
      Azores, Portugal (administrative division)}, {{1836, 10, 2}, Falmouth (city)}};
```

There are 43 locations in the brief itinerary. If we think in terms of legs of the journey—the first being from Plymouth to Madeira Island, the second from Madeira Island to Praia—then there are 42 legs.

```
Length[beagleBriefItinerary]
```

Setting out

We can visualize the first leg of the journey using one of *Mathematica*'s geospatial commands. We start with **GeoListPlot**, which places the starting and ending points on a map. The *GeoLabels→True* option labels placenames, the *GeoRange→"Country"* option sets the scale so we can see the whole of the United Kingdom, and the *Joined→True* option shows a geodesic path between the two points.

```
GeoListPlot[{beagleBriefItinerary[[1, 2]], beagleBriefItinerary[[2, 2]]},
 GeoLabels → True, GeoRange → "Country", Joined → True]
```

If we would rather see a relief map, we can request one like this.

```
GeoListPlot[{beagleBriefItinerary[[1, 2]], beagleBriefItinerary[[2, 2]]},
 GeoLabels → True, GeoRange → "Country",
 Joined → True, GeoBackground → "ReliefMap"]
```

We can compute with geospatial entities. For example, the distance between Plymouth and Madeira Island (along a geodesic path) is

```
GeoDistance[beagleBriefItinerary[[1, 2]], beagleBriefItinerary[[2, 2]]]
```

GeoPosition takes a place entity and returns latitude and longitude. Here are the coordinates for Plymouth.

```
GeoPosition[beagleBriefItinerary[[1, 2]]]
```

We can use those coordinates to plot a map at a larger scale (i.e., zoomed in). Here we use the **GeoGraphics** command, set the scale to the size of the city and include a scale bar. This is a contemporary map, of course. In a later chapter we will learn how to overlay historical map images on contemporary digital maps, a process known as *georectification*.

```
GeoGraphics[GeoPosition[beagleBriefItinerary[[1, 2]]],
 GeoRange → "City", GeoScaleBar → "Kilometers"]
```

An interactive map for the voyage of the *Beagle*

Next we will build an interactive map. We will start with two panels, one for a small-scale map to provide context and one for a large-scale map showing the leg of the journey. The function below creates two map images given an itinerary and an integer representing which leg of the voyage we are visualizing. We scale the range of the context map to be the whole world, and the detail map to the distance between the starting and ending points.

```
mapImageLeg[itin_, n_] :=
 With[{dist = GeoDistance[itin[[n, 2]], itin[[n + 1, 2]]]},
  Column[{GeoGraphics[{GeoStyling["OutlineMap"],
       Red, Thick, GeoPath[{itin[[n, 2]], itin[[n + 1, 2]]}]},
      GeoRange → All, ImageSize → {300, Automatic}],
     GeoListPlot[{itin[[n, 2]], itin[[n + 1, 2]]}, GeoRange → dist,
      GeoScaleBar → Placed[{0, 400} km, {Right, Bottom}], GeoLabels → True,
      Joined → True, ImageSize → {300, Automatic}]}, Center, Scaled[0.01]]]
```

```
mapImageLeg[beagleBriefItinerary, 1]
```

It takes some time to create a map, so we will render each of the 42 maps in advance. Then we can step forwards and backwards through them quite quickly. In order to do this, we will want to **Map** the rendering function (*mapImageLeg*) across the list of numbers from 1 to 42. Generating such a list is quite easy with the **Range** command.

```
Range[3]
```

The command below simply renders all of the maps. We suppress the output because we don't need to see the rendered maps. It takes a while to execute this the first time, so be patient.

```
Map[mapImageLeg[beagleBriefItinerary, #] &, Range[42]];
```

Now we can use the **Manipulate** command to create an interactive interface for our map viewer. This will give us a slider that allows us to explore each leg of the journey, one at a time. The list *{n,1,42,1}* is known as an 'iterator'. It says that *n* can take values between 1 and 42 in increments of 1. If you press the plus sign beside the slider, it will open up an interface that allows you to step

forward and back one frame at a time, or to animate the voyage. Spend a little time familiarizing yourself with the **Manipulate** interface. You will discover that its default animation speed is too fast (it skips frames) and needs to be slowed down. In some of the maps, the *Beagle* is shown impossibly traveling over land. We will remedy this problem in a future visualization.

```
Manipulate[mapImageLeg[beagleBriefItinerary, n], {n, 1, 42, 1}]
```

Adding dates

Now that we have an interactive map, we can add starting and ending dates to each leg of the journey. We use the **DateObject** command to convert each **DateList** to an entity, and then a **Row** command to put them side-by-side. (The **Invisible** command puts a space between the two dates that is the width of a lowercase "m"). We then **Center** the dates in a **Column** above the maps.

```
itineraryBrowser[itin_] := Manipulate[Column[
    {Row[{DateObject[itin[[n, 1]]], Invisible["m"], DateObject[itin[[n + 1, 1]]]}],
     mapImageLeg[itin, n]}, Center], {n, 1, Length[itin] - 1, 1}]
```

```
itineraryBrowser[beagleBriefItinerary]
```

Summary

Investigating the named entities in a text allows you to more easily answer questions about *who*, *what*, *where* and *when*. When those entities are used to query computable data, you can go well beyond the internal evidence of your sources, linking them to an ever-expanding network of relevant information. Researchers have always done this kind of exploration manually, of course. Programming with *Mathematica* allows some of this exploration to be performed by machine. We have seen a couple of examples of simple reference tools that can be automatically generated to contextualize our sources. As we go on, we will find more and more ways that programming can make research easier, faster, more powerful and much more efficient.

Generalizing the Examples

Collocations

So far we have investigated a number of ways of automatically finding aspects of a text that are distinctive or meaningful. In our search for named entities, for example, we found that capitalization can be very useful. And as we have seen, once we have successfully associated a string with an entity, we can use that entity to retrieve or compute new information.

Another approach is to search the text for *collocations*, phrases or n-grams whose meaning goes beyond the meanings of the individual words that comprise them. In *Origin of Species*, for example, Darwin uses the phrase 'natural selection' to refer to a set of ideas that go well beyond the meanings of either 'natural' or 'selection'. Here we partially automate the process of discovering collocations.

We begin by generating bigram frequencies for the whole text of *Origin*. These are returned as an association, sorted in order of descending frequency.

```
originBigrams = WordCounts[origin, 2];
```

As we discovered before, the most frequent n-grams contain stopwords.

```
Short[originBigrams, 5]
```

We can use the **Select** command to choose bigrams that occur five or more times in the text.

```
originBigramsFreq5 = Select[originBigrams, # ≥ 5 &];
```

These make up about 10% of the total number of bigrams found in the text.

```
Length[originBigramsFreq5] / Length[originBigrams] // N
```

Now we want to get rid of the frequent bigrams that contain one or more stopwords. To do this, we can use the **Select** command along with the *nonStopwordQ* function we developed earlier. First note that we are interested in retrieving the **Keys** from our association. Here are the first ten.

```
Keys[originBigramsFreq5][[1 ;; 10]]
```

Let's select the bigrams from the top 50 most frequent that do not include a stopword. There is only one example, 'natural selection'.

```
Select[Keys[originBigramsFreq5][[1 ;; 50]],
 (nonStopwordQ[#[[1]]] && nonStopwordQ[#[[2]]]) &]
```

In the next 50 we find another example, 'organic beings'.

```
Select[Keys[originBigramsFreq5][[50 ;; 100]],
 (nonStopwordQ[#[[1]]] && nonStopwordQ[#[[2]]]) &]
```

As we dig into the less frequent bigrams, we will begin to find examples that don't really convey much that is distinctive about *Origin*.

```
Select[Keys[originBigramsFreq5][[700 ;; 750]],
 (nonStopwordQ[#[[1]]] && nonStopwordQ[#[[2]]]) &]
```

The difference between a phrase like 'natural selection' or 'larger genera' and one like 'for instance' or 'we shall' is that the former consist of an adjective followed by a noun. So we will limit our search to frequent bigrams that consist of an adjective and noun or a pair of nouns.

Here is our list of 'interesting' frequent bigrams in *Origin*. Note that some of them are named entities ("South America", "North America" and "New Zealand"). Many of these terms are the kinds of things that you might want to look up in the index of a book.

```
originBigramsFreq5Interesting =
  Select[Keys[originBigramsFreq5], (nonStopwordQ[#[[1]]] && nonStopwordQ[#[[2]]]) &&
     ((adjectiveQ[#[[1]]] || nounQ[#[[1]]]) && nounQ[#[[2]]]) &];
```

```
viewData[originBigramsFreq5Interesting]
```

Studying these particular bigrams, we can begin to get a sense of Darwin's technical language. We've already noted that he used the phrase "natural selection" to refer to a particular collection of concepts, and the same is also true of his frequent use of "sexual selection". We can use the **Cases** command to pull out other bigrams involving "selection", then use our *textSearch* function to investigate his use of a particular term. We can see, for example, that he uses "methodical selection" to refer to the activities of breeders.

```
Cases[originBigramsFreq5Interesting, {_, "selection"}]
```

```
textSearch[origin, "methodical selection"]
```

It is not surprising that a book entitled *Origin of Species* would discuss "species", but the word appears in a number of frequent collocations that suggest the range of the discussion. The same is true of Darwin's use of "forms". In some cases, the same adjective is frequently used with both words (e.g., "new", "allied", "dominant", "living", "extinct", "existing"). In other cases, a particular adjective is used only with one term (e.g., "distinct species", "doubtful forms"). The analysis of an author's use of vocabulary at this level can serve a variety of purposes: better understanding his or her conceptual world; studying the rise and fall of particular collocations over time or their spread through a particular community; resolving questions of disputed authorship, and so on.

```
Cases[originBigramsFreq5Interesting, {_, "species"}]
```

```
Cases[originBigramsFreq5Interesting, {_, "forms"}]
```

We can also study the distribution of frequently used adjectives. Darwin frequently uses “distinct”, for example, in discussing taxonomic categories, and “domestic” in an overlapping but not identical set of contexts.

```
Cases[originBigramsFreq5Interesting, {"distinct", _}]
```

```
Cases[originBigramsFreq5Interesting, {"domestic", _}]
```

Visualizing cooccurrence within chapters

At this point we have developed a number of techniques for studying words in close proximity to one another: pattern matching, n-grams, keyword in context listings, collocations, and custom functions like *stringFindNear*. A different approach is to study the distribution of particular words or terms across a source or collection of sources. We might ask, for example, whether a particular word or term is found throughout a work or limited to one or more chapters.

Let’s begin by creating a function that counts the number of instances of a word or phrase in each chapter of *Origin*. First we create a list of all of the words in the book in order.

```
originWords = TextWords[origin];
```

```
Short[originWords, 5]
```

```
Length[originWords]
```

Recall that each chapter begins with the word “chapter” in uppercase, followed by a space and an integer. Some chapters have a “SUMMARY OF CHAPTER” section that we don’t want to mistake for the beginning of a new chapter. So the chapter number is important.

```
textSearch[origin, "CHAPTER"]
```

We can use the following commands to display the word positions for each instance of “CHAPTER” along with some context. Then we simply remove the positions that don’t correspond to actual chapter beginnings.

```
Position[originWords, "CHAPTER"]
```

```
Map[{#, originWords[[First[#] ;; First[#] + 3]]} &, Position[originWords, "CHAPTER"]]
```

```
originChapterBeginnings =
 Complement[Position[originWords, "CHAPTER"], {{39 096}, {62 529}, {84 971}}]
```

Now that we have this list, we know, for example, that all the words between position 1 and position 1709 fall into the Introduction, all of the words between 1710 and 13377 fall into Chapter 1, and so on. Next we create a function that, given a word position, returns the chapter of *Origin* that contains that instance (we will think of the Introduction as Chapter 0). The **Piecewise** command is perfect for this. It checks a series of conditions and returns the first one that matches. The last condition returns a -1 if the input number is outside the bounds of possible positions. That shouldn’t happen.

```
originChapterLookup[pos_Integer] :=
 Piecewise[{{0, 1 ≤ pos < 1710}, {1, 1710 ≤ pos < 13 378},
   {2, 13 378 ≤ pos < 18 269}, {3, 18 269 ≤ pos < 24 470}, {4, 24 470 ≤ pos < 40 192},
   {5, 40 192 ≤ pos < 52 453}, {6, 52 453 ≤ pos < 63 627}, {7, 63 627 ≤ pos < 75 594},
   {8, 75 594 ≤ pos < 85 590}, {9, 85 590 ≤ pos < 95 349}, {10, 95 349 ≤ pos < 105 721},
   {11, 105 721 ≤ pos < 117 156}, {12, 117 156 ≤ pos < 125 749},
   {13, 125 749 ≤ pos < 140 375}, {14, 140 375 ≤ pos ≤ 149 982}, {-1, True}}]
```

Now we can check our function. The Chapter 4 summary begins at position 39,096. When we test this position, the function should tell us that it is in Chapter 4.

```
originChapterLookup[39 096]
```

Armed with this function we can get the positions for a particular word.

```
Position[originWords, "geological"]
```

Test each position to see which chapter it falls into.

```
Map[originChapterLookup, Flatten@Position[originWords, "geological"]]
```

Then use the **Tally** command to count the occurrences per chapter.

```
Tally[Map[originChapterLookup, Flatten@Position[originWords, "geological"]]]
```

For reasons that are explained in the 'Programming with *Mathematica*' section below, this information will be more useful to us in the form of a flat list with 15 elements. Each element will either be a zero (if the term doesn't occur in that chapter) or an integer indicating the number of occurrences. We can generate this list by using **BinCounts** instead of **Tally**. Take a moment to compare the output of the two commands.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "geological"]], {0, 15}]
```

The **ArrayPlot** command allows us to visualize this list. The *ColorFunction→"TemperatureMap"* option plots numbers from low to high on a spectrum from dark blue to red. The other options are necessary to get the chapter numbering right on the ticks across the bottom of the figure.

```
chticks = Prepend[Range[14], "Intro"];
ct = Map[{#, Rotate[chticks[[#]], Pi / 2]} &, Range[15]];
```

```
ArrayPlot[{{4, 0, 0, 0, 4, 1, 2, 0, 0, 27, 21, 10, 4, 2, 9}},
 ColorFunction → "TemperatureMap",
 FrameLabel → {Style["geological", Italic], "Chapter"},
 RotateLabel → False, FrameTicks → {None, ct}, ImageSize → Large]
```

In addition to searching *darwinOriginWords* for the position of individual words, we would also like to find the positions of collocations (or more generally, n-grams). We can use the **SequencePosition** command for this task.

```
SequencePosition[originWords, {"geological", "record"}]
```

We can make a function that checks a string to see if it is a single word or a collection of words, then uses **Position** or **SequencePosition** as appropriate. When we are matching multiple words, we return only the position of the first one.

```
StringSplit["geological"]
```

```
StringSplit["geological record"]
```

```
getPosition[wordlist_, str_] :=
 Module[{termlist = StringSplit[str]},
  If[Length[termlist] == 1,
   Flatten@Position[wordlist, First@termlist],
   Flatten@Map[First, SequencePosition[wordlist, termlist]]]]
```

```
getPosition[originWords, "geological"]
```

```
getPosition[originWords, "geological record"]
```

Finally, we will bundle everything we have done so far into a function that lets us visualize the cooccurrence of a number of terms at once.

```
originChapterCooccurrence[terms_List] :=
 Module[{termarray, cht, cticks, rticks},
  termarray = Map[BinCounts[
      Map[originChapterLookup, getPosition[originWords, #]], {0, 15}] &, terms];
  cht = Prepend[Range[14], "Intro"];
  cticks = Map[{#, Rotate[cht[[#]], Pi / 2]} &, Range[15]];
  rticks = Map[{#, terms[[#]]} &, Range[Length[terms]]];
  ArrayPlot[termarray, ColorFunction → "TemperatureMap",
   FrameLabel → {None, "Chapter"}, RotateLabel → False,
   FrameTicks → {rticks, cticks}, ImageSize → Large]]
```

Here are some of the people who are mentioned in *Origin*. When we study the figure, we can see patterns of cooccurrence that may help us to understand the role that various individuals play in the work. Even if we don't know who Owen, Agassiz and Cuvier were, we can see that they mostly show up in the same chapters. We can see that Lyell dominates Chapter 9, Hooker Chapters 11 and 12, and Owen Chapters 10 and 13. We can see that Murchison, Sedgwick and Dawson are similar to Lyell, and so on.

```
originChapterCooccurrence[
 {"Lyell", "Murchison", "Sedgwick", "Dawson", "Humboldt", "Muller",
  "Wallace", "Hooker", "Bentham", "Huxley", "Owen", "Agassiz", "Cuvier"}]
```

We can also check for cooccurrences between people and places, as shown in the next figure.

```
originChapterCooccurrence[{"Scotland", "Lyell", "Australia",
  "Hooker", "Wallace", "Malay", "Galapagos", "Chile", "Humboldt"}]
```

Another thing we can do with a visualization like this is study phrases to see how widely they are distributed across the text. These are all of the terms that were used to modify the word "species", for example, sorted by decreasing frequency.

```
StringRiffle /@ Cases[originBigramsFreq5Interesting, {_, "species"}]
```

```
originChapterCooccurrence[
 StringRiffle /@ Cases[originBigramsFreq5Interesting, {_, "species"}]]
```

We can see that the more frequent collocations ("distinct species", "new species", etc.) are spread widely throughout the text, whereas some of the less frequent ones ("pure species", "original species", "endemic species") only occur in a single chapter.

Programming with *Mathematica*

Applying and using pure functions

Suppose we want to add two numbers together. As we've seen, *Mathematica* has a **Plus** command which is usually written with infix notation.

```
17 + 25
```

We can also write this with functional notation.

```
Plus[17, 25]
```

The two numbers that we give to the **Plus** command are called *arguments*. We have seen that we can create a pure function by replacing one or more of the arguments with a **Slot** (#) and adding an ampersand to the end (shorthand for the **Function** command). This pure function will add 17 to things. Note that when we evaluate it, it converts the **Plus** command to infix notation and it gives the slot a number (1 in this case).

```
Plus[17, #] &
```

We could also create a pure function with **Plus** that expects two arguments. Here we give them numbers ourselves.

```
Plus[#1, #2] &
```

Once we have a pure function expression (whether it expects to receive one argument or two), we can use it by applying it to values. Here is how we add 17 to 25 using our 'Plus 17' pure function.

```
(Plus[17, #] &)[25]
```

We could also write this with infix notation.

```
(17 + # &)[25]
```

Here is how we add 17 to 25 using our pure function that expects to receive two arguments.

```
(Plus[#1, #2] &)[17, 25]
```

We could also write this with infix notation.

```
(#1 + #2 &)[17, 25]
```

Using pure functions like this adds a level of conceptual complexity that we usually don't need, at least if we just want to add a few numbers together. Pure functions become very useful when we want to apply them repeatedly, however. One case that we have seen is **Map**. Here we add 17 to a list of numbers.

```
Map[(# + 17 &), {25, 50, 75, 150}]
```

Another case where pure functions are very useful is with commands like **Select**. Here is a pure function that checks to see whether a number is between 7 and 17, inclusive.

```
7 ≤ # ≤ 17 &
```

Here we use the pure function to **Select** numbers from a list that match this criterion.

```
Select[{3, 45, 32, 13, 4, 77, 12, 8, 19, 9, 44, 121, 32, 10}, 7 ≤ # ≤ 17 &]
```

Or we could use the **CountsBy** command to count how many numbers in a list match the criterion and how many don't.

```
CountsBy[{3, 45, 32, 13, 4, 77, 12, 8, 19, 9, 44, 121, 32, 10}, 7 ≤ # ≤ 17 &]
```

Vectors and matrices

In addition to supporting computation with data structures like strings, lists and associations, *Mathematica* also supports computation with mathematical objects. We have already encountered integers, rational numbers and real numbers. Some of the most useful mathematical objects for digital research methods are *vectors* and *matrices*, the subject of the branch of mathematics known as *linear algebra*. You don't need to be familiar with linear algebra to understand this section (if you are, it won't hurt, of course.)

In *Mathematica*, a vector is represented as a list. If we want to describe something in Cartesian coordinates, for example, we use a pair of numbers representing the horizontal (x) and vertical (y) position. We can plot a number of such points with the **Graphics** and **Point** commands. The command **Blue** changes the color of all of the points that follow, and the **PointSize[Medium]** command tells Graphics to draw slightly larger dots. The *Axes→True* option is required to see the x and y axes.

```
Graphics[{Blue, PointSize[Medium], Point[{2, 3}], Point[{1, -1}],
  Point[{-2, -2}], Point[{-1, 3}]}, ImageSize → Small, Axes → True]
```

Another interpretation of each of these vectors is as an arrow from the origin {0,0} to {x,y}. We can use the **Arrow** command instead of **Point** if we wish to visualize this.

```
Graphics[{Blue, Arrow[{{0, 0}, {2, 3}}],
  Arrow[{{0, 0}, {1, -1}}], Arrow[{{0, 0}, {-2, -2}}],
  Arrow[{{0, 0}, {-1, 3}}]}, ImageSize → Small, Axes → True]
```

Because each of our points or arrows can be specified with a pair of numbers, we can think of each position in the vector as representing a different dimension. This example is two-dimensional, but we could easily make a three-dimensional example using vectors with three positions, {x,y,z}. We call vectors with *n* elements "*n*-dimensional". They are more difficult to visualize but not necessarily more difficult to work with. In fact, we have already started working with them.

In the previous section we wanted to visualize the number of occurrences of a particular word or term in each chapter of *Origin*. Our original output looked like this:

```
Tally[Map[originChapterLookup, Flatten@Position[originWords, "geological"]]]
```

There are four tokens of "geological" in the Introduction, four in Chapter 4, one in Chapter 5, two in Chapter 6, and so on. To use the **ArrayPlot** command, however, we needed a *vector* instead. (In this context, we can think of "array" as another word for vector or matrix.) To get the corresponding vector we used the **BinCounts** command instead of **Tally**.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "geological"]], {0, 15}]
```

This is a fifteen-dimensional vector that represents the distribution of the word "geological" throughout the chapters of *Origin of Species*.

A matrix is represented in *Mathematica* as a nested list, a list of lists. If we wanted to visualize the occurrence of both "geological" and "record" in various chapters (not necessarily as a collocation) we would need to provide a list of lists to our plotting function. Here is the vector for the word "record".

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "record"]], {0, 15}]
```

Combining the two vectors we have this nested list.

```
{{4, 0, 0, 0, 4, 1, 2, 0, 0, 27, 21, 10, 4, 2, 9},
 {0, 4, 0, 1, 1, 0, 4, 0, 0, 16, 8, 1, 0, 2, 9}}
```

We can visualize it as a matrix with **MatrixForm**. This makes it easier to see that it has two rows and fifteen columns.

```
MatrixForm[{{4, 0, 0, 0, 4, 1, 2, 0, 0, 27, 21, 10, 4, 2, 9},
  {0, 4, 0, 1, 1, 0, 4, 0, 0, 16, 8, 1, 0, 2, 9}}]
```

We can also visualize it with **ArrayPlot**.

```
ArrayPlot[{{4, 0, 0, 0, 4, 1, 2, 0, 0, 27, 21, 10, 4, 2, 9},
  {0, 4, 0, 1, 1, 0, 4, 0, 0, 16, 8, 1, 0, 2, 9}}, ColorFunction → "TemperatureMap"]
```

Vector distance and similarity

In addition to helping us with visualization, vector representation gives us a way to determine whether two things are similar or not. Let's return to the two dimensional case. We want to say that two vectors that are pointing in the same general direction are more similar to one another (or less distant from one another) than two vectors that are pointing in different directions. The pair of blue vectors below are similar (or not very distant from one another) and the green and red one are different (or distant).

```
Graphics[{Blue, Arrow[{{0, 0}, {2, 3}}],
  Arrow[{{0, 0}, {2, 4}}], Red, Arrow[{{0, 0}, {-2, -2}}], Green,
  Arrow[{{0, 0}, {-1, 3}}]}, ImageSize → Small, Axes → True]
```

One measure of distance or similarity between vectors is called **CosineDistance**, implemented in *Mathematica* by a command of the same name. For the blue vectors the value is very small.

```
CosineDistance[{2, 3}, {2, 4}] // N
```

For the red and green vector the value is much larger.

```
CosineDistance[{-2, -2}, {-1, 3}] // N
```

We can use the same measure on n-dimensional vectors. In the previous section, for example, we noticed that Murchison and Sedgwick appeared in the same contexts. We expect the **CosineDistance** between their respective vectors to be low.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "Murchison"]], {0, 15}]
```

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "Sedgwick"]], {0, 15}]
```

```
CosineDistance[{0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0}] // N
```

On the other hand, Murchison and Wallace appeared in different contexts. So their **CosineDistance** should be higher.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "Wallace"]], {0, 15}]
```

```
CosineDistance[{0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0},
 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0}]
```

For frequent collocations, the **CosineDistance** between the two terms should be low. Here is the calculation for “natural” and “selection”.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "natural"]], {0, 15}]
```

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "selection"]], {0, 15}]
```

```
CosineDistance[{1, 4, 4, 4, 56, 47, 61, 35, 7, 9, 14, 7, 7, 35, 27},
  {0, 30, 3, 3, 81, 57, 65, 42, 5, 4, 9, 6, 4, 15, 23}] // N
```

For frequent words that aren’t collocates (e.g., “natural” and “beings”) the measure should be higher.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "beings"]], {0, 15}]
```

```
CosineDistance[{1, 4, 4, 4, 56, 47, 61, 35, 7, 9, 14, 7, 7, 35, 27},
  {6, 2, 2, 10, 19, 5, 6, 1, 7, 2, 5, 6, 5, 10, 22}] // N
```

In the next chapter we will see that the ability to measure the distance between vectors allows us to automate a variety of very useful tasks.

Mathematica Commands to Review

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration
- ArrayPlot (GE)
- Arrow (PW)
- BinCounts (GE)
- Center (BE)

- CosineDistance (PW)
- CountsBy (PW)
- DateDifference (BE)
- DateList (BE)
- DateObject (BE)
- GeoGraphics (BE)
- GeoListPlot (BE)
- GeoPosition (BE)
- Graphics (PW)
- Interval (BE)
- Invisible (BE)
- Manipulate (BE)
- Piecewise (GE)
- Point (PW)
- SequencePosition (GE)
- SparseArray (FE)
- StringContainsQ (BE)
- TimelinePlot (BE)
- UnitConvert (BE)

ch05

Chapter 05: Information Retrieval

Overview

In the first few chapters we discovered that much can be learned about a text simply by studying the words and n-grams that occur most frequently in it. Using pattern matching, we were also able to locate keywords in the text and display them in context. Some of those keywords referred to entities like people, places and dates which we used to retrieve computable data. We can do much more with a mathematical approach to textual analysis, however. Given a text, we can use computational methods to figure out what it is about. Given one text, we can find others that are similar to it. And given a large collection of texts, we can automatically *cluster* them into groups of related documents. These techniques all fall under the broad heading of *information retrieval*, and what they have in common are a variety of methods for determining when two texts are similar to one another, or when a text is relevant to a particular query or information need.

Basic Examples

Working with pages

Up until now we have been working as if we did not have pagination information for our copy of *Origin of Species*. We were able to use string matching to extract chapters based on internal evidence in the text, but for most of our examples, we simply worked with sequences of characters or words. If you are analyzing some kinds of texts, such as web pages or e-books, this may be the best that you can do. Most such documents are designed to *reflow* gracefully so they can be read

on screens with different resolutions. Under such conditions, the traditional page is not really a natural unit of analysis, and it makes more sense to work with named subsections, sections or chapters.

Origin of Species was first published as a *codex* in 1859, however, so the page is definitely an appropriate unit of analysis. If we want to do a page-based analysis of some sort, we have a number of options. One is to try to estimate where the page breaks would most likely be. Since *Mathematica* is designed to take advantage of computable data, it knows about many characteristics of familiar objects. We have already seen evidence of this in our use of commands like **WordData**, **WolframAlpha** and **WikipediaData**. As we progress we will draw on computable data whenever possible.

The **FormulaLookup** command allows us to search through a very large set of formulas that describe how things in the world behave. The following example shows that we can use *Mathematica* to estimate the number of words per page for books, manuscripts, magazines and screenplays, and that we can take into account variables like pagination, spacing and font (if we know them).

```
FormulaLookup["words per page"]
```

Assuming we don't know anything about *Origin*'s pagination, spacing or font, we can get a formula for the number of words per page with the **FormulaData** command as follows.

```
FormulaData[{"WordsPerPage", "Book"}]
```

This says that *Mathematica*'s best guess is 250 words per page on average. (Alternately, the total number of pages is 1/250th of the total number of words). We can substitute values into this formula with a **Rule**. How many pages for 150,000 words?

```
FormulaData[{"WordsPerPage", "Book"}, {"w" → 150 000}]
```

How many words in 27 pages?

```
FormulaData[{"WordsPerPage", "Book"}, {"p" → 27}]
```

But we don't have to estimate the pagination in this case. In fact, we can request a copy of *Origin* from **ExampleData** that is formatted as a list of strings, where each string represents a header, a space, or the contents of one page.

```
originLines = ExampleData[{"Text", "OriginOfSpecies"}, "Lines"];
```

```
Head[originLines]
```

```
Length[originLines]
```

Chapter titles get their own lines.

```
originLines[[1]]
```

The space between a title and text is a null string (i.e., “”). When we evaluate the expression, we don't see anything.

```
originLines[[2]]
```

We can use the **FullForm**, **Head** and **StringLength** commands to see that this apparent nothingness is actually a string of zero length.

```
FullForm[originLines[[2]]]
```

```
Head[originLines[[2]]]
```

```
StringLength[originLines[[2]]]
```

The third line of *originLines* is the string containing the text of the first page.

```
originLines[[3]]
```

As with the example of sentence lengths in Chapter 1, we can study both the number of characters and the number of words per line.

```
ListPlot[Map[StringLength, originLines], Filling → Axis,
 AxesLabel → {"Line Number", "Length in Characters"}, ImageSize → Large]
```

```
ListPlot[Map[WordCount, originLines], Filling → Axis,
 AxesLabel → {"Line Number", "Length in Words"}, ImageSize → Large]
```

We can also plot the distribution of line lengths in words, using the **Histogram** command. The figure below shows us that there are more than 800 lines that are 50 words or shorter; the rest of the lines are typically a few hundred words.

```
Histogram[Map[WordCount, originLines]]
```

How many lines are blank (i.e., used as spacers)?

```
Count[originLines, ""]
```

That leaves about fifty lines which contain between 1 and 50 words. Let's have a look at those. We can see that many of them are summary statements.

```
viewData[Select[originLines, 1 ≤ WordCount[#] ≤ 50 &]]
```

Searching pages

It would be nice to have a search function that returns pages where a term or phrase appears. We can actually do that quite easily with the **Select** and **StringContainsQ** commands. For example, the word 'electricity' appears on one page of *Origin*.

```
Select[originLines, StringContainsQ[#, "electricity"] &]
```

It would also be nice to know which line contains that page in the *originLines* list. That will require a little bit more work. First we are going to make a list of rules where we have a line number pointing to the string with the actual contents of that line. The beginning of this list will look like this:

```
{1 → "INTRODUCTION.",
 2 → "",
 3 → "When on board H.M.S. 'Beagle,' as naturalist, ...",
 4 → "",
 5 → "My work is now nearly finished; but as it will take me ...",
 ...}
```

We need a total of 1515 line numbers.

```
Length[originLines]
```

We can use the **Range** command to generate a list of numbers from 1 to n. Here is how we generate the numbers from 1 to 3.

```
Range[3]
```

We can use the **Thread** and **Rule** commands to create a nested list where each line number is paired with the contents of that line. Study the next example.

```
Thread[Rule[Range[3], {a, b, c}]]
```

So this is the code we need to create the nested list of line numbers and line contents shown above.

```
Thread[Rule[Range[Length[originLines]], originLines]]
```

Now we need to modify the pure function that our **Select** command uses. This version looked at

each element in *originLines* and returned those strings that contained the word 'electricity.

```
StringContainsQ[#, "electricity"] &
```

But each element in our list with line numbers is a **Rule**. The first part of the rule is the line number, and the second part of the rule is the string. So we need our pure function to search only in the second part. This is what the pure function looks like now.

```
StringContainsQ[#[[2]], "electricity"] &
```

We can now put everything together into a function to search *originLines*. We hand the results off to the **TabView** command, which uses the line number to label each tab.

```
lineSearch[linelist_, str_] :=
 TabView[Select[Thread[Rule[Range[Length[linelist]], linelist]],
   StringContainsQ[#[[2]], str] &]]
```

```
lineSearch[originLines, "electricity"]
```

Here is another example.

```
lineSearch[originLines, "crab"]
```

The document vector model

In the last chapter we saw that we can use vectors to study the distribution of a word or phrase throughout a text. The expression below shows how many times the word 'organic' occurs in the Introduction and each of the fourteen chapters of *Origin*, represented as a fifteen-dimensional vector.

```
BinCounts[
 Map[originChapterLookup, Flatten@Position[originWords, "organic"]], {0, 15}]
```

We also learned that we can use a measure called **CosineDistance** to figure out if two different words are distributed throughout the text in a more-or-less similar fashion.

But what if we want to study the similarity or difference of two texts? These texts could be whole documents, pieces of documents like chapters, sections or pages, or even sentences or short passages. We can use another, different vector representation to accomplish this.

Recall that we have already developed methods to determine the frequency of words. Here is a list of word frequencies for the whole of *Origin*.

```
originWordFreqs = WordCounts[origin, IgnoreCase → True];
```

```
Short[originWordFreqs, 5]
```

Suppose we take the first page of the book.

```
originLines[[3]]
```

We can convert it to a list of lowercase words, remove the stopwords, and then get rid of duplicates. (This representation, the *bag of words*, was introduced in Chapter 1).

```
line3words = Union@DeleteStopwords@ToLowerCase@TextWords[originLines[[3]]]
```

Then we can look up the frequency of each of those words in *originWordFreqs*, and **Sort** in order of descending frequency. (Note our use of the **Greater** command in this context. We also used **Rest** here to get rid of the first element of *line3words*, the "...", because its frequency was not counted by the **WordCounts** command.)

```
originWordFreqs[[{"board", "beagle", "naturalist"}]]
```

```
Sort[originWordFreqs[[Rest@line3words]], Greater]
```

Some of these words, like ‘philosophers’ or ‘mysteries’, only appear once in the book, on this page. Other words, like ‘species’, occur many times throughout the book. If we want to compare this page with other pages, a word like ‘philosophers’ doesn’t really do anything for us. This page, unlike every other page, contains that one word, so it is different from all the other pages in that sense. Whether a page contains the word ‘species’ or not is more useful, unless the word occurs on every page in the book. Our hunch is that pages that contain many of the same high-frequency words will be more similar to one another than they will be to pages that don’t contain many of the same words. We can capture this idea with a vector representation.

Here are the 200 most frequent words in *Origin*, not including stopwords.

```
freq200 = DeleteStopwords[Keys[originWordFreqs]][[1 ;; 200]]
```

Suppose we use these words, in this order, to specify a 200-dimensional vector. Then we can represent any given page by putting a one in the column if that word appears on that page, and a zero otherwise. Page one, for example, contains the words ‘species’ and ‘period’, but does not contain the words ‘forms’, ‘varieties’, ‘selection’, etc. Its vector will look like this:

We don’t need to figure out the vector by hand, of course. Instead, we will compute it using the *getFreqVector* function shown here. This function is derived and explained in the ‘Programming with *Mathematica*’ section below.

```
getFreqVector[textstr_, freq_] :=
 ReplacePart[ConstantArray[0, {Length[freq]}],
  Map[(First@First@Position[freq, #] → 1) &,
   Intersection[DeleteStopwords@ToLowerCase@TextWords[textstr], freq]]]
```

Here is the vector for the first page of *Origin*.

```
line3vec = getFreqVector[originLines[[3]], freq200]
```

This is the second page of the book, and its vector representation.

```
originLines[[5]]
```

```
line5vec = getFreqVector[originLines[[5]], freq200]
```

Now to determine how similar or different two vectors are, we will use a mathematical operation called the *vector product*. In *Mathematica* you use the **Dot** command to multiply vectors, like this:

```
Dot[{0, 0}, {0, 1}]
```

This can also be written as

```
{0, 1}.{0, 0}
```

Given two vectors *{a,b,c}* and *{d,e,f}* the vector product is defined as

```
{a, b, c}.{d, e, f}
```

Note that if either *a* or *d* is 0, then their product *ad* will also be zero. If both are 1, then their product will be 1.

```
{0, b, c}.{0, e, f}
```

```
{0, b, c}.{1, e, f}
```

```
{1, b, c}.{1, e, f}
```

The same logic goes for *b* and *e*, and for *c* and *f*. So for each column where both vectors have a 1, one will be added to the vector product. Study the following examples.

```
{0, 0, 0}.{0, 1, 1}
```

```
{1, 0, 0}.{0, 1, 1}
```

```
{1, 0, 0}.{1, 1, 1}
```

```
{1, 1, 0}.{1, 1, 1}
```

```
{1, 1, 1}.{1, 1, 1}
```

To test how many of the 200 most frequent words two pages have in common, we simply multiply their vectors.

```
line3vec.line5vec
```

We can read the result as saying that these two pages have 4 of the 200 most frequent words in common, not including stopwords.

It is important to note that this particular model is much more general than the example we have shown. Here we used the vector to represent a single page, and the vector product to determine the similarity of two pages. But we could just have easily determined the word frequencies for a whole corpus of documents (say the complete works of Darwin, or the complete works of a large group of nineteenth-century naturalists), and then used each vector to represent the high-frequency terms found in an individual document (say Darwin's 1861 letter to *The Field* on the "Influence of the form of the brain on the character of fowls".) The document vector model gives us one very powerful way of characterizing a large space of texts and of measuring similarity or difference between texts in that space.

Show me more like this

Once we have computed the vector representation for a particular text (whether it is a page or some other smaller or larger unit), we can compare that vector with others to accomplish a variety of goals. We can find other texts that are similar to one we are interested in, for example.

Let's start by computing *freq200* vectors for each line in our *originLines* list. Recall that each element in this list is either a header (like "INTRODUCTION."), a null string used as a spacer (""), or a string containing the text of a page. In the case of a spacer, the vector is going to consist of a list of 200 zeros because none of the high-frequency words appear in the empty string. That is not a problem for us, however, because the vector product measure only counts matching ones and ignores matching zeros. So the spacer lines will be maximally distinct from any lines containing the text of pages.

```
originFreq200Vectors = Map[getFreqVector[#, freq200] &, originLines];
```

We can look up the vector for the first page (line 3 of *originLines*) like this:

```
originFreq200Vectors[[3]]
```

Now suppose in the course of our research we come across the following page and find it to be quite interesting for whatever reason. We can use the document vector model to find other pages that are most similar to it.

```
originLines[[1487]]
```

The vector product gives us a way of measuring how similar this page is to another page. We see, for example that the page we are interested in has little in common with line 3 (the first page of text) or line 5 (the second page of text).

```
originFreq200Vectors[[1487]].originFreq200Vectors[[3]]
```

```
originFreq200Vectors[[1487]].originFreq200Vectors[[5]]
```

We can **Map** the vector product across all of the vectors for the book and **ListPlot** the results. Note that the maximum similarity is 24, where the vector is multiplied by itself. The next most similar

pages share 14 high-frequency words, then a few pages share 12 words, and so on. The **Tooltip** command allows us to hover over a particular point with the mouse and see the exact value of that point.

```
ListPlot[Tooltip[Map[originFreq200Vectors[[1487]].# &, originFreq200Vectors]],
 PlotRange → Full, ImageSize → Full,
 AxesLabel → {"Line Num", "Dot[Line 1487, Line Num]"}]
```

The following function allows us to find the line numbers of *n* other lines with the highest similarity to the one we are interested in. It is derived and explained in the 'Programming with *Mathematica*' section below.

```
findSimilarLines[freqvs_, line_, n_] :=
 Rest[MaximalBy[Thread[Rule[Range[Length[freqvs]], freqvs]],
    freqvs[[line]].#[[2]] &, n + 1]][[All, 1]]
```

If we ask for the pages that are most similar to *originLines[[1487]]* we get the following list.

```
findSimilarLines[originFreq200Vectors, 1487, 4]
```

You can have a look at each of the results and see what you think. In my opinion the closest match is this one.

```
originLines[[1381]]
```

The example is not perfect, but it is still pretty impressive. Using nothing more than the presence or absence of a small collection of terms, we are able to say that one text is *similar* to another one. Or to put it a different way, if you are interested in one of these texts, chances are that you will find that the other one is *relevant* to your research, too. In fact, there are a number of ways we can improve the performance of the document vector technique without complicating it too much. For one thing, we only used vectors with 200 terms. We might find that we get better performance simply by increasing the length of our vectors, which is a simple modification to make. Another thing that can improve performance, sometimes greatly, is to use a more sophisticated scheme than coding for the presence or absence of a given term with a one or zero. In the next section we will learn about a measure called TF-IDF which can be used to weight document vectors.

Summary

The document vector model allows us to represent any text as a multidimensional vector of numbers. Using techniques from linear algebra we can then determine the similarity or difference of a pair of texts by performing computations on their respective vectors. These measurements of similarity serve as the basis for a variety of tasks in information retrieval. The example that we considered was one of finding texts related to a sample text ('show me more like this one'). In the next section we will develop related methods that allow us to solve other kinds of problems, such as determining what a text is about, or automatically grouping texts into categories.

Generalizing the Examples

TF-IDF: Measuring the importance of a word

In order to figure out what a text is about, we need some way of measuring the importance of a given word in the text. We will use a measure called *TF-IDF*, which stands for 'term frequency-inverse document frequency'. There are a lot of different ways to calculate TF-IDF, but they all capture the basic intuition that there are three categories of word.

1. A word that occurs on (practically) every page doesn't tell you anything special about a particular page. It is a stopword.

2. A word that occurs only once or is sprinkled a few times through the whole document or corpus of documents probably can be safely ignored.
3. A word that occurs a number of times on one page but is relatively rare in the document or overall corpus plays an important role in figuring out what that page is about.

We will simply introduce one method of calculating TF-IDF here and leave some of the other options until the 'Further Exploration' section below.

Determining document frequencies for the whole book

As before, we will continue to work with Darwin's *Origin of Species*. We start by splitting the text into a list of words and converting all to lowercase. We then determine the word frequencies for the whole document. Instead of using the **WordCounts** command, we will demonstrate another method of doing this. (This particular method was the subject of one of the exercises in Chapter 1.)

We start with the **StringSplit** command, which breaks a string apart based on a pattern of separators.

```
StringSplit["This is a test", Whitespace]
```

Note that splitting on whitespace leads to problems with punctuation.

```
StringSplit[
 "This is a test. It includes, among other things, punctuation.", Whitespace]
```

A better way is to use anything that is not a word character to split the string.

```
StringSplit["This is a test. It includes, among other things, punctuation.",
 Except[WordCharacter] ..]
```

Now that we know enough to use **StringSplit** to convert the text of *Origin* into a list of lowercase words, we can do so. The total word count that we get when we do this is a bit different from the output of the **WordCount** command. This method (incorrectly) treats 'H.M.S.' as three separate words rather than a single one, for example.

```
originWholeList =
  Map[ToLowerCase, StringSplit[origin, Except[WordCharacter] ..]];
```

```
Length[originWholeList]
```

```
Short[originWholeList, 3]
```

Next we need to count the number of tokens of each word type. We use the **Tally** command for this. Tally outputs a list of lists. Each element is a list consisting of a type, followed by the number of tokens of that type.

```
Tally[{a, b, c, a, b, d, a, e, c, b, d, b}]
```

The output of **Tally** is alphabetized by word types, but we usually prefer to have results sorted by inverse frequency. We use the **Sort** command with a pure ordering function to do this. (Pure functions and sorting were discussed in the 'Programming with *Mathematica*' sections of Chapters 1 and 2, but you don't really have to understand them to follow this example. You just have to know that the function basically tells **Sort** in what order to put the results it outputs).

```
docFreq = Sort[Tally[originWholeList], #1[[2]] > #2[[2]] &];
```

```
Short[docFreq, 6]
```

In Chapter 2 we learned how to retrieve a list of stopwords using the **WordData** command. Once we have such a list, we can look at the fifty most common words in *Origin* that are not stopwords. Looking at these words gives us a pretty good sense of what the book as a whole is about.

```
stopwords = WordData[All, "Stopwords"];
```

```
Take[Select[Take[docFreq, 200], Not[MemberQ[stopwords, First[#]]] &], 50]
```

Suppose, however, that we wish to know what a *portion* of the book is about. In the 'Basic Examples' section of this chapter, we simply used the presence or absence of one of these terms on a given page as an indicator of what that page was about. Here we will use TF-IDF instead.

What is Chapter 9 of *Origin of Species* about?

To be concrete, let's say that we are interested in trying to figure out what Chapter 9 of *Origin* is about. As shown previously, we can use pattern matching to pull out Chapter 9 as a string.

```
ch9 = StringCases[origin, "CHAPTER 9. " ~~ Shortest[x__] ~~ "CHAPTER" → x][[1]];
```

Next we split the Chapter 9 string into separate words and convert each to lowercase using the same method we just used for the whole book.

```
ch9List = Map[ToLowerCase, StringSplit[ch9, Except[WordCharacter] ..]];
```

```
Short[ch9List, 6]
```

The **Union** command allows us to create a bag of words. These are all of the distinct word types that appear in Chapter 9 of *Origin*. There are 1640 of them.

```
ch9Terms = Union[ch9List];
```

```
Length[ch9Terms]
```

```
Short[ch9Terms, 6]
```

The *term frequencies* are simply the frequencies of all of the words that appear in Chapter 9. Again, we are using the method we just developed. We don't have to worry about removing stopwords, because their TF-IDF score will be relatively low.

```
ch9TermFreq = Sort[Tally[ch9List], #1[[2]] > #2[[2]] &];
```

```
Short[ch9TermFreq, 6]
```

The *docFreq* list that we generated earlier contains document frequencies for the whole text of *Origin*. Here, we only want document frequencies for the terms that appear in Chapter 9. First we need a way of determining whether something appears in a list or not. The **MemberQ** command does that.

```
MemberQ[{a, b, c}, b]
```

```
MemberQ[{a, b, c}, e]
```

```
MemberQ[ch9Terms, "physical"]
```

```
MemberQ[ch9Terms, "physician"]
```

Now we need to go through the *docFreq* list and pull out document frequencies for the terms that appear in Chapter 9. We will use the **Select** command to do this. **Select** takes a list and a function describing the elements that are to be selected. Since each element in the *docFreq* list is itself a list with two items (the word type and its frequency)...

```
Short[docFreq]
```

... we use a pure function based on **MemberQ** to do the matching based on the word type. (Again, you don't have to understand pure functions to follow the example. The list *ch9DocFreq* is a subset of the *docFreq* list containing document frequencies for only those terms that appear in Chapter 9).

```
ch9DocFreq = Select[docFreq, MemberQ[ch9Terms, #[[1]]] &];
```

```
Short[ch9DocFreq, 6]
```

Looking at *ch9TermFreq* and *ch9DocFreq* we can see, for example, that the word 'the' appears 695 times in Chapter 9 of *Origin*, and 10141 times in the book as a whole.

Calculating TF-IDF

Now we have term frequencies for every term that appears in Chapter 9, and we have document frequencies for every term that appears in Chapter 9. So we have enough information to compute the TF-IDF for every term that appears in Chapter 9. Here is one way to calculate it.

```
tfidf[termfreq_, docfreq_, numdocs_] :=
 Log[termfreq + 1.0] Log[numdocs / docfreq]
```

Let's try to calculate the TF-IDF for one term that appears in Chapter 9, 'tapir'.

```
MemberQ[ch9Terms, "tapir"]
```

We can retrieve its term frequency and document frequency. The fact that they are the same number tells us that 'tapir' occurs three times in *Origin*, and all three instances are in Chapter 9.

```
Cases[ch9TermFreq, {"tapir", _}]
```

```
Cases[ch9DocFreq, {"tapir", _}]
```

We are comparing Chapter 9 to the other chapters in *Origin*. Since there are 15 of them, we set the number of documents to 15. The TF-IDF score for "tapir" is

```
tfidf[3, 3, 15]
```

Let's compare that score to the score for a stopword

```
Cases[ch9TermFreq, {"the", _}]
```

```
Cases[ch9DocFreq, {"the", _}]
```

```
tfidf[695, 10 141, 15]
```

Very much lower! Let's compare the score for 'tapir' with the score for a word that occurs frequently throughout *Origin*, 'selection'.

```
Cases[ch9TermFreq, {"selection", _}]
```

```
Cases[ch9DocFreq, {"selection", _}]
```

```
tfidf[4, 383, 15]
```

If we were comparing *Origin* with another text (maybe one that was not about evolution), then the TF-IDF score for 'selection' would be relatively high. But when we compare one part of *Origin* with another part, the TF-IDF score for 'selection' may be relatively low in some cases, and high in others, depending on which part of the text we are focusing on.

The importance of using logarithms

The **Log** function plays an important role in determining the TF-IDF. As we saw in Chapter 1—and as we can see if we look at *docFreq* above—some terms occur much more frequently than others. In our TF-IDF calculation we need to prevent these more frequent terms from dominating the less frequent ones. So we use the **Log** command to take the natural logarithms of the two parts of our equation (the TF and the IDF). First, a quick review of what logarithms do.

A logarithm makes it easier to compare numbers that differ greatly in scale. Suppose, for example,

you have a list of values like the following

```
{3., 870., 75., 88 430., 18., 12 007.,
 239 021., 7689., 51 982., 11., 785 324., 360 400.}
```

If you plot those on a linear scale, the handful of very large values dominate your view. Most of the other values appear so close to zero that it is hard to discriminate among them in a meaningful way.

```
ListPlot[{3., 870., 75., 88 430., 18.,
  12 007., 239 021., 7689., 51 982., 11., 785 324., 360 400.},
 Filling → Axis, PlotRange → Full, AxesLabel → {None, "Raw Value"}]
```

If we take the **Log** of each of our values we get the following list.

```
Log[{3., 870., 75., 88 430., 18., 12 007.,
  239 021., 7689., 51 982., 11., 785 324., 360 400.}]
```

If we plot this list, we get the following figure. The largest value is still the 11th one, the second largest is still the 12th, and so on. The smallest value (the 1st) corresponds to the smallest value in our original list. We can now use the same graph to compare small values as easily as large ones.

```
ListPlot[{1.09861, 6.76849, 4.31749, 11.39, 2.89037,
  9.39325, 12.3843, 8.94755, 10.8587, 2.3979, 13.5739, 12.795},
 Filling → Axis, PlotRange → Full, AxesLabel → {None, "Logs"}]
```

Drawing the figure like this, with a linear scale on the x axis, hides the fact that the difference between the two smallest numbers (3 and 11) is nowhere near as large as the difference between the two largest (360,400 and 785,324). It is more useful to draw the figure with a logarithmic scale on the x axis. That is what the **ListLogPlot** command does. Note that we give it our original data as input. Compare this figure to the previous two.

```
ListLogPlot[{3., 870., 75., 88 430., 18.,
  12 007., 239 021., 7689., 51 982., 11., 785 324., 360 400.},
 Filling → Axis, PlotRange → Full, AxesLabel → {None, "Value"}]
```

When we look at this figure, we can see that the first element is a bit larger than 2 (it is 3) and the 10th element is a bit larger than 10 (it is 11). We can also see that the largest element is a bit less than 1 million (10^6) and the second largest is a bit less than half a million. Using a graph with a log scale, we can make rapid comparisons between values that a vastly different in scale: the 10th element is about a hundred times smaller than the 2nd one, a thousand times smaller than the 8th one and ten thousand times smaller than the 4th one.

So what would the TF-IDF measures look like without logarithms? This is the revised formula

```
(termfreq + 1.0) ( numdocs / docfreq)
```

Here are the measures for 'tapir', 'the' and 'selection'

```
(3 + 1.0) ( 15 / 3)
```

```
(695 + 1.0) ( 15 / 10 141)
```

```
(4 + 1.0) ( 15 / 383)
```

Now a very frequent term (like the stopword 'the') has a higher TF-IDF measure than a highly relevant, but less frequent term (like the word 'selection'). This is not the result we want, hence the use of **Log** in our formula. (We used the natural logarithm (base e) in our code, but the base doesn't matter.)

Computing TF-IDF measures for every word in Chapter 9

We are in a position to compute the TF-IDF measure for all of the terms that Darwin used in Chapter 9. We can build up a function to do so step-by-step. Let's start by determining what inputs our function will need and what kind of output it will return. In order to compute TF-IDF, we will need a

list of terms appearing in the chapter. We have already created that for Chapter 9 and assigned it to the symbol *ch9Terms*. We will need a list of term frequencies (got it: *ch9TermFreq*). And we will need a list of document frequencies for terms appearing in Chapter 9 (done: *ch9DocFreq*). The output that we want our function to return will include a list of terms in order of descending TF-IDF score. This is what our function looks like so far.

```
computeTFIDF[termlist_, tflist_, dflist_] :=
 Module[{outlist, ...},
  outlist = {};
  ...
  Return[outlist]]
```

For each term in the term list, we want to compute the TF-IDF and **Append** it to the list we are going to output. We can loop through each of the terms with a **Do** loop.

```
Do[Print[t * 2], {t, {1, 2, 3}}]
```

Here is an example of **Append** in action.

```
Append[{a, b, c}, z]
```

Our function now looks like this

```
computeTFIDF[termlist_, tflist_, dflist_] :=
 Module[{outlist, ...},
  outlist = {};
  Do[
  ...
  outlist=Append[outlist, {t...tfidf[...]}],
   {t,termlist}];
  Return[outlist]]
```

For each term, we need a way to get its TF and DF so we can calculate its TF-IDF. Setting aside the problem of how we are going to do this for a moment, we can make room for it in our function.

```
computeTFIDF[termlist_, tflist_, dflist_] :=
 Module[{outlist,tf,df},
  outlist = {};
  Do[
  tf=...;
  df=...;
  outlist=Append[outlist, {t,tf,df,tfidf[tf,df,15.0]}],
   {t,termlist}];
  Return[outlist]]
```

To get the TF and DF for a given term, we will use the **Cases** command. As we have seen, it takes a list and a pattern, and returns each element of the list that matches the pattern.

```
Cases[{a, b, 123, c, 45, d, e, 67}, _Integer]
```

```
Cases[{a, b, 123, c, 45, d, e, 67}, _Symbol]
```

If we want to use **Cases** to extract something from a nested list (i.e., a list of lists), we can use a named pattern and a rule, as shown below.

```
Cases[{{a, 123}, {b, 45}, {6, c}, {d, 7}, {89, e}}, {x_, _Integer} → x]
```

If we only wanted the first element, we could select it with **Part**.

```
Cases[{{a, 123}, {b, 45}, {6, c}, {d, 7}, {89, e}}, {x_, _Integer} → x][[1]]
```

Since our lists of term frequencies and document frequencies are nested lists, this is the version of **Cases** that we need for the function that we are developing. The final version is

```
computeTFIDF[termlist_, tflist_, dflist_] :=
 Module[{outlist, tf, df},
  outlist = {};
  Do[
   tf = Cases[tflist, {t, x_} → x][[1]];
   df = Cases[dflist, {t, x_} → x][[1]];
   outlist = Append[outlist, {t, tf, df, tfidf[tf, df, 15.0]}],
   {t, termlist}];
  Return[outlist]]
```

Now that we have this function, we can do the computation and save the results.

```
ch9TFIDF = Sort[computeTFIDF[ch9Terms, ch9TermFreq, ch9DocFreq], #1[[4]] > #2[[4]] &];
```

Looking at the TF-IDF scores

The output of the *computeTFIDF* function includes a lot of information. For each term there is a list consisting of the term, its TF, DF, and TF-IDF. Here are the first ten entries in the TF-IDF list. We have used the **Grid** command to lay everything out in a table, and the **Prepend** command to add headers for each column. The *Alignment→Right* option tells *Mathematica* we want the columns to be right justified.

```
Grid[Prepend[ch9TFIDF[[1 ;; 10]], {"Term", "TF", "DF", "TF-IDF"}], Alignment → Right]
```

The fifty terms in Chapter 9 with the highest TF-IDF score are listed below. Take a moment to study them.

```
Take[ch9TFIDF, 50][[All, 1]]
```

Consulting *Wikipedia,* WordData and *Wolfram Alpha*

Note that the highest scoring terms by this measure—such as 'teleostean'—are nowhere near the most frequent terms Darwin uses in *Origin*. In fact he only uses 'teleostean' three times in the whole book. But all three times are in Chapter 9. The same is true of 'tapir' and 'pebbles'.

```
Cases[docFreq, {"teleostean", _}]
```

```
kwic[ch9, "teleostean", 3]
```

```
Cases[docFreq, {"tapir", _}]
```

```
kwic[ch9, "tapir", 3]
```

```
Cases[docFreq, {"pebbles", _}]
```

```
kwic[ch9, "pebbles", 3]
```

From the KWIC display we can see that 'teleostean' has something to do with fishes. If we are not clear on what this term means, we can use the **WikipediaData** command to look it up.

```
WikipediaData["teleost", "SummaryPlaintext"]
```

If we just want the definition of the word 'teleost' we can use **WordData**.

```
WordData["teleost", "Definitions"]
```

For more exotic requests, we can call **WolframAlpha**. The command below, for example, provides us with an interactive graph of the frequency history of the word 'teleost' over more than 400 years, based on a sample of a million English-language volumes from *Google Books*. The usage of this word peaked in the early 1970s.

```
WolframAlpha["teleost", IncludePods → "BookMatchFrequency:WordData",
 AppearanceElements → {"Pods"}, PodStates → {"Hyponym:WordData__More"}]
```

So what is Chapter 9 of *Origin* about?

Look again at the high TF-IDF terms from Chapter 9: 'teleostean', 'pebbles', 'decay', 'conchologists', 'wear', 'tear', 'mineralogical', 'levels', 'grinding', 'gravel', 'sand', 'sedimentary', 'wears', 'wearing', 'watermark', 'tidal' ... The general sense is of geology, water, erosion, and fossils. When we look at Darwin's own summary for the chapter, we can see that this impression is accurate.

```
StringTake[ch9, 538]
```

The *term frequency-inverse document frequency* (TF-IDF) measure gives us one way of determining whether a particular term plays an important role in a page, chapter or other part of a text. In our example we used terms with high TF-IDF to figure out what Chapter 9 of *Origin of Species* is about. When used with a group of documents (a *corpus*), TF-IDF can help to automatically determine what each document is about and whether or not it is *relevant* to a particular search query.

Programming with *Mathematica*

Coding texts as vectors

In the document vector model, we want to represent a given text in terms of the presence or absence of high-frequency keywords. The simplest implementation is simply to count each presence as one and each absence as zero, as we did in the 'Basic Examples' section above.

The first thing we need to know is which terms that appear on our page are also columns in our vector. We can figure that out by using the **Intersection** command. Here are all the terms from *originLines[[3]]* which are also in *freq200*.

```
Intersection[line3words, freq200]
```

We need to find the position of each one of those words in the *freq200* list, because that is going to correspond to its column number in the vector. The term "america" is the 86th column of the vector.

```
Position[freq200, "america"]
```

We can write this as a pure function and apply it to the string "america" to get the same result.

```
(Position[freq200, #] &) ["america"]
```

If we **Map** the pure function across our list of words in the intersection, we get all the positions.

```
Map[Position[freq200, #] &, Intersection[line3words, freq200]]
```

This information isn't quite in the most useful format, however. For one thing, we want to pull each position out of the nested lists it is in.

```
First[{{86}}]
```

```
First@First[{{86}}]
```

So now our pure function looks like this, and returns a flat list of positions.

```
Map[First@First@Position[freq200, #] &, Intersection[line3words, freq200]]
```

We are going to do one more thing that we will need in a moment. We are going to return a list of rules, where each position points to the number one. Compare this expression with the previous one.

```
Map[(First@First@Position[freq200, #] → 1) &, Intersection[line3words, freq200]]
```

OK, why did we do that? There is a command in *Mathematica* called **ConstantArray** which we can

use to get a list of constant values. Here is how we generate a list of 10 zeros. We are actually going to need 200 of them but that is an easy change to make.

```
ConstantArray[0, {10}]
```

Say we wanted to change the 3rd and 7th positions from zeros to ones. We can do that with the **ReplacePart** command.

```
ReplacePart[ConstantArray[0, {10}], {3 → 1, 7 → 1}]
```

So having gotten our positions into the form of a list of rules, we can use them with ReplacePart and ConstantArray to generate our 200-dimensional vector.

```
ReplacePart[ConstantArray[0, {200}], Map[
  (First@First@Position[freq200, #] → 1) &, Intersection[line3words, freq200]]]
```

At this point, all we need to do is roll everything up into a function. This is the function that appears in the example above.

```
getFreqVector[textstr_, freq_] :=
 ReplacePart[ConstantArray[0, {Length[freq]}],
   Map[(First@First@Position[freq, #] -> 1) &,
    Intersection[DeleteStopwords@ToLowerCase@TextWords[textstr], freq]]]
```

Finding similar vectors

In our 'Basic Examples' section we used the vector product (**Dot** command) to determine how similar two vectors are. Here is what the **Dot** command is calculating when you give it two vectors:

```
Dot[{a, b, c}, {d, e, f}]
```

This product can also be written as

```
{a, b, c}.{d, e, f}
```

We stored a vector for each line of *originLines* in *originFreq200Vectors*. So if we wanted to compute the similarity of *originLines[[1487]]* and *originLines[[3]]* all we had to do was compute this vector product:

```
originFreq200Vectors[[1487]].originFreq200Vectors[[3]]
```

Note that we get the same result regardless of the order.

```
originFreq200Vectors[[3]].originFreq200Vectors[[1487]]
```

We can also write this vector product using a pure function which we then apply to another vector to get a result. Since we are multiplying the same two vectors here, we get the same result.

```
(originFreq200Vectors[[1487]].# &)[originFreq200Vectors[[3]]]
```

Now suppose we want to search through a list of items and return the largest value *by some measure*. The **MaximalBy** command does precisely that. Here we are saying that we want to search through *originFreq200Vectors* and multiply each one by *originFreq200Vectors[[1487]]*. The largest value will be returned.

```
MaximalBy[originFreq200Vectors, originFreq200Vectors[[1487]].# &]
```

We could ask it to return the largest two values by adding an argument to the command.

```
MaximalBy[originFreq200Vectors, originFreq200Vectors[[1487]].# &, 2]
```

This is going in the right direction, but it isn't exactly what we need. What we really want is to return the line number in *originLines* associated with the most similar vector. (Or the line numbers, if we are asking for multiple results.)

So we need to pair each vector with its line number. In this chapter we learned we can do this by using the **Thread** and **Rule** commands.

```
Thread[Rule[Range[3], {a, b, c}]]
```

To get a list of each line number paired with its associated vector we will use this expression:

```
Thread[Rule[Range[Length[originFreq200Vectors]], originFreq200Vectors]]
```

But now each element in the list we are testing with **MaximalBy** is not a vector, it is a rule with a number pointing to a vector. So we have to modify our pure function so it looks like this:

```
originFreq200Vectors〚1487〛.#〚2〛 &
```

If we want to get the top four matches, we have to ask for five of them and throw away the first one. (That is because the text vector is maximally similar to itself). Since results are returned in descending order, we can use the **Rest** command to take the rest of the result list after we have dropped the first element. Here are the four closest matches to *originLines〚1487〛*.

```
Rest[MaximalBy[
    Thread[Rule[Range[Length[originFreq200Vectors]], originFreq200Vectors]],
    originFreq200Vectors〚1487〛.#〚2〛 &, 5]][[All, 1]]
```

Now we can roll everything up into the function that was used above in the text.

```
findSimilarLines[freqvs_, line_, n_] :=
 Rest[MaximalBy[Thread[Rule[Range[Length[freqvs]], freqvs]], freqvs〚line〛.#[[2]] &, n
+ 1]][[All, 1]]
```

Linking an ordered word list to the text string

In Chapter 2 we considered the problem of finding a pair of words near one another in a text. Here we consider a related problem. When we work with ordered lists of normalized words (like *ch9* or *originWholeList*) we occasionally come across something of interest. We might be looking at n-grams, using a KWIC display or using pattern matching to search. When this happens we would like to consult the original text, since it is easier to read than an ordered list of normalized words. So given an ordered word list, it would be nice to have a function that finds and displays that part of the text string and some material before and after.

Suppose we see something that piques our curiosity.

```
{"after", "a", "certain", "unknown", "number", "of", "generations",
 "some", "bird", "had", "given", "birth", "to", "a", "woodpecker"}
```

Our function should take a list like this and return the substring from the whole text string that contains it (*origin* in our case). We start by turning our ordered word list into a string. If we simply try to **StringJoin** the words in the list, there won't be any spaces in the resulting string.

```
StringJoin[{"after", "a", "certain", "unknown", "number", "of", "generations",
  "some", "bird", "had", "given", "birth", "to", "a", "woodpecker"}]
```

Instead we need to use the **StringRiffle** command which intersperses each word with a space.

```
StringRiffle[{"after", "a", "certain", "unknown", "number", "of", "generations",
  "some", "bird", "had", "given", "birth", "to", "a", "woodpecker"}]
```

Next we try using **StringPosition** with the *IgnoreCase* option set to *True*.

```
StringPosition[origin, StringRiffle[
  {"after", "a", "certain", "unknown", "number", "of", "generations", "some",
   "bird", "had", "given", "birth", "to", "a", "woodpecker"}], IgnoreCase → True]
```

It doesn't work, however. The problem is that there must be some punctuation in the original string which was lost in the normalization process. The phrase "some bird had given birth" sounds reason-

ably unique and is unlikely to contain any internal punctuation (or diacritics). If we try coverting this subset to a string and matching it, we are successful.

```
StringPosition[origin,
 StringRiffle[{"some", "bird", "had", "given", "birth"}], IgnoreCase → True]
```

Whenever possible, however, our goal is to let the computer do the work. We shouldn't have to hunt through our list for some subset that is probably going to match. What we want is some way of matching our ordered list to a string, allowing for the possibility of deletion of characters (in the case of punctuation marks) and mutation (in the case of diacritics).

In fact, *Mathematica* has a command called **LongestCommonSubsequence** which finds the largest such matching string.

```
LongestCommonSubsequence[origin, StringRiffle[
  {"after", "a", "certain", "unknown", "number", "of", "generations", "some",
   "bird", "had", "given", "birth", "to", "a", "woodpecker"}], IgnoreCase → True]
```

If we use **StringPosition** to find this, it will return a result.

```
StringPosition[origin,
 LongestCommonSubsequence[origin, StringRiffle[{"after", "a", "certain",
    "unknown", "number", "of", "generations", "some", "bird", "had", "given",
    "birth", "to", "a", "woodpecker"}], IgnoreCase → True], IgnoreCase → True]
```

We would like to display the whole match, not just part of it. The length of the whole list we are trying to match gives us a window of sorts: we know that all of our words have to be within a certain distance from one another. In this case our window is 87 characters.

```
StringLength[
 StringRiffle[{"after", "a", "certain", "unknown", "number", "of", "generations",
   "some", "bird", "had", "given", "birth", "to", "a", "woodpecker"}]]
```

If we take that many characters to either side of the range that **StringPosition** returns, we will be sure to capture all the words in our list and a little bit of context to either side. The function below takes a text, an ordered list of words and an integer reflecting how many lines of additional context we would like to see, and returns a tabbed display showing the text string that contains the ordered list of words.

```
orderedWordListFind[txt_, wordlist_, lines_] :=
 Module[{within},
  within = (80 * lines) + StringLength[StringRiffle[wordlist]];
  TabView[Map[StringTake[txt, {#[[1]] - within, #[[2]] + within}] &,
    StringPosition[txt, LongestCommonSubsequence[txt,
      StringRiffle[wordlist], IgnoreCase → True], IgnoreCase → True]]]]
```

Here it is in action

```
orderedWordListFind[origin,
 {"after", "a", "certain", "unknown", "number", "of", "generations",
  "some", "bird", "had", "given", "birth", "to", "a", "woodpecker"}, 1]
```

If you study the code for the function, you will see that we used **Map** inside of it. In the example above, the ordered list of normalized words that we are interested in only appears once in *Origin*. But that is not necessarily going to be the case, as the next example shows.

```
orderedWordListFind[origin, {"our", "domestic", "productions"}, 2]
```

Mathematica Commands to Review

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration

- Append (GE)
- ConstantArray (BE)
- Do (GE)
- Dot (BE)
- Equal (FE)
- FormulaData (BE)
- FormulaLookup (BE)
- FullForm (BE)
- Greater (BE)
- Histogram (BE)
- Intersection (BE), (PM)
- ListLogPlot (GE)
- Log (GE)
- LongestCommonSubsequence (GE)
- MaximalBy (BE)
- Take (GE)
- Tally (GE)
- Thread (BE)
- Tooltip (BE)

Exercises

1. Using the **WolframAlpha** command we were able to generate an interactive graph showing the word frequency history of the word 'teleost' over more than 400 years, based on a million-volume sample from *Google Books*. In that case, we asked *Wolfram Alpha* to send us formatted *pods*. We can also request computable data which is designed to be subject to further processing in *Mathematica*. Try retrieving the word frequency history data for 'tapir' and 'teleost' (or any other pair of terms) and plotting them in the same graph. The following tutorial may be useful

   tutorial/DataFormatsInWolframAlpha

ch06

Chapter 06: Internet Sources

Overview

We have been using a text from *Mathematica*'s built-in **ExampleData** to develop facility with some basic digital research methods, but almost all sources for a research project will probably come from the internet in one form or another. In this chapter we develop methods for retrieving online sources (including downloading in bulk), processing webpages, creating indexes and working with files on our local machine. We also explore techniques for manually archiving versions of webpages so that we can cite them without worrying that they will change in the future.

Basic Examples

URL: Uniform resource locator

If you can visit a site with a web browser, you can write a computer program to automatically retrieve information from that website. I should say at the outset, however, that you have to be careful not to violate the terms of service of the site in question. Many sites, especially those that provide journal articles or other scholarly resources, *expressly forbid* the use of computer programs to harvest or process their resources automatically. Always read the terms of use for the site and make sure to ask your university librarian or a company representative if you are in any doubt about what you can or cannot do. That said, I have found that many resource providers are quite supportive of text mining and other digital research projects and some will go out of their way to provide access if you let them know what you would like to do, and why.

When you want to visit a site in your browser, you have to provide a URL (uniform resource locator). URLs identify a web resource, tell you where the resource is located and how to access it. They have a common format which looks like this

```
http : // darwin - online.org.uk
```

The first part of the URL

```
http
```

is called the 'protocol identifier' or 'scheme'. It allows you to specify how information is going to be exchanged between your computer and the computer which has the resource (the server). The hypertext transfer protocol (HTTP) is the standard protocol for the World Wide Web.

The next part of the URL is the Internet domain. This is a human- and machine-readable name that will ultimately be translated into an IP address (more on this below).

```
darwin - online.org.uk
```

We can get computable data about domain names, including a list of properties.

darwin-online.org.uk (internet domain) ✓

```
EntityTypeName[ darwin-online.org.uk (internet domain) ]
```

```
EntityProperties["InternetDomain"]
```

How many monthly visitors does the *Darwin Online* site receive?

```
Entity["InternetDomain", "http://darwin-online.org.uk"]["SiteMonthlyVisitors"]
```

Each device connected to the Internet has an IP address (Internet Protocol) which ultimately indicates where it is. This consists of four numbers, each ranging from 0 to 255, separated by dots. We can look up the IP address for the *Darwin Online* site with the following command.

```
Entity["InternetDomain", "http://darwin-online.org.uk"]["DNSLookupName"]
```

IP addresses are tied to particular physical locations, and we can request this information in *Mathematica,* too. Despite the "uk" top-level domain of this particular address, we can see that the site is actually hosted in Ashburn, Virginia in the United States.

```
EntityTypeName[ 50.16.190.67 (IP address) ]
```

```
EntityProperties["IPAddress"]
```

```
Entity["IPAddress", "50.16.190.67"]["HostLocation"]
```

```
CityData[ Ashburn (city) , "Coordinates"]
```

If we wish, we can request more information about the site from *Wolfram Alpha*.

http://darwin-online.org.uk

Retrieving information from the web

Mathematica has a variety of commands that allow us to retrieve web pages or elements from them. We can use the **Import** command with the "Elements" property to see what kind of information is available for a particular web page.

```
Import["http://darwin-online.org.uk", "Elements"]
```

Once we know what elements are available, we can request the one(s) that we want. This command retrieves only the title of the page, returning a string.

```
Import["http://darwin-online.org.uk", "Title"]
```

We can also request the text of a page and a list of links to other pages. These are both very useful when we are automatically processing large numbers of websites.

```
viewData@Import["http://darwin-online.org.uk", "Plaintext"]
```

```
viewData@Import["http://darwin-online.org.uk", "Hyperlinks"]
```

To get the most out of an online resource, we will typically want to retrieve some of the other elements, especially the source code for the page. These options are discussed in detail below.

Files and directories

Once we have retrieved online information, we usually want to store a copy on our local machine. *Mathematica* provides access to the files and folders of your operating system with a number of commands. I am using OS X, so your results may be a bit different on Windows or Linux.

To see the current working directory (i.e., folder), use **Directory**. My working directory is currently set to my OS X home directory.

```
Directory[]
```

You can change the directory with **SetDirectory**. Here I am changing my working directory to my *Downloads* folder. Everywhere in the text that I use a file path like */Users/wjt/Downloads/* you should substitute an appropriate path on your own system.

```
SetDirectory["/Users/wjt/Downloads/"]
```

```
Directory[]
```

To get a list of files in the directory, use **FileNames**. (In OS X and Linux, which are both based on the Unix operating system, a directory can contain hidden *dotfiles*. These have names beginning with a dot, and do not usually appear when you view the directory with a file viewer like Finder.)

```
FileNames[]
```

Optionally, you can include a pattern which limits the listing to matching file names. This shows me the PDF files in my *Downloads* folder.

```
FileNames["*.pdf"]
```

You can test to see if a directory or file already exists.

```
FileExistsQ["expressionemoti01darwgoog.pdf"]
```

```
DirectoryQ["/Users/wjt/Downloads/"]
```

If a directory does not exist, you can create it.

```
If[Not[DirectoryQ["/Users/wjt/Downloads/temp"]],
 CreateDirectory["/Users/wjt/Downloads/temp"]]
```

Now when we check the directory listing, we see that *temp* has been created.

```
FileNames[]
```

We can delete the directory if we no longer need it.

```
DeleteDirectory["/Users/wjt/Downloads/temp"]
```

Downloading a text file

One of the most basic digital research tasks is to retrieve an online resource and save a local copy. Suppose we want a copy of *Vestiges of the Natural History of Creation*, written by Robert Chambers and published anonymously in 1844. We find a Project Gutenberg e-text at the *Internet Archive*.

https://archive.org/details/vestigesofthenat07116gut

Project Gutenberg books are nice for digital reseach because they are raw text (ASCII) files that have been typed in by a human being. As such they tend to contain fewer errors than texts that were created by optical character recognition and the file format is easier to handle than PDF (both of these other options are discussed in later chapters).

We can download the resource directly to our local machine. To get a file from the *Internet Archive* you have to provide a different URL than the one you visit in your web browser, and you have to know the name of the text file that you want. The download URL begins like this

```
https : // archive.org / download /
```

(The HTTPS scheme is a secure version of HTTP.) You need to add a path to the resource that you want. If it helps, you can think of this like specifying a directory on the server. The part of the path that is specific to this particular resource at the *Internet Archive*, *vestigesofthenat07116gut*, is called its *identifier*.

```
https : // archive.org / download / vestigesofthenat07116gut /
```

We know the identifier, but we don't yet know what the name of the file is. So we will use the **Import** command to get all the hyperlinks on the page and **Select** to retrieve the one that contains a *.txt* file extension.

```
Select[Import["https://archive.org/download/vestigesofthenat07116gut/",
  "Hyperlinks"], StringContainsQ[#, ".txt"] &]
```

Now that we know what the file name is, we can use **URLSave** to save a copy of the file directly to our own computer. The **FileNameJoin** command adds a path to the name of the local file that we are creating. Instead of using the somewhat cryptic filename from the Internet Archive, I am going to use the more explicit directory name as my file name. I am also going to be explicit about which directory I want to put it in, even though that is my current working directory. There is less chance of error that way.

```
URLSave["https://archive.org/download/vestigesofthenat07116gut/vstc10.txt",
 FileNameJoin[{"/Users/wjt/Downloads/", "vestigesofthenat07116gut.txt"}]]
```

I can now check to make sure the file is there

```
FileNames["*.txt", "/Users/wjt/Downloads/"]
```

Using the **Import** command we can load the contents of the file into memory and assign it to a symbol. Using **Short** we can see that the file has some header information before the text begins,

and some footer information after it ends.

```
vestigesFull = Import["/Users/wjt/Downloads/vestigesofthenat07116gut.txt"];
```

```
Head[vestigesFull]
```

```
StringLength[vestigesFull]
```

```
Short[vestigesFull, 6]
```

Before we analyze the text in any way, we need to clip off the header and footer. Most Project Gutenberg texts have lines indicating the beginning and ending of the text proper with three asterisks. If we **Map** the **StringTake** command over the text, we can see these two lines. The **StringPosition** command shows us where in the text those lines are.

```
StringTake[vestigesFull, #] & /@
 StringPosition[vestigesFull, Shortest["*** " ~~ "START" | "END" ~~ __ ~~ " ***"]]
```

```
StringPosition[vestigesFull, Shortest["*** " ~~ "START" | "END" ~~ __ ~~ " ***"]]
```

The **StringDrop** command lets us delete ranges of characters from a string. Here we use it twice, to delete the header and footer. Note that we need to calculate how many characters to trim off the end.

```
(StringLength[vestigesFull] - 476 729) + 1
```

```
vestigesTrimmed = StringDrop[StringDrop[vestigesFull, -13 342], 1280];
```

```
Short[vestigesTrimmed, 10]
```

Comparing high frequency words of *Vestiges* and *Origin*

Now that we have the text of *Vestiges*, we can do the same kinds of textual analysis we did with *Origin*. We can determine the most frequently occuring words, for example. Not surprisingly, they are stopwords. Some of the least frequent terms appear to be (and are) very large numbers, from an interesting discussion of Babbage's algorithmic view of the geological record.

```
vestigesWordFreqs = WordCounts[vestigesTrimmed, IgnoreCase → True];
```

```
Short[vestigesWordFreqs, 10]
```

We can also compare the high frequency terms in *Vestiges* with those in *Origin*. The following function plots two columns with arrows joining terms that appear in both. Its workings are explained in the 'Programming with *Mathematica*' section below.

```
listMatchGraph[left_, right_, lbl_] :=
 Module[{idxl, idxr, leftidx, rightidx, lself, rself, lcoords, rcoords, gr},
  idxl = ToString /@ Range[Length[left]];
  idxr = ToString /@ Range[Length[right]];
  leftidx = StringRiffle /@ Thread[List[idxl, left]];
  rightidx = StringRiffle /@ Thread[List[idxr, StringJoin[#, " "] & /@ right]];
  lself = Thread[Rule[leftidx, leftidx]];
  rself = Thread[Rule[rightidx, rightidx]];
  lcoords = Thread[Rule[leftidx, Map[{-8, #} &, -Range[Length[leftidx]]]]];
  rcoords =
   Thread[Rule[rightidx, Map[{8, #} &, -Range[Length[rightidx]]]]];
  gr = DeleteCases[Thread[
     Rule[leftidx, rightidx[[First@First@(Position[right, #] /. {} → {{0}})]] & /@
       left]], _ → List];
  Show[GraphPlot[Join[lself, rself, gr], VertexLabeling → True,
    VertexCoordinateRules → Join[lcoords, rcoords], SelfLoopStyle → None,
    EdgeRenderingFunction → ({Gray, Arrow[#1 + {{1.6, 0}, {-1.6, 0}}]} & ),
    VertexRenderingFunction → ({Black, Text[#2, #1]} &),
    ImageSize → Large], PlotLabel → lbl, LabelStyle → {12}]]
```

The most frequently occuring terms in each book (not including stopwords) are plotted in order of decreasing frequency.

```
listMatchGraph[DeleteStopwords@Keys[vestigesWordFreqs][[1 ;; 130]],
 DeleteStopwords@Keys[originWordFreqs][[1 ;; 136]],
 "Vestiges vs. Origin\nHigh Frequency Terms"]
```

Looking at this figure, we can see, for example, that although both books are about species, Chambers focuses more on time and animals and Darwin more on forms and varieties. The idea of natural selection, which plays such an important role in *Origin*, is not to be found in *Vestiges*.

Batch downloading files

Since we can write code to download a single file directly to our local machine, nothing prevents us from automatically downloading files in bulk. As an example, we will download the works of Huxley that appear in the Project Gutenberg collection of the *Internet Archive*.

The *Internet Archive* has an advanced search page at the following URL. The *.php* file extension tells us that this page is written in the server scripting language PHP. Such pages often have *dynamic* content, that is, they are updated every time that someone requests the page. If you visit this page in a web browser, you can fill out a number of fields indicating the title, creator, collection, date range or other information you are interested in. When you submit your query, the system will respond with matching texts, images, movies, audio files, or whatever.

```
http : // archive.org / advancedsearch.php
```

We can also use the *Mathematica* **URLBuild** command to enter our query directly as a URL. The following example requests all of the materials in the Project Gutenberg collection that were created by Thomas Henry Huxley. The only field that we are interested in is the *identifier*. We ask for 50 rows of information beginning with the first page. (If we need to request more information we can make subsequent calls for *"page"→2*, *"page"→3*, and so on). We ask that the output be returned as *comma separated values* (CSV).

```
URLBuild[{"http://archive.org/", "advancedsearch.php"},
 {"q" → "creator:\"Huxley, Thomas Henry, 1825-1895\" AND collection:Gutenberg",
  "fl[]" → "identifier", "rows" → 50, "page" → 1, "output" → "csv"}]
```

The above command does nothing but create the URL corresponding to the search that we want, however. In order to actually get the information we use the **URLFetch** command. What is returned

is one long string, so we use **StringSplit** to break the results into a list. Each identifier has a quotation mark character at the beginning and end which we remove with **StringTrim**. The first element that is returned is simply the word “identifier” and we don’t need that, so we get rid of it with **Rest**.

```
huxleyGutenbergIAIdentifiers =
 Rest@StringTrim[StringSplit[URLFetch@URLBuild[{"http://archive.org/",
       "advancedsearch.php"}, {"q" → "creator:\"Huxley, Thomas
          Henry, 1825-1895\" AND collection:Gutenberg",
       "fl[]" → "identifier", "rows" → 50, "page" → 1, "output" → "csv"}]], "\""]
```

Since there are 47 identifiers and we asked for a maximum of 50, we know we don’t need to request any more pages of results.

```
Length[huxleyGutenbergIAIdentifiers]
```

Before we request the actual files, we are going to need a place to put them. I check to see if a directory exists on my machine, and create it if not.

```
If[Not[DirectoryQ["/Users/wjt/Downloads/huxley"]],
 CreateDirectory["/Users/wjt/Downloads/huxley"]]
```

Now we need a function that downloads a file to our directory, given an Internet Archive identifier and the name of the directory where we want to put it. This function also lets us specify that we only want files with a particular extension (*.txt* in this case). The code is a generalization of the method that we used above to download *Vestiges*. Note that we check to make sure that the file hasn’t already been downloaded, so we avoid doing unnecessary work.

```
downloadIAFile[idstr_, dirstr_, ext_] :=
 Module[{baseurl, target, fname},
  Pause[2];
  baseurl = "https://archive.org/download/";
  target = FileNameJoin[{dirstr, idstr <> ext}];
  If[FileExistsQ[target],
   PrintTemporary["File " <> target <> " was already downloaded"],
   PrintTemporary["Downloading " <> target];
   fname =
    Select[StringCases[URLFetch[baseurl <> idstr, "Content"],
      Shortest["href=\"" ~~ x__ ~~ "\""] → x], StringContainsQ[#, ext] &];
   If[fname ≠ {},
    URLSave[baseurl <> idstr <> "/" <> First@fname, target],
    PrintTemporary["No text file for " <> target]]]]
```

The **Pause** command causes *Mathematica* to wait for two seconds. The reason that we are doing this is because we are going to make a number of requests to the *Internet Archive* servers, and it is polite not to ask for things too quickly. When a person visits a web page it usually takes them at least a few seconds to follow a link to the next page. When a computer program visits a site, you don’t want it to request dozens of pages within a few milliseconds of one another. The **PrintTemporary** command gives us some feedback while the download is in progress. When the download is complete, the messages will be deleted from the notebook.

Let's try requesting a single file. We can just choose the first one from our list of identifiers.

```
First[huxleyGutenbergIAIdentifiers]
```

```
downloadIAFile["hasisadrasadvent02633gut",
 "/Users/wjt/Downloads/huxley", ".txt"]
```

We check to make sure the file appeared in the proper directory.

```
SetDirectory["/Users/wjt/Downloads/huxley"];
FileNames["*.txt"]
```

Since that worked, we can now download all of the Huxley files from the Gutenberg collection by

mapping our download function across our list of identifiers. We have about fifty files to download and each takes at least a few seconds so this job will take a few minutes.

```
Map[downloadIAFile[#, "/Users/wjt/Downloads/huxley", ".txt"] &,
  huxleyGutenbergIAIdentifiers];
```

Indexing a set of files for basic searching

Now that we have downloaded all of the files, we would like to be able to search for information across the whole group. In order to do this, we need to know whether or not a particular term appears in a particular file, and if so, where. We are going to build an *index*, a data structure to keep track of which terms appear in which files. For each term and each file, we need to know the word position(s) where that term occurs.

To make the job easier, we start by creating an association of texts. This takes a minute or so. Note that the code tests to see whether there is a file called *huxleyTexts.assoc.txt*. The first time you run the code that file will not exist. When the association is created, however, the **Put** command is used to store a copy of the association to the folder where the Huxley texts are located. If you were to quit *Mathematica* and then start it up and rerun the code, it would discover that the *huxleyTexts.assoc.txt* file exists and load the association using the **Get** command rather than trying to recreate it from scratch. Using **Put** and **Get** to store the results of computations can save you a lot of time.

```
SetDirectory["/Users/wjt/Downloads/huxley"];
If[FileExistsQ["huxleyTexts.assoc.txt"],
  huxleyTexts = Get["huxleyTexts.assoc.txt"],
  huxleyTexts =
   Association[Map[# → ToLowerCase@TextWords@Import@# &, FileNames["*.txt"]]];
  Put[huxleyTexts, "huxleyTexts.assoc.txt"]];
```

Number of texts

```
Length@huxleyTexts
```

Number of words in each text

```
Length /@ Values@huxleyTexts
```

Total number of words

```
Total[Length /@ Values@huxleyTexts]
```

Note that we did not trim off the Project Gutenberg headers and footers, so they are inflating our word count a bit, and they will be searchable in our index. We are unlikely to confuse Huxley's own words with the boilerplate, as long as we don't do anything like try to compute word frequencies from our index. If necessary, we could go back and process each file to get rid of the headers and footers, as we did with the trimmed version of *Vestiges*.

Each entry in the *huxleyTexts* association looks like this

```
filename → {each, word, in, the, text, is, listed, in, order ...}
```

The next function takes the association of texts that we just created, and the name of a file, and it creates an index for that file.

```
indexWords[fileassoc_, filename_] :=
 ReplaceAll[GroupBy[
   Select[Thread[List[fileassoc[filename], Range@Length[fileassoc[filename]]]],
    nonStopwordQ[#[[1]]] &], First → Last], x_List → (filename → x)]
```

Let's try running it on one of the files so we can see what kind of output it creates. We can see that the word 'grapes' appears once in this text, at the 7280th word position. The word 'sons' appears twice in this text. Each entry is of the form

```
term → filename → {position1, position2, ...}
```

We can see that the phrase 'Project Gutenberg' appears a number of times both at the beginning of the file (e.g., in word positions 2 and 3) and at the end (in word positions 7284 and 7285).

```
Short[indexWords[huxleyTexts, "thedarwinianhypo02927gut.txt"], 30]
```

We need to keep track of the filenames because we are going to **Merge** all of our separate file indexes into one large one. The code which builds the combined index for all of the files is shown below. Once again, we are testing to see if the index was already created, so we can **Get** it rather than rebuild it. (This isn't always what you want to do. If you change any of the files that you are indexing, or add new ones to the directory, you need to rebuild your index. A more complete strategy would be to check the file modification *timestamps* and make sure that none are more recent than the timestamp for the index. We won't worry about that refinement right now, however.)

```
SetDirectory["/Users/wjt/Downloads/huxley"];
If[FileExistsQ["huxleyIndex.assoc.txt"],
  huxleyIndex = Get["huxleyIndex.assoc.txt"],
  huxleyIndex = Merge[Map[indexWords[huxleyTexts, FileNameJoin[{# <> ".txt"}]] &,
      huxleyGutenbergIAIdentifiers], Identity];
  Put[huxleyIndex, "huxleyIndex.assoc.txt"]];
```

We can now look up a word in the index like this

```
huxleyIndex["agnostic"]
```

The information will be more useful to us if we can see each use of the word in context, so we turn to that next.

Search terms in context

Given a filename and a word position, we can look up the context in the *huxleyTexts* association.

```
searchTermContext[textassoc_, wordpos_, filename_] :=
 textassoc[filename][[wordpos - 3 ;; wordpos + 3]]
```

```
searchTermContext[huxleyTexts, 3437, "collectedessaysv15905gut.txt"]
```

The function below applies *searchTermContext* to every match and displays the results in a scrollable window. It is explained in the 'Programming with *Mathematica*' section below. On the righthand side of each row, the name of the file and the word position for the search term are shown in green text in curly braces.

```
displaySearch[textassoc_, indexassoc_, term_] :=
 Module[{results},
  results =
   Flatten[
    Thread[List[Values@huxleyIndex[term][[#]], Keys@huxleyIndex[term][[#]]] & /@
     Range[Length[Keys@huxleyIndex[term]]], 1];
  Framed[Pane[Style[TableForm[Append[searchTermContext[textassoc, #[[1]], #[[2]]],
        Style[{#[[2]], #[[1]]}, Darker@Green]] & /@ results],
     Small], {Full, 200}, Scrollbars → True]]
 ]
```

```
displaySearch[huxleyTexts, huxleyIndex, "agnostic"]
```

Summary

Any online resource that can be retrieved with a web browser can be accessed with computer programs, and processing of large numbers of digital sources can be readily automated (although the terms of use on some sites prohibit you from doing so). Once you have retrieved a large batch

of files and stored them locally, you want to have some way of searching them. Creating a simple text index is one possibility, but by no means the only one. We will explore some other options below. We will also look in more detail at the way that text sources are marked up for presentation online.

Generalizing the Examples

Markup languages: HTML

When information is presented online, the content provider needs some way of indicating how different elements are supposed to look, how they are related to one another, and, at least to some extent, what they mean. All of this *metadata* can be provided in the form of *tags* which are added to a text file. The process of adding such information is called 'marking up' a file, hence 'markup languages'. This is easier to understand once you play with it a bit, so we will work through a few simple examples using the **EmbeddedHTML** command.

In the Hypertext Markup Language (HTML) you can indicate that some text is to be emphasized with the *em* tag. Emphasized text is usually rendered with italic font.

```
EmbeddedHTML["This <em>italicized</em> word is in italics"]
```

The **EmbeddedHTML** command takes an HTML string and creates a button. When you click on the button above in your *Mathematica* notebook, it opens the string in your default web browser. There you should see the text with the word 'italicized' in italics.

Tags that are used to indicate a span of material have a beginning tag (like <*em*>) and an ending tag (like <*/em*>). The paragraph tag works in a similar fashion to the emphasis tag.

```
EmbeddedHTML["<p>This the <em>first</em>
   paragraph.</p><p>This the <em>second</em> paragraph.</p>"]
```

Note that tags have to be *properly nested*. That means that the following is OK

```
< outer > < inner > ... < / inner > < / outer >
```

But that this is not OK

```
< outer > < inner > ... < / outer > < / inner >
```

Some tags are used in a single location. The line break tag is an example. Note that the syntax for these kind of tags looks like <*br/*>. Compare this example with the previous one.

```
EmbeddedHTML["<p>This the <br/><em>first</em>
   paragraph.</p><p>This the <br/><em>second</em> paragraph.</p>"]
```

A string that begins with an ampersand and ends with a semicolon represents a special character in HTML. Suppose, for example, that you want to put a left angle bracket on your page. Since that character is used in every tag, you need to have some way of telling the web browser that you are *mentioning* the character rather than *using* it to define a tag. In HTML, you can represent a left angle bracket with *<* (the 'lt' stands for 'less than'). Likewise, if you need to mention an ampersand you can do so with *&* as shown in the example below. The *strong* tag is usually rendered as boldface.

```
EmbeddedHTML["It is <strong>true</strong> that 4&lt;8 &amp; 4&gt;2."]
```

Sometimes special characters are represented with HTML *character codes*. These begin with an ampersand and end with a semicolon. In the middle they have a number sign (#) followed by an integer. Below we will see an example that uses the code *‘* to stand for a left single quotation mark and *’* to stand for a right single quotation mark. It also uses the code *–* to represent what is called an *en dash*, a dash that is the width of the lowercase letter *n*. It is wider than a hyphen and narrower than an *em dash,* which is the width of an uppercase *M*.) You can use

the **FromCharacterCode** command to discover what symbol is being represented.

```
FromCharacterCode[{8216, 8217, 8211}]
```

You can also convert a string containing an HTML character code into an integer with **StringDelete** and **ToExpression**, then use that to look up the character code.

```
ToExpression[StringDelete["&#8216;", Except[DigitCharacter]]]
```

```
FromCharacterCode@
 ToExpression[StringDelete["&#8216;", Except[DigitCharacter]]]
```

The example strings that we have been looking at are HTML fragments. A minimal HTML page needs to include a number of other tags that indicate what kind of document it is, where the HTML begins and ends, a head section that includes the title and other metadata, and a body section that includes the visible body of the page. Here is a more complete HTML page.

```
EmbeddedHTML["<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>"]
```

One of the most useful HTML tags is the anchor tag, which is used to create a hyperlink. Suppose that we want to create a link to the URL *http://www.darwinproject.ac.uk* and we want to label the link "Darwin Correspondence Project". Then the HTML for the hyperlink looks like this

```
< a href =
 "http://www.darwinproject.ac.uk" > Darwin Correspondence Project < / a >
```

The part inside the tag that looks like *href="URL"* is called an *attribute*. Attributes are typically pairs of names and values, and are used to provide more information.

Since we are putting our HTML inside a string to test it with the **EmbeddedHTML** command, we have to *escape* the interior quotation marks by putting backslashes in front of them. This is another example of *mentioning* something rather than *using* it. We don't want *Mathematica* to interpret the inside pair of quotation marks as beginning and ending a string.

```
EmbeddedHTML["<a href=\"http://www.darwinproject.ac.uk\">Darwin
   Correspondence Project</a>"]
```

When you visit a web page with a browser you see the rendered page rather than the HTML (and other web technologies) that lay behind it. But most browsers also allow you to view the page source, so you can see how the web developer(s) accomplished a particular effect. Studying the page source also allows you to write programs that extract information from web pages, a process known as *scraping*. We will explore some examples below.

Manually archiving a web page in the Wayback Machine

Before we do any scraping, however, I want to save a copy of a web page in the Internet Archive's *Wayback Machine*. This site allows you to explore the way that particular web pages looked in the past. (At least approximately; we will defer a more detailed explanation of web archives for the time being.) The Internet Archive automatically archives web sites at regular intervals, but if you have a URL that you want added to the *Wayback Machine* at at particular time, you can go to the following page and paste the URL that you want to archive in the box under "Save Page Now."

```
https://archive.org/web/
```

On July 15, 2015, I saved a copy of this page

```
http://www.darwinproject.ac.uk
```

The archived page is at

```
https://web.archive.org/web/20150715195311/http://www.darwinproject.ac.uk
```

By using the archived page for our example (rather than the live page) we can be sure that the example will continue to work as long as we want it to. Note that other pages that are linked from an archived page will not be automatically archived.

HTML Page source

We have seen that we can use the **Import** command to request the title, hyperlinks and plaintext from a web page. We can also request the HTML source. If you are not already familiar with HTML, spend a few minutes looking at the source for the *Darwin Correspondence Project* homepage as it appeared on July 15, 2015. Notice that some of the tags are already familiar: *!DOCTYPE*, *html*, *head*, *title*, *body*, *a* and *em*. This particular page also uses the web scripting language JavaScript, Cascading Style Sheets (CSS) for formatting, some layout tags like *div* and *span*, and many other elements.

```
Import[
 "https://web.archive.org/web/20150715195311/http://www.darwinproject.ac.uk",
 "Elements"]
```

```
darwinCorrespondenceProjectNews = Import[
   "https://web.archive.org/web/20150715195311/http://www.darwinproject.ac.uk",
   "Source"];
```

```
viewData@darwinCorrespondenceProjectNews
```

If your goal is simply to extract information from the page, you often don't need to understand how the whole thing is formatted. You simply need to identify the tags that appear in the immediate vicinity of the information you want to grab, and use those to define a pattern.

An interactive pattern matching tool

We can take advantage of *Mathematica*'s **Manipulate** command to develop a tool that allows us to interactively define patterns for scraping marked-up files. The pattern matching tool has a window where we can type a pattern and hit the ENTER key to submit it. Underneath are two more windows. The top one shows the HTML source with the matched parts in bold. The bottom one shows the information that would be extracted if we used that pattern. The code is explained in the 'Programming with *Mathematica*' section below.

```
characterPositionComplement[len_, plist_] :=
 Select[Thread[Join[{Prepend[Part[#, 2] & /@ (plist + 1), 1],
     Append[Part[#, 1] & /@ (plist - 1), len]}]], #[[1]] ≤ #[[2]] &]
```

```
highlightCharacterPositions[str_, plist_] :=
 Module[{len = StringLength[str], graylist, boldlist},
  boldlist = Thread[Join[{plist, Bold}]];
  graylist = Thread[Join[{characterPositionComplement[len, plist], Gray}]];
  Return[Row[
    Style[StringTake[str, First[#]], #[[2]] & /@ Sort[Join[boldlist, graylist]]]]]
```

```
patternMatchTool[src_] :=
 Manipulate[
  Module[{pos},
   pos = StringPosition[src, p];
   Column[{Pane[highlightCharacterPositions[src, pos], Scrollbars → True,
       ImageSize → {Full, 200}], Pane[Column[StringTake[src, pos]],
       Scrollbars → True, ImageSize → {Full, 200}]}]],
  {p, "title", InputField[#, FieldSize → {60, 3}, BaseStyle → 12] &}]
```

The tool is set up to match the default pattern “title”. Note how the title tags and attributes are highlighted in the source code. Try entering each of these other patterns to get a feel for how the tool works.

This pattern matches all of the Cascading Style Sheet (CSS) files used by the page:

```
WordBoundary~~WordCharacter..~~".css"
```

This pattern shows us material that has been formatted with HTML comment tags. Note that the last comment was added by the Internet Archive when the page was saved (as were some of the other comments).

```
Shortest["<!--"~~__~~"-->"]
```

This pattern scrapes out the paragraph that describes the website:

```
Shortest["<section class=\"site_description\">"~~__~~"</section>"]
```

This pattern shows us the menu entries under “Themes”.

```
Shortest["id=\"menu-themes-"~~__~~"\""]
```

This pattern shows us all of the parts of the page that contain items used for navigating the site:

```
Shortest["<div class=\"nav-item-container\">"~~__~~"</div>"]
```

And this pattern scrapes the links and titles of the news items:

```
Shortest["<h2 class=\"news_item_title\">"~~__~~"</h2>"]
```

```
patternMatchTool[darwinCorrespondenceProjectNews]
```

Scraping information from a web page

Now suppose we want to pull out the titles of the news items from this page. We can use the last pattern from above as follows:

```
StringCases[darwinCorrespondenceProjectNews,
 Shortest["<h2 class=\"news_item_title\">" ~~ titles__ ~~ "</h2>"] → titles]
```

If we only want the labels from the hyperlinks, we can modify the pattern as shown below. (We have added stuff to the pattern to match the anchor tags).

```
Shortest[
 "<h2 class=\"news_item_title\"><a " ~~ __ ~~ ">" ~~ __ ~~ "</a></h2>"]
```

Then the **StringCases** command becomes

```
StringCases[darwinCorrespondenceProjectNews,
  Shortest["<h2 class=\"news_item_title\"><a " ~~ __ ~~
     ">" ~~ titles__ ~~ "</a></h2>"] → titles] // TableForm
```

Suppose we want to scrape the date that each news item was posted, too. We modify the pattern to capture still more material.

```
StringCases[darwinCorrespondenceProjectNews, Shortest[
    "<h2 class=\"news_item_title\"><a " ~~ __ ~~ ">" ~~ titles__ ~~ "</a></h2>" ~~
     __ ~~ "Posted on " ~~ dates__ ~~ " in"] → {titles, dates}] // TableForm
```

Once you get the hang of it, scraping can be remarkably quick and powerful. One drawback, however, is that scraping is also very brittle. Often the content provider need only make a single change to their website to inadvertently break your scraper. This isn't a problem if you only need to scrape the page once, or once in a while. Just archive a copy in the *Wayback Machine* and scrape the archived copy. If you scrape a site frequently you will find that you need to revise your scrapers from time to time.

Markup languages: XML

Unlike HTML, which uses a limited set of predefined tags to indicate how data should be rendered or displayed, the Extensible Markup Language (XML) uses tags to *describe* data. There are many standards for representing various kinds of metadata with XML, but users are also free to define their own tags. This flexibility, and the fact that XML is both human- and machine-readible, makes it a very powerful way to represent textual and numeric information.

So that we have a short piece of XML to look at, I am going to request information about a biography of Darwin written by Adrian J. Desmond and James R. Moore. To do this, I am using a web service created by the Online Computer Library Center (OCLC) called Classify. If you send the book ISBN in a URL to OCLC Classify, the system will respond with a file of bibliographic information marked up with XML. In order to see the raw XML, I am converting it to a string before passing it off to *viewData*.

```
Import[
 "http://classify.oclc.org/classify2/Classify?isbn=0393311503&summary=true",
 "Elements"]
```

```
biographyXMLstring = StringRiffle[Flatten[Import[
      "http://classify.oclc.org/classify2/Classify?isbn=0393311503&summary=true
        ", "Data"]], "\n"];
```

```
viewData@biographyXMLstring
```

If you aren't already familiar with XML, spend a few moments looking at the file. Note that it begins with a tag that indicates that the file is marked up with XML, *?xml*. The *work* tag contains information about the book, mostly stored in attributes like *author*, *editions*, *format* and *title*. The *author* and *authors* tags contain information about the authors, some of it redundant. The *lc* and *viaf* attributes of the *author* tag hold Library of Congress and OCLC identifiers, respectively. We will make use of these later. The *ddc* tag contains Dewey Decimal call numbers (576.82092). You would use these to look for a copy of the book in most North American public libraries. The *lcc* tag contains Library of Congress call numbers (QH31.D2) which indicate where the book would be filed in most North American academic libraries.

Just as with HTML, we can extract information by using patterns to scrape the XML string. Here is how we would pull out the author names, birth years, LC and VIAF identifiers.

```
StringCases[biographyXMLstring,
  Shortest["<author lc=" ~~ lc__ ~~ "viaf=" ~~ viaf__ ~~ ">" ~~ lname__ ~~ "\n" ~~
     fname__ ~~ Longest[byear : DigitCharacter ..] ~~ a__ ~~ "</author>"] →
   {lname, fname, "b.", byear, "LC", lc, "VIAF", viaf}] // TableForm
```

Symbolic XML

Rather than scraping XML, it sometimes makes more sense to *parse* it, to break it down in a systematic way. When you import XML into *Mathematica* it is converted to a form called 'symbolic XML', which is native to the *Mathematica* language. In symbolic XML, the whole file is represented by an **XMLObject**, and each tag by an **XMLElement**.

A regular XML expression like

```
< tag attribute = "value" > data < / tag >
```

Is converted to a symbolic XML expression of the form

```
XMLElement["tag", {"attribute" → "value"}, {"data"}]
```

Here is the record for Desmond and Moore's Darwin biography, represented in symbolic XML.

```
biographySymbolicXML = Import[
  "http://classify.oclc.org/classify2/Classify?isbn=0393311503&summary=true",
  "XML"]
```

Symbolic XML takes a bit of getting used to, but it is actually pretty easy to extract the information that you want. Suppose you want to pull out the same information that we scraped above. Go into the above output and copy a sample **XMLElement**, then paste it into the notebook:

```
XMLElement["author", {"lc" → "n78041786", "viaf" → "93867795"},
 {"Moore, James R. 1947- [Author]"}]
```

We are going to use this **XMLElement** as the basis for a pattern that can extract all author details from the XML. Wrap the expression in a **Cases** command. (We need to include *Infinity* as the level specification for our command because the information is nested deeply in the **XMLObject**).

```
Cases[biographySymbolicXML,
 XMLElement["author", {"lc" → "n78041786", "viaf" → "93867795"},
  {"Moore, James R. 1947- [Author]"}], Infinity]
```

Next we have to replace each piece of specific information with a named pattern. That way, instead of matching only the record for James Moore, our **Cases** statement will match all authors.

```
Cases[biographySymbolicXML,
  XMLElement["author", {"lc" → lc_, "viaf" → viaf_}, {a_}] →
   {a, "LC", lc, "VIAF", viaf}, Infinity] // TableForm
```

Here are a few more examples of pulling information from symbolic XML. The title of the work

```
Cases[biographySymbolicXML,
 XMLElement["work", {__, "title" → title_}, {__}] → title, Infinity]
```

The number of editions

```
Cases[biographySymbolicXML,
 XMLElement["work", {__, "editions" → eds_, __}, {__}] → eds, Infinity]
```

The most frequently used Library of Congress call number

```
Cases[biographySymbolicXML, XMLElement["lcc", {},
   {XMLElement["mostPopular", {__, "sfa" → lcc_}, {}], __}] → lcc, Infinity]
```

RSS Feeds

One case where XML parsing is very handy is in dealing with RSS feeds. RSS (Really Simple Syndication) is an XML format for storing information about frequently changed information on a website. A blog, for example, will typically have an RSS feed that you can subscribe to using a *feed reader*. When you check your feed reader, it contacts the site, checks to see if there is a blog post that you haven't read, and, if so, downloads some information about that post. Many news sites, too, use RSS feeds for new stories; libraries often use them to advertise books that have recently been purchased; retailers use them for new product information, and so on.

The *New York Times* has RSS feeds for a variety of topics that are covered frequently in the newspaper. On July 15, 2015, I archived a copy of their RSS feed for Darwin to the Wayback Machine.

The Darwin RSS feed link is

```
http://topics.nytimes.com/top/reference/timestopics/people/d/charles_robert_darwin/index.html?rss=1
```

and the link in the Wayback Machine is

```
https://web.archive.org/web/20150715200540/http://topics.nytimes.com/top/reference/timestopics/people/d/charles_robert_darwin/index.html?rss=1
```

We can use this link to explore some of the things we can do with RSS. The **Import** command, for example, shows us which elements we can request.

```
Import[
 "https://web.archive.org/web/20150715200540/http://topics.nytimes.com/top/
   reference/timestopics/people/d/charles_robert_darwin/index.html?rss=1",
 "Elements"]
```

If we just want to read the feed (as we would with a feed reader), we can actually open it as a new *Mathematica* notebook with the **CreateDocument** command.

```
CreateDocument[Import[
  "https://web.archive.org/web/20150715200540/http://topics.nytimes.com/top/
    reference/timestopics/people/d/charles_robert_darwin/index.html?rss=1",
  "RSS"]]
```

If we want to scrape the feed, we can import it as a string.

```
nytDarwinSource = Import[
    "https://web.archive.org/web/20150715200540/http://topics.nytimes.com/top/
      reference/timestopics/people/d/charles_robert_darwin/index.html?rss=1"
      , "Source"];
```

```
Head[nytDarwinSource]
```

The StringCases command lets us pull out individual entries.

```
nytDarwinItems =
  StringCases[nytDarwinSource, Shortest["<item>" ~~ __ ~~ "</item>"]];
```

```
nytDarwinItems[[1]]
```

Here is a little function to replace the HTML codes for ampersands and angle brackets with characters, delete everything in angle brackets, then replace all other HTML codes with blank space.

```
cleanHTMLMarkup[str_] :=
 StringReplace[
  StringDelete[StringReplace[str, {"&amp;" → "&", "&lt;" → "<", "&gt;" → ">"}],
   Shortest["<" ~~ __ ~~ ">"]], Shortest["&" ~~ __ ~~ ";"] → " "]
```

When we clean out the HTML markup, we see that it makes the remaining text much easier to read.

```
cleanHTMLMarkup["&lt;a
   href=\"http://www.nytimes.com/2015/05/16/opinion/it-is-in-fact-rocket-
   science.html?partner=rssnyt&amp;emc=rss\"&gt;&lt;img
   src=\"http://static01.nyt.com/images/2015/05/16/opinion/16Mlodinow/16
   Mlodinow-thumbStandard.jpg\" border=\"0\" height=\"75\"
   width=\"75\" hspace=\"4\" align=\"left\"/&gt;&lt;/a&gt;Why
   do we reduce great discoveries to epiphany myths?"]
```

Here is a function that scrapes information from the RSS feed and displays it in a format of our choosing. Note that we also include a link that will automatically open in our web browser if we want

to read the full story. Since RSS feeds deal with material that changes frequently, there is no guarantee that links will continue to work indefinitely (especially since we archived the feed on the Wayback Machine). If you are monitoring a feed for up-to-date news, however, this is nice to have.

```
prettyPrintItem[it_] :=
 Module[{title, author, pubdate, desc, link},
  title = StringCases[it, Shortest["<title>" ~~ t__ ~~ "</title>"] → t];
  author = StringCases[it, Shortest["<author>" ~~ a__ ~~ "</author>"] → a];
  pubdate = StringCases[it, Shortest["<pubDate>" ~~ pd__ ~~ "</pubDate>"] → pd];
  desc = StringCases[it, Shortest["<description>" ~~ __ ~~ "</description>"]];
  link = StringCases[it, Shortest["<link>" ~~ l__ ~~ "</link>"] → l];
  Column[{If[title ≠ {}, Text@Style[cleanHTMLMarkup@First@title, Bold]],
    Text[If[author ≠ {}, First@author, "Anon"] <>
      ", " <> If[pubdate ≠ {}, First@pubdate, "nd"]],
    Text@Style[cleanHTMLMarkup@First@desc, Medium], Hyperlink[
     Style["Read in web browser", {Medium, FontFamily → Times}], link],
    ""}]]

prettyPrintItem[nytDarwinItems[[8]]]
```

This function gets all of the items in the feed, formats them, and displays them in a version of the *viewData* window.

```
prettyPrintItems[items_] :=
 Framed[
  Pane[Column[prettyPrintItem /@ items], {Automatic, 200}, Scrollbars → True]]

prettyPrintItems[nytDarwinItems]
```

We can also scrape specific information from an RSS feed. Here are all of the links

```
StringCases[nytDarwinSource, Shortest["<link>" ~~ __ ~~ "</link>"]]
```

We can grab one of the links and display the text in a window.

```
Framed[Pane[Text@Style[StringRiffle[cleanHTMLMarkup /@ StringCases[Import[
         "http://opinionator.blogs.nytimes.com/2014/11/30/evolution-and-the-
           american-myth-of-the-individual/?partner=rssnyt&amp;emc=rss
           ", "Source"],
       Shortest["<p class=\"story-body-text" ~~ __ ~~ "</p>"]]],
    Medium], {Automatic, 200}, Scrollbars → True]]
```

Programming with *Mathematica* (Under Development)

```
listMatchGraph[left_, right_, lbl_] :=
 Module[{idxl, idxr, leftidx, rightidx, lself, rself, lcoords, rcoords, gr},
  idxl = ToString /@ Range[Length[left]];
  idxr = ToString /@ Range[Length[right]];
  leftidx = StringRiffle /@ Thread[List[idxl, left]];
  rightidx = StringRiffle /@ Thread[List[idxr, StringJoin[#, " "] & /@ right]];
  lself = Thread[Rule[leftidx, leftidx]];
  rself = Thread[Rule[rightidx, rightidx]];
  lcoords = Thread[Rule[leftidx, Map[{-8, #} &, -Range[Length[leftidx]]]]];
  rcoords =
   Thread[Rule[rightidx, Map[{8, #} &, -Range[Length[rightidx]]]]];
  gr = DeleteCases[Thread[
     Rule[leftidx, rightidx[[First@First@(Position[right, #] /. {} → {{0}})]] & /@
       left]], _ → List];
  Show[GraphPlot[Join[lself, rself, gr], VertexLabeling → True,
    VertexCoordinateRules → Join[lcoords, rcoords], SelfLoopStyle → None,
    EdgeRenderingFunction → ({Gray, Arrow[#1 + {{1.6, 0}, {-1.6, 0}}]} & ),
    VertexRenderingFunction → ({Black, Text[#2, #1]} &),
    ImageSize → Large], PlotLabel → lbl, LabelStyle → {12}]]
```

```
displaySearch[textassoc_, indexassoc_, term_] :=
 Module[{results},
  results =
   Flatten[
    Thread[List[Values@huxleyIndex[term][[#]], Keys@huxleyIndex[term][[#]]] & /@
     Range[Length[Keys@huxleyIndex[term]]], 1];
  Framed[Pane[Style[TableForm[Append[searchTermContext[textassoc, #[[1]], #[[2]]],
        Style[{#[[2]], #[[1]]}, Darker@Green]] & /@ results],
     Small], {Full, 200}, Scrollbars → True]]
 ]
```

Given a length and a list of character positions, return a list consisting of character ranges outside the first positions (all of the anomalous cases return pairs where first element is greater than second)

```
characterPositionComplement[len_, plist_] :=
 Select[Thread[Join[{Prepend[Part[#, 2] & /@ (plist + 1), 1],
     Append[Part[#, 1] & /@ (plist - 1), len]}]], #[[1]] ≤ #[[2]] &]
```

```
characterPositionComplement[87, {{2, 2}, {17, 31}, {44, 49}}]
```

```
characterPositionComplement[87, {{1, 2}, {17, 31}, {44, 49}}]
```

```
characterPositionComplement[87, {{1, 3}, {17, 31}, {44, 87}}]
```

```
characterPositionComplement[87, {{1, 13}, {9, 31}, {24, 87}}]
```

Given a string and a list of character positions, render the positions in bold and the rest of the string in gray

```
highlightCharacterPositions[str_, plist_] :=
 Module[{len = StringLength[str], graylist, boldlist},
  boldlist = Thread[Join[{plist, Bold}]];
  graylist = Thread[Join[{characterPositionComplement[len, plist], Gray}]];
  Return[Row[
    Style[StringTake[str, First[#]], #[[2]]] & /@ Sort[Join[boldlist, graylist]]]]]
```

Pattern has Head Pattern

```
FullForm[Hold[StringPosition["AABBBAABABBCCCBAAA", _ ~~ x_]]]
```

Mathematica Commands to Review (Under Development)

- BE: Basic Examples, GE: Generalizing the Examples, PM: Programming with *Mathematica*, FE: Further Exploration
- CityData (BE)
- CreateDirectory (BE)
- CreateDocument (GE)
- DeleteDirectory (BE)
- Directory (BE)
- DirectoryQ (BE)
- EmbeddedHTML (GE)
- Except (GE)
- FileExistsQ (BE)
- FileNameJoin (BE)
- FileNames (BE)
- FromCharacterCode (GE)
- Get (BE)
- GroupBy (BE)
- ImageSize (GE)
- Import (BE)
- Infinity (GE)
- Manipulate (GE)
- Merge (BE)
- Not (BE)
- Pause (BE)
- ReplaceAll (BE)
- SetDirectory (BE)
- StringDelete (GE)
- StringDrop (BE)
- StringTrim (BE)
- TableForm (GE)
- ToExpression (GE)
- URLBuild (BE)
- URLFetch (BE)
- URLSave (BE)
- XMLElement (GE)
- XMLObject (GE)

ch07

Chapter 07: Image Processing (Under Development)

Overview

The digital research techniques that we have learned up to this point mostly rely on having access to machine-readable text. In addition to human-readable formats like raw text, HTML and XML, machine-readable text can also be stored in thousands of different file formats like Adobe's Portable Document Format (PDF), Microsoft *Word* documents or *Excel* spreadsheets, and so on. *Mathematica* can import many of these formats natively (see the documentation page listing all formats). It is also possible for text to be stored in the form of a digital picture, a *page image*, that was created by scanning or photographing a physical page. If the text on these page images was printed or typed, it can often be converted to machine-readable text by a process known as Optical Character Recognition (OCR). At present, handwritten text cannot be easily extracted from page images, although that may change in the near future. In this chapter we start by doing some work with page images, then we turn to techniques that allow us to work with and process digital images more generally.

Basic Examples

Page images in a PDF

If you are working with traditional printed or typed sources, you can create page images by *digitizing* your sources with a scanner or photographing them with a digital camera. People who do archival research now typically return from the archives with (tens of) thousands of digital images of documents. Managing these effectively is an important part of digital research. Rather than scan paper documents, we will download some page images to work with, but the basic principles are the same.

Let's begin with a PDF. The *Darwin Online* site has a link to a paper on coral islands written by Darwin, edited by D. R. Stoddart and published in the *Atoll Research Bulletin* in 1962.

```
http://darwin-online.org.uk/converted/pdf/1962_CoralIslands_F1576.pdf
```

Rather than use the original, I archived a copy in the Wayback Machine on July 28, 2015. If we use that copy, we don't have to worry about the file changing on the original site.

```
https://web.archive.org/web/20150728202603/http://darwin-
online.org.uk/converted/pdf/1962_CoralIslands_F1576.pdf
```

PDFs can have multiple layers. What you see when you open the PDF in *Acrobat* or *Acrobat Reader* are the page images. In addition, the PDF can have a layer of searchable (that is, machine-readable) text associated with each page. The easiest way to see if a particular PDF has this layer is to open it in *Acrobat* and try to search for a word you can see on the page. If search doesn't work, the file is missing a text layer. That is the case for the Darwin paper on coral islands.

In *Mathematica*, we begin by using **Import** and requesting the elements associated with the file.

```
Import[
 "https://web.archive.org/web/20150728202603/http://darwin-online.org.uk/
   converted/pdf/1962_CoralIslands_F1576.pdf", "Elements"]
```

If the file had a text layer, we could use **Import** to get it, but this fails, giving us an empty string.

```
viewData[Import[
  "https://web.archive.org/web/20150728202603/http://darwin-online.org.uk/
    converted/pdf/1962_CoralIslands_F1576.pdf", "Plaintext"]]
```

We can, however, request the number of pages for the document.

```
Import[
 "https://web.archive.org/web/20150728202603/http://darwin-online.org.uk/
   converted/pdf/1962_CoralIslands_F1576.pdf", "PageCount"]
```

We can also get all of the page images for the document in a list. We check the length of our list to make sure we got all of the pages.

```
coralIslandsPageImages = Import[
   "https://web.archive.org/web/20150728202603/http://darwin-online.org.uk/
     converted/pdf/1962_CoralIslands_F1576.pdf", "Images"];
```

```
Length[coralIslandsPageImages]
```

We can get a sense of the document by using the **ConformImages** command to make all of the pages the same height (this is what the *Tiny* and *Pillarbox* options do). We can see there is a title page, some handwritten pages, and a few maps / figures.

```
ConformImages[coralIslandsPageImages, Tiny, "Pillarbox"]
```

The **TextRecognize** command performs OCR on a page image. Sometimes you have to adjust the scale of the image to get good results. Here I use the **ImageResize** and **Scaled** commands to double the size of the page image for page 2 before doing OCR on it.

```
viewData[TextRecognize[ImageResize[coralIslandsPageImages[[2]], Scaled[2]]]]
```

If you look through the OCR results, you will see that they are good, but by no means perfect. Depending on the quality of the page images, the typeface, lighting, and many other factors, OCR can range from excellent to unusable. Whether or not you can use OCR for your own research depends on the nature of your project.

At present, *Mathematica*'s PDF importing capabilities are also somewhat buggy. If you are having trouble with using PDFs, be sure to do a search on mathematica.stackexchange.com to see if there are other techniques that you can take advantage of.

(Under Development)

The rest of these sections contain working code and the odd comment but have not been written up yet. If you are trying to get something to work and having trouble with it, send me an email.

Visualizing all of the page images for a book

Source for page images is Darwin's *Expression of the Emotions in Man and Animals* (1873)
We're grabbing the zipped file of images

```
downloadIAFile["expressionofem00darw", "/Users/wjt/Downloads", "_jp2.zip"];
```

```
FileNames["/Users/wjt/Downloads/*zip"]
```

A list of filenames you can process

```
Import["/Users/wjt/Downloads/expressionofem00darw_jp2.zip", "FileNames"][[1]]
```

Can import page image without uncompressing folder

```
ImageResize[Import["/Users/wjt/Downloads/expressionofem00darw_jp2.zip",
  "expressionofem00darw_jp2/expressionofem00darw_0175.jp2"], 20]
```

Here doing it with my compressed folders

Get my compressed folders from GitHub

```
URLSave[
 "https://github.com/williamjturkel/Digital-Research-Methods/blob/master/darwin
   -expression/expressionofem00darw_jp2_thumbs.zip?raw=true",
 "/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs.zip"]
```

```
Import[
  "/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs.zip", "FileNames"][[3]]
```

```
Import["/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs.zip",
 "expressionofem00darw_jp2_thumbs/expressionofem00darw_0002_thumb.jpg"]
```

```
URLSave[
 "https://github.com/williamjturkel/Digital-Research-Methods/blob/master/darwin
   -expression/expressionofem00darw_jp2_ocr.zip?raw=true",
 "/Users/wjt/Downloads/expressionofem00darw_jp2_ocr.zip"]
```

```
Import["/Users/wjt/Downloads/expressionofem00darw_jp2_ocr.zip", "FileNames"][[3]]
```

```
StringTake[Import["/Users/wjt/Downloads/expressionofem00darw_jp2_ocr.zip",
  "expressionofem00darw_jp2_ocr/expressionofem00darw_0009_ocr.txt"], 500]
```

Given a zipped folder of jpegs, make a folder of tiny page images - this takes at least 15 mins to run so probably want to make zipped folder of page images available on GitHub or somewhere else (Zenodo?) then comment out this code

```
pageImageThumbs[inzip_, outfolder_] :=
 Module[{filelist, outfile},
  filelist = Import[inzip, "FileNames"];
  If[Not[DirectoryQ[outfolder]], CreateDirectory[outfolder]];
  Do[
   PrintTemporary["Processing " <> ToString[f]];
   outfile = ImageResize[Import[inzip, f], 20];
   Export[FileNameJoin[{outfolder, FileBaseName[f] <> "_thumb.jpg"}], outfile],
   {f, filelist}]
  ]
```

```
pageImageThumbs["/Users/wjt/Downloads/expressionofem00darw_jp2.zip",
"/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs"]
```

```
ImageResize::imginv: Expecting an image or graphics instead of $Failed. >>
```

can start by visualizing from first to last on horizontal axis and darkness on vertical axis

get a list of page thumbnails

```
returnThumbList[ebkt_] :=
 Map[Import, FileNames["*.jpg", ebkt]]
```

```
emotionsThumbs =
  returnThumbList["/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs"];
```

```
First[emotionsThumbs]
```

```
Mean[Mean[ImageData[ColorConvert[First[emotionsThumbs], "Grayscale"]]]]
```

Note that the {10,10} scale each page image; that SetAlphaChannel is used to make the images 40% transparent; that bdata associates page numbers from 1 to 400 something with the mean brightness of that page; that the Opacity[0] is used to make the plot points invisible behind each page image; that Inset puts each page image at the right coordinates

Describe this with floating (empty pages) / sinking (dark, heavy pages) metaphor

```
With[{bdata = Thread[List[Range[Length[emotionsThumbs]],
      Mean[Mean[ImageData[ColorConvert[#, "Grayscale"]]]] & /@ emotionsThumbs]]},
 Graphics[{Opacity[0], Point[bdata],
   MapThread[Inset[SetAlphaChannel[#1, .6], #2, Center, {10, 10}] &,
    {emotionsThumbs, bdata}]},
  Frame → True, ImageSize → Large, AspectRatio → 1,
  FrameLabel → {"Page Number", "Brightness"},
  PlotRange → {{1, Length[emotionsThumbs]}, {0, 1}}]]
```

Going to want to roll up the graphing function

OCR

Given a zipped file of page images create a folder of pages with OCRed text - this one takes about 50 minutes, so will want to Zenodo a folder of OCRed text files

```
pageImageOCR[inzip_, outfolder_] :=
 Module[{filelist, outfile},
  filelist = Import[inzip, "FileNames"];
  If[Not[DirectoryQ[outfolder]], CreateDirectory[outfolder]];
  Do[
   PrintTemporary["Processing " <> ToString[f]];
   outfile = TextRecognize[Import[inzip, f]];
   Export[FileNameJoin[{outfolder, FileBaseName[f] <> "_ocr.txt"}], outfile],
   {f, filelist}]
  ]
```

```
pageImageOCR["/Users/wjt/Downloads/expressionofem00darw_jp2.zip",
"/Users/wjt/Downloads/expressionofem00darw_jp2_ocr"]
```

'British Library' visualization

Vertical axis is size of OCR text file, horizontal axis is JPEG file size

Given a list of files, return a list of file sizes, scaled between 0 and 1

```
scaledFileSizes[flist_] :=
 Module[{fslist, max},
  fslist = Map[FileByteCount, flist];
  max = Max[fslist];
  Return[Map[N[(# / max), 5] &, fslist]]]
```

```
With[{bdata = MapThread[List, {scaledFileSizes[FileNames["*.jpg",
        "/Users/wjt/Downloads/expressionofem00darw_jp2_thumbs"]],
      scaledFileSizes[FileNames["*_ocr.txt",
        "/Users/wjt/Downloads/expressionofem00darw_jp2_ocr"]]}]},
 Graphics[{Opacity[0], Point[bdata],
   MapThread[Inset[SetAlphaChannel[#1, .6], #2, Center, {.01, .01}] &,
    {emotionsThumbs, bdata}]},
  Frame → True, ImageSize → Large, AspectRatio → 1,
  FrameLabel → {"JPEG Thumb File Size", "OCR File Size"},
  PlotRange → {{.7, 1.01}, {-.01, .25}}]]
```

Automatic image extraction

Need to show how to grab page tiffs in the first place

Source for this page image is Darwin's *Expression of the Emotions in Man and Animals* (1873) https://archive.org/details/expressionemoti01darwgoog

```
emotionsPage113GrayscaleImage = ColorConvert[ImageTrim[Import[
      "/Users/wjt/Datasets/darwin-expression-tiffs/expressionemoti01darwgoog_
        0113.tif"], {{175, 170}, {790, 1240}}], "Grayscale"];

emotionsPage113BWImage = MorphologicalBinarize[emotionsPage113GrayscaleImage];
Show[emotionsPage113BWImage, ImageSize → Medium]
```

Find contours

```
emotionsPage113TextRemoved =
  DeleteSmallComponents[ColorNegate[emotionsPage113BWImage]];
Show[emotionsPage113TextRemoved, ImageSize → Medium]
```

Get bounding boxes for morphological components and display results

```
emotionsPage113BoundingBoxes =
 ComponentMeasurements[emotionsPage113TextRemoved, "BoundingBox"]

emotionsPage113Chicken =
  ImageTrim[emotionsPage113GrayscaleImage, emotionsPage113BoundingBoxes[[1, 2]]];

Show[emotionsPage113Chicken, ImageSize → Medium]

HighlightImage[emotionsPage113GrayscaleImage,
 Graphics[{EdgeForm[Orange], Opacity[.3],
   Rectangle @@ emotionsPage113BoundingBoxes[[1, 2]]}]]
```

Roll up into a dynamic / manipulate

OCR on this page - playing with scale a bit changes results, but no obvious win

```
TextRecognize[ImageResize[emotionsPage113GrayscaleImage, Scaled[2]]]
```

Detecting faces

```
emotionsPage216GrayscaleImage = ColorConvert[ImageTrim[Import[
      "/Users/wjt/Datasets/darwin-expression-tiffs/expressionemoti01darwgoog_
        0216.tif"], {{175, 170}, {790, 1240}}], "Grayscale"];

emotionsPage216Faces = FindFaces[emotionsPage216GrayscaleImage];
Map[ImageResize[ImageTrim[emotionsPage216GrayscaleImage, #], 100] &,
 emotionsPage216Faces]
```

Photogrammetry

This can draw on the same tools as georectification - but do some automatic image alignments, too

Load Edinburgh rephotography image **http://www.flickr.com/groups/flickrcommons/discuss/72157613061097398**

```
edinburgh1 = Import[
    "/Users/wjt/Dropbox/Code/mathematica/wjt-presentations/scott_3102127793_0
      e6f260033.jpg"];
edinburgh2 = Import[
    "/Users/wjt/Dropbox/Code/mathematica/wjt-presentations/scott_3231940512_1
      bc20a7cab.jpg"];
{edinburgh1, edinburgh2}
```

Standardized versions

```
standardizeImage[im_] :=
 ImageResize[ColorConvert[im, "Grayscale"], {400}]

edinburghStandardizedImages =
  {standardizeImage[edinburgh1], Lighter[standardizeImage[edinburgh2]]};
```

Use computer vision techniques to find matching *keypoints* in the two images

```
edinburghMatches =
  (ImageCorrespondingPoints[##] &) @@ edinburghStandardizedImages;
```

```
MapThread[Show[#1, Graphics[{Black, Background → Lighter@Orange, FontSize → 16,
      MapIndexed[Inset[#2[[1]], #1] &, #2]}]] &, {edinburghStandardizedImages,
  Join[{edinburghMatches[[1, 1 ;; 3]], edinburghMatches[[2, 1 ;; 3]]}]}]
```

Transform images into the same space, overlay them and create a dynamic image explorer

```
edinburgh3 = ImageAlign[edinburgh1, edinburgh2,
   TransformationClass → "Rigid", Method → Automatic];
Manipulate[ImageCompose[edinburgh1, {edinburgh3, t}], {t, 0, 1}]
```

Georectification

National Library of Scotland

http://maps.nls.uk/os/6inch-england-and-wales/index.html

Original

http://maps.nls.uk/view/102352952#zoom=5&lat=3427&lon=10587&layers=BT

Saved view in Wayback Machine

https://web.archive.org/web/20150529201823/http://maps.nls.uk/view/102352952

Need to use FindGeometricTransformation once three points have been assigned.

```
devonshireCXXIII1856 =
```

```
;
```

```
GeoGraphics[GeoPosition[{50.359858, -4.162152}],
 GeoRange → Quantity[1, "Kilometers"]]
```

```
firestoneBayContemporary =
```

```
;
```

```
ImageDimensions[firestoneBayContemporary]
```

```
DynamicModule[{pt = {{200, 200}, {220, 200}, {240, 200}}},
 {LocatorPane[Dynamic[pt], Show[firestoneBayContemporary],
   {{0, 0}, {524, 525}}], Dynamic[pt]}]
```

```
ImageDimensions[devonshireCXXIII1856]
```

```
devonshireCXXIII1856Resized = ImageResize[devonshireCXXIII1856, 525];
```

```
ImageDimensions[devonshireCXXIII1856Resized]
```

```
DynamicModule[{pt = {{200, 200}, {220, 200}, {240, 200}}},
 {LocatorPane[Dynamic[pt], Show[devonshireCXXIII1856Resized],
   {{0, 0}, {525, 314}}], Dynamic[pt]}]
```

```
FindGeometricTransform[{{34.8, 281.}, {201.2, 335.5}, {383.6, 148.5}},
 {{10., 186.}, {182.8, 239.}, {373.2, 41.5}}]
```

```
ImagePerspectiveTransformation[
 devonshireCXXIII1856Resized, TransformationFunction[
  (0.9618265368889533`  0.00370517784129947` | 24.492571552628835`
   0.01928460940040043994`  0.9654267829359132` | 101.23777227991553`)
                0                  0                      1
  ], DataRange → Full, Padding → 0]

525 - 314

ImageCompose[firestoneBayContemporary, {ImagePerspectiveTransformation[
   devonshireCXXIII1856Resized, TransformationFunction[
    (0.9618265368889533`  0.00370517784129947` | 24.492571552628835`
     0.01928460940040043994`  0.9654267829359132` | 101.23777227991553`)
                  0                  0                      1
    ], DataRange → Full, Padding → 0], .7}, {0, 0}, {0, 0}]
```

Image classification

Use *Mathematica*'s powerful new machine learning features to automatically classify extracted images into photographs and drawings (further subdivision is easy on large datasets).

```
standarizePage[pg_] := ColorNegate[
  MorphologicalBinarize[ColorConvert[pg, "Grayscale"], Method → "MinimumError"]]

identifyImageBoundingBoxes[spg_] :=
 ComponentMeasurements[DeleteSmallComponents[spg],
  "BoundingBox"]

getExtractedImageList[pg_] :=
 Module[{bblist = identifyImageBoundingBoxes[standarizePage[pg]]},
  Return[Map[ImageTrim[pg, #] &, bblist[[All, 2]]]]]

highlightExtractedImages[pg_] :=
 Module[{bblist = identifyImageBoundingBoxes[standarizePage[pg]]},
  Show[pg, Graphics[{EdgeForm[{Orange}],
     Opacity[0], Rectangle @@@ bblist[[All, 2]]}], ImageSize → Small]]

demoImageExtraction[pgimglist_] :=
 Module[{returnlist = {}},
  Do[returnlist = Append[returnlist,
     {i, highlightExtractedImages[i], Last[getExtractedImageList[i]]}],
   {i, pgimglist}];
  Return[returnlist]]
```

Training

Get images extracted from Darwin's *Expression* and use them to train a machine learner. *Mathematica*'s **Classify** 'super-function' chooses an appropriate learning method and features, but can be tuned as necessary.

```
emotionsPageImageList = Map[Import,
   FileNames["*.jpg", {"/Users/wjt/Datasets/darwin-expression-image-jpegs/"}]];

emotionsSortTrainingSet = {emotionsPageImageList[[1]], emotionsPageImageList[[5]],
   emotionsPageImageList[[9]], emotionsPageImageList[[14]],
   emotionsPageImageList[[18]], emotionsPageImageList[[19]],
   emotionsPageImageList[[20]], emotionsPageImageList[[27]],
   emotionsPageImageList[[25]], emotionsPageImageList[[28]],
   emotionsPageImageList[[32]], emotionsPageImageList[[34]]};

Map[Show[#, ImageSize → Tiny] &, emotionsSortTrainingSet]
```

```
emotionsImageSortingClassifier = Classify[
  {emotionsPageImageList[[1]] → "Drawing", emotionsPageImageList[[5]] → "Drawing",
   emotionsPageImageList[[9]] → "Drawing", emotionsPageImageList[[14]] → "Drawing",
   emotionsPageImageList[[18]] → "Drawing", emotionsPageImageList[[19]] → "Photo",
   emotionsPageImageList[[20]] → "Photo", emotionsPageImageList[[25]] → "Photo",
   emotionsPageImageList[[27]] → "Photo", emotionsPageImageList[[28]] → "Photo",
   emotionsPageImageList[[32]] → "Drawing", emotionsPageImageList[[34]] → "Photo"}]
```

Testing

```
emotionsSortTestingSet =
  Complement[emotionsPageImageList, emotionsSortTrainingSet];
```

```
emotionsDemoImageSorting[eximglist_] :=
 Module[{returnlist = {}},
  Do[returnlist = Append[returnlist, {i, emotionsImageSortingClassifier[i]}],
   {i, eximglist}];
  Return[returnlist]]
```

```
TableForm[Partition[
  emotionsDemoImageSorting[ImageResize[#, 90] & /@ emotionsSortTestingSet],
  5, 5, 1, {}], TableSpacing → {5, 2}]
```

Automatically identifying images

```
{Show[emotionsSortTestingSet[[1]], ImageSize → Tiny],
 ImageIdentify[emotionsSortTestingSet[[1]]]}
```

```
{Show[emotionsSortTestingSet[[2]], ImageSize → Tiny],
 ImageIdentify[emotionsSortTestingSet[[2]]]}
```

```
{Show[emotionsSortTestingSet[[8]], ImageSize → Tiny],
 ImageIdentify[emotionsSortTestingSet[[8]]]}
```

This is what **ImageIdentify** thinks that is...

Belgian Sheepdog (dog breed) ✓

```
{Show[emotionsSortTestingSet[[9]], ImageSize → Tiny],
 ImageIdentify[emotionsSortTestingSet[[9]]]}
```

```
{Show[emotionsSortTestingSet[[12]], ImageSize → Tiny],
 ImageIdentify[emotionsSortTestingSet[[12]]]}
```

ch08

Chapter 08: (Under Development)

Overview

This chapter contains working code for example projects that I intend to develop for the next edition. If you are trying to get something to work and having trouble with it, send me an email.

Hanging indent format

```
Style[StringTake[origin, 250], "Text", LineIndent → 1]
```

Convert Roman numerals

```
IntegerString[1842, "Roman"]
```

```
FromDigits["MDCCCXLII", "Roman"]
```

In *Mathematica* 10.2 and later...

```
RomanNumeral[1842]
```

```
FromRomanNumeral["MDCCCXLII"]
```

Date computations

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "BirthDate"]
```

In *Mathematica* 10.2 and later

```
JulianDate[ Sun 12 Feb 1809 ]
```

```
FromJulianDate[2.3818256666666665`*^6]
```

This code comes from the Suggestions Bar

```
▶ month calendar  DateObject[{1809, 2, 11},
                    TimeObject[{23, 59, 60.}, TimeZone → -4.], TimeZone → -4.]
```

Language identification

```
unknownLanguage1 = StringTake[ExampleData[{"Text", "AeneidLatin"}], 500]
```

```
unknownLanguage2 = StringTake[ExampleData[{"Text", "HomerOdysseyGreek"}], 500]
```

```
unknownLanguage3 = StringTake[ExampleData[{"Text", "UNHumanRightsMaori"}], 500]
```

Results aren't very impressive for historical languages (although Catalan is at least a Romance language)

```
LanguageIdentify /@ {unknownLanguage1, unknownLanguage2, unknownLanguage3}
```

As an exercise, try training classifier for Latin, Greek and Maori and see if it does better http://www.wolfram.com/language/gallery/determine-the-language-of-a-text/

First recorded use of a word

```
WolframAlpha["evolved", {{"FirstRecordYear:WordData", 1}, "Plaintext"}]
```

```
WolframAlpha["evolution", {{"FirstRecordYear:WordData", 1}, "Plaintext"}]
```

Word frequency history

```
WolframAlpha["platypus", IncludePods → "BookMatchFrequency:WordData",
 AppearanceElements → {"Pods"}, InputAssumptions → {"*C.platypus-_*Word-"},
 TimeConstraint → {30, Automatic, Automatic, Automatic}]
```

Overall word frequency

We can use computable data from WolframAlpha to write a function that shows us how frequently a word occurred per billion words of text in a given year.

```
wordFrequencyPerBillionWordsPerYear[word_, yr_] :=
 Module[{data, idx},
  data = WolframAlpha[word, {{"BookMatchFrequency:WordData", 1},
     "ComputableData"}, PodStates → {"BookMatchFrequency:WordData__Raw"}];
  If[Not[MissingQ[data]],
   idx = Select[data, #[[1, 1]] == yr &];
   Return[idx[[1, 2, 1]]]]]

wordFrequencyPerBillionWordsPerYear["darwinian", 2005]

wordFrequencyPerBillionWordsPerYear["darwinian", 1959]
```

Wikipedia search popularity

```
WolframAlpha["charles darwin",
 {{"PopularityPod:WikipediaStatsData", 1}, "Content"},
 InputAssumptions → {"*C.wikipedia-_*WikipediaStatsDataPropertyClass-"}]

Short[WolframAlpha["charles darwin",
  {{"PopularityPod:WikipediaStatsData", 1}, "TimeSeriesData"},
  InputAssumptions → {"*C.wikipedia-_*WikipediaStatsDataPropertyClass-"}], 5]
```

Why the spike in Feb 2009? Bicentennial of his birthday

```
PersonData[Interpreter["ComputedPerson"]["Darwin"], "BirthDate"]
```

KWIC revisited

A different method

```
origin7Grams = Partition[ToLowerCase[originWords], 7, 1];

Style[TableForm[Cases[origin7Grams, {_, _, _, "monstrous", _, _, _}]], Small]
```

This version doesn't suffer the problem of overlapping windows

```
Style[
 TableForm[Cases[origin7Grams, {_, _, _, "domestic", _, _, _}][[1 ;; 8]]], Small]
```

Pretty printing a KWIC

This is a bit slow. May want to refactor to make it faster

```
nGramstoKWICDict[ngrams_] :=
 Module[{kwicdict, keyidx},
  kwicdict = {};
  keyidx = Quotient[Length[First[ngrams]], 2] + 1;
  Do[
   kwicdict = Join[kwicdict, {Join[{Extract[k, keyidx], k}]}],
   {k, ngrams}];
  Return[Sort[kwicdict]]]

kwicRow[kwic_] :=
 Module[{tail, len, triple},
  tail = First[Rest[kwic]];
  len = Length[tail];
  triple = {
    StringJoin[Riffle[Take[tail, Quotient[len, 2]], " "]],
    Extract[tail, Quotient[len, 2] + 1],
    StringJoin[Riffle[Take[tail, -(Quotient[len, 2])], " "]]};
  Return[triple]]
```

```
prettyPrintKWIC[kwicdict_, target_] :=
 Module[{sublist},
  sublist = {};
  sublist = Cases[kwicdict, {target, __}];
  Grid[Map[kwicRow, sublist], Frame → True, Alignment → {{Right, Center, Left}}]]

kwicDict = nGramstoKWICDict[origin7Grams];

prettyPrintKWIC[kwicDict, "embryo"]
```

A wordlist version of searching for one element near another

```
alice = ExampleData[{"Text", "AliceInWonderland"}];
aliceWordlist = TextWords[ToLowerCase[alice]];

Flatten[Position[aliceWordlist, "mad"]]

Flatten[Position[aliceWordlist, "hatter"]]

Tuples[{Flatten[Position[aliceWordlist, "mad"]],
  Flatten[Position[aliceWordlist, "hatter"]]}]

Map[Sort, Select[Tuples[{Flatten[Position[aliceWordlist, "mad"]],
    Flatten[Position[aliceWordlist, "hatter"]]}], Abs[#[[1]] - #[[2]]] < 30 &]]

aliceWordlist[[6750 ;; 6789]]

TabView[Map[Column@
    {StringRiffle@aliceWordlist[[Max[0, #[[1]] - 10] ;; Max[0, #[[1]] - 1]]],
     Style[StringRiffle@aliceWordlist[[#[[1]] ;; #[[2]]]], Bold],
     StringRiffle@aliceWordlist[[Min[Length[aliceWordlist], #[[2]] + 1]
        ;; Min[Length[aliceWordlist], #[[2]] + 10]]} &,
  Map[Sort, Select[Tuples[{Flatten[Position[aliceWordlist, "mad"]],
      Flatten[Position[aliceWordlist, "hatter"]]}], Abs[#[[1]] - #[[2]]] < 30 &]]]]

searchNearWordList[wlist_, kw1_, kw2_, within_] :=
 TabView[Map[Column@
     {StringRiffle@wlist[[Max[0, #[[1]] - 10] ;; Max[0, #[[1]] - 1]]],
      Style[StringRiffle@wlist[[#[[1]] ;; #[[2]]]], Bold], StringRiffle@
       wlist[[Min[Length[wlist], #[[2]] + 1] ;; Min[Length[wlist], #[[2]] + 10]]} &,
   Map[Sort, Select[Tuples[{Flatten[Position[wlist, kw1]],
       Flatten[Position[wlist, kw2]]}], Abs[#[[1]] - #[[2]]] < within &]]]]

searchNearWordList[aliceWordlist, "mad", "hatter", 30]

searchNearWordList[aliceWordlist, "march", "hare", 30]
```

A network graph of collocations

Rewrite bigrams as a list of graph edges by using **Rule** and **@@@** then pull out the largest set of connected components.

```
Graph[Rule @@@ originBigramsFreq5Interesting]

Subgraph[Graph[Rule @@@ originBigramsFreq5Interesting],
 WeaklyConnectedComponents[Graph[Rule @@@ originBigramsFreq5Interesting]][[1]],
 VertexLabels → "Name"]
```

```
With[{g = Graph[Rule @@@ originBigramsFreq5Interesting]},
 Pane[Subgraph[g, WeaklyConnectedComponents[g][[1]], VertexLabels →
    Placed["Name", Center, Panel[#, FrameMargins → 0, Background → White] &],
   GraphStyle → "SimpleLink", GraphLayout → {"SpringElectricalEmbedding",
     "RepulsiveForcePower" → -2}, EdgeShapeFunction →
    GraphElementData["ShortFilledArrow", "ArrowSize" → 0.004],
   EdgeStyle → Gray, ImageSize → {2200, 1600}], {Full, 400}, Scrollbars → True]]
```

More fine-grained vector visualization

```
Length[originWords]
```

```
MatrixPlot[SparseArray[Position[originWords, "Lyell"] /. {n_} → ({1, n} → 1),
  {1, 149 982}, 0], ImageSize → Full]
```

```
MatrixPlot[SparseArray[Position[originWords, "Murchison"] /. {n_} → ({1, n} → 1),
  {1, 149 982}, 0], ImageSize → Full]
```

Another example: first row is “varieties”, second is “variation”

```
MatrixPlot[
 SparseArray[Join[Map[Prepend[#, 1] → 1 &, Position[originWords, "varieties"]],
   Map[Prepend[#, 2] → 2 &, Position[originWords, "variation"]]], {2, 151 205}, 0],
 ImageSize → Full, FrameTicks → {None, True}]
```

TF-IDF revisited

Suppose you find a page that is particularly relevant to your research. You can use the TF-IDF measure to find other pages that are closely related to that one. We will demonstrate this for pages from *Origin*, but the exact same technique works for a large corpus of documents and is useful in the creation of tools like custom search engines.

In Chapter 5, we worked through the calculation of TF-IDF step-by-step. Here we just want to bundle all of those calculations into a single function that we can apply over and over. The function below takes a text string, a list representing document frequencies for the whole document or corpus, and an integer indicating how many documents are being compared.

```
textTFIDF[txtstr_, docfreq_, numdocs_] :=
 Module[{outlist, txt, termlist, tflist, dflist, tf, df},
  outlist = {};
  txt = ToLowerCase[StringSplit[txtstr, Except[WordCharacter] ..]];
  termlist = Union[txt];
  tflist = Sort[Tally[txt], #1[[2]] > #2[[2]] &];
  dflist = Select[docfreq, MemberQ[termlist, #[[1]]] &];
  Do[
   tf = Cases[tflist, {t, x_} → x][[1]];
   df = Cases[dflist, {t, x_} → x][[1]];
   outlist = Append[outlist, {t, tf, df, (Log[tf + 1.0] Log[numdocs / df])}],
   {t, termlist}];
  Return[Sort[outlist, #1[[4]] > #2[[4]] &]]]
```

Let’s confirm that this function works exactly the same as the code above. We can do this by calculating the TF-IDF for all of the terms in Chapter 9 of *Origin of Species* and then comparing our new results to the ones we already calculated. We use the command **Equal** to confirm they are identical.

```
testCh9TFIDF = textTFIDF[ch9, docFreq, 15];
```

```
Equal[testCh9TFIDF, ch9TFIDF]
```

The shorthand for **Equal** is two equal signs (==)

```
testCh9TFIDF == ch9TFIDF
```

Armed with this function, we can now pick any page in *Origin* and find other pages that are closely related. Suppose the following page piques our interest.

```
originLines[[13]]
```

We can calculate the TF-IDF for all the terms on the page. We use the **Length** command to fill in the value for the number of documents (i.e., pages).

```
textTFIDF[originLines[[13]], docFreq, Length[originLines]]
```

Next, find related pages...

Clustering with normalized compression distance

This is not the most compelling version of this - the units are too large and the side-by-side comparison makes them look unrelated. Need to find a better example.

```
normalizedCompressionDistance[x_String, y_String, compression_: "GZIP"] :=
 Module[{cx, cy, cxy},
  cx = StringLength[ExportString[x, {compression, "Text"}]];
  cy = StringLength[ExportString[y, {compression, "Text"}]];
  cxy = StringLength[ExportString[x <> y, {compression, "Text"}]];
  Return[N[(cxy - Min[cx, cy]) / Max[cx, cy]]]]
```

```
makeNCDMatrix[t1_, t2_] :=
 Module[{matrix},
  matrix = {};
  matrix = Table[1., {Length[t1]}, {Length[t2]}];
  Do[
   Do[
    matrix[[x, y]] =
     normalizedCompressionDistance[ToString[t1[[x]]], ToString[t2[[y]]]],
    {y, x, Length[t2]}],
   {x, Range[Length[t1]]}];
  Return[matrix]]
```

```
inDictQ[w_] :=
 If[Length[DictionaryLookup[w]] > 0, True, False]
```

```
originPartition = Partition[originWords, Floor[N[Length[originWords]] / 40]];
```

```
vestigesPartition = Partition[StringSplit[vestigesTrimmed],
   Floor[N[Length[StringSplit[vestigesTrimmed]]] / 40]];
```

```
bookDistances = makeNCDMatrix[originPartition, vestigesPartition];
```

```
ArrayPlot[Rescale[1 - bookDistances], ColorFunction → "BlueGreenYellow",
 ColorRules → {0.0 → White}, FrameTicks → Automatic]
```

Example: partition tuples *Origin* 7, *Vestiges* 36 look close

```
dictionaryOrigin7 = Union[Select[originPartition[[7]], inDictQ]];
```

```
dictionaryVestiges36 = Union[Select[vestigesPartition[[36]], inDictQ]];
```

```
Complement[Intersection[dictionaryOrigin7, dictionaryVestiges36], stopwords]
```

Put up pages side by side with commonalities marked

```
Module[{shared, sharedpositions1,
  color1, colored1, sharedpositions2, color2, colored2},
 shared = Complement[Intersection[dictionaryOrigin7,
    dictionaryVestiges36], stopwords];
 sharedpositions1 = Boole[Map[MemberQ[shared, #] &, originPartition[[7]]]];
 color1 = Map[If[# == 1, Darker@Blue, Black] &, sharedpositions1];
 colored1 = MapThread[Style[##] &, {originPartition[[7]], color1}];
 sharedpositions2 = Boole[Map[MemberQ[shared, #] &, vestigesPartition[[36]]]];
 color2 = Map[If[# == 1, Darker@Blue, Black] &, sharedpositions2];
 colored2 = MapThread[Style[##] &, {vestigesPartition[[36]], color2}];
 Row[{Pane[Riffle[Map[ToString[#, StandardForm] &, colored1], " "] // StringJoin,
    {350}, Scrollbars → True],
   Pane[Riffle[Map[ToString[#, StandardForm] &, colored2], " "] // StringJoin,
    {350}, Scrollbars → True]}]]
```

Character-level file browsing

Want a more sophisticated version of this idea (fixed width font, left pad beginning and ends of text, and replace everything like newline characters with wingdings - something more like the classic hexadecimal file editors).

```
viewCharacterPosition[textstr_, part_] :=
 Manipulate[
  Module[{txt},
   txt = StringTake[textstr, part];
   (StringTake[txt, {Max[1, char - 100], char - 1}] <> "▲" <>
     StringTake[txt, {char, Min[StringLength[txt], char + 200]}])],
  {char, 1, part, 1, Appearance → "Labeled"}, ContentSize → {600, 200}]

(*viewCharacterPosition[vestigesFull,2000]*)
```

Scraping Internet Archive identifiers

When we retrieved the Internet Archive identifiers for Huxley's works in Project Gutenberg, we used the advance search functionality. Another way to accomplish the same goal is to use basic search and simply scrape the information we want.

```
huxleyGutenbergIAIdentifiers2 =
  URLFetch["https://archive.org/search.php?query=" <> URLEncode[
     "creator:\"Huxley, Thomas Henry, 1825-1895\" AND collection:Gutenberg"]];

Select[StringCases[huxleyGutenbergIAIdentifiers2,
  Shortest["data-id=\"" ~~ x__ ~~ "\""] → x], StringContainsQ[#, "gut"] &]
```

Compare word lengths in two texts

We can see that although *Origin* is a longer work than *Vestiges* (there are more words in each length category) the overall distributions of word lengths are quite similar.

```
PairedHistogram[StringLength /@ originWords,
 StringLength /@ TextWords[vestigesTrimmed],
 ChartStyle → {"Pastel", None}, ChartLegends → {"Origin", "Vestiges"}]
```

OCLC WorldCat Identities

```
oclcWorldCatIdentitiesBaseURL = "http://worldcat.org/identities/";

darwinLCCN = "lccn-n78095637";
```

```
Import[oclcWorldCatIdentitiesBaseURL <> darwinLCCN, "Elements"]
```

This function gets list of associated names and IDs. Note that here we are scraping HTML but we could also use XML

```
getWorldCatIdentitiesANamesIDs[id_] :=
 StringCases[Import[oclcWorldCatIdentitiesBaseURL <> id, "Source"],
  Shortest["<a property=\"knows\"" ~~ __ ~~
     "href=\"http://www.worldcat.org/identities/" ~~ aid__ ~~
     "\" title" ~~ __ ~~ ">" ~~ a__ ~~ " </a>"] → {aid, a}]
```

```
getWorldCatIdentitiesANamesIDs[darwinLCCN]
```

Huxley

```
getWorldCatIdentitiesANamesIDs["viaf-66511085"]
```

```
getWorldCatIdentitiesNeighborhood[id_] :=
 Module[{neighbor1, outgraph, nname, ntemp},
  neighbor1 = getWorldCatIdentitiesANamesIDs[id];
  outgraph = Map["darwin" → #[[2]] &, neighbor1];
  Do[
   nname = n[[2]];
   ntemp = getWorldCatIdentitiesANamesIDs[n[[1]]];
   outgraph = Join[outgraph, Map[nname → #[[2]] &, ntemp]],
   {n, neighbor1}];
  Return[outgraph]]
```

```
darwinIDNeighborhood = getWorldCatIdentitiesNeighborhood[darwinLCCN];
```

```
Short[darwinIDNeighborhood]
```

```
Pane[Graph[darwinIDNeighborhood, VertexLabels → Placed["Name", Center,
    Panel[#, FrameMargins → 0, Background → White] &], GraphStyle → "SimpleLink",
  GraphLayout → {"SpringElectricalEmbedding", "RepulsiveForcePower" → -2},
  EdgeShapeFunction → GraphElementData["ShortFilledArrow", "ArrowSize" → 0.004],
  EdgeStyle → Gray, ImageSize → {2200, 1600}], {Full, 400}, Scrollbars → True]
```

Finding email addresses

```
buildQuery[namestr_] :=
 URLBuild[{"http://www.google.com", "search"},
  {"q" → StringRiffle[TextWords[namestr]] <> " email"}]
```

Fetch search results

```
getResults[namestr_] :=
 URLFetch[buildQuery[namestr]]
```

Given an HTML page of Google search results, return list of most common things that look like emails

```
getEmail[resultstr_] :=
 Flatten@Commonest[(StringTrim[#, Repeated["."]] &) @
      (StringCases[#, Except[WhitespaceCharacter] .. ~~
           "@" ~~ Except[WhitespaceCharacter] ..] &) @
       (StringReplace[#, Except[WordCharacter | "@" | "."] → " "] &) @
        (StringDelete[#, Shortest["&" ~~ __ ~~ ";"]] &) @
         StringDelete[#, Shortest["<" ~~ __ ~~ ">"]] & /@
    StringCases[resultstr, Repeated[_, {30}] ~~ "@" ~~ Repeated[_, {30}]]]
```

```
getEmail[getResults["William Turkel University of Western Ontario"]]
```

```
getEmail[getResults["William Turkel UWO History"]]
```

Get emails for all the people in a list, waiting between 3 and 17 seconds between queries

```
getEmails[nlist_] :=
 Module[{rlist},
  rlist = {};
  Do[
   PrintTemporary[n];
   Pause[RandomInteger[{3, 17}]];
   rlist = Append[rlist, Join[n, getEmail[getResults[n]]]],
   {n, nlist}];
  Return[rlist]]
```

Use a different search engine

```
getEmail[URLFetch@URLBuild[{"http://www.bing.com", "search"},
    {"q" → StringRiffle[TextWords["William Turkel UWO History"]] <> " email"}]]
```

Could try revising this to make use of new **TextCases** feature in *Mathematica* version 10.2

Scraping Darwin's Itinerary for the *Beagle* voyage

A more extensive example of scraping.

Link and data from original site...

```
Import["http://darwin-
online.org.uk/content/frameset?viewtype=text&itemID=A575&pageseq=1", "Elements"]
```

```
Import["http://darwin-
online.org.uk/content/contentblock?itemID=A575&basepage=1&hitpage=1&viewtype=text",
"Elements"]
```

```
StringCases[Import["http://darwin-
online.org.uk/content/contentblock?itemID=A575&basepage=1&hitpage=1&viewtype=text",
"Source"], "<p ><b >Day</b>" ~~ Shortest[__] ~~ "</p>"]
```

Rather than using original site, however, work from an Internet Archive crawl. That way don't have to worry about site changing.

```
https://web.archive.org/web/20150526181938/http://darwin-
online.org.uk/content/frameset?viewtype=text&itemID=A575&pageseq=1
```

```
Import[
 "https://web.archive.org/web/20150526181938/http://darwin-online.org.uk/
   content/frameset?viewtype=text&itemID=A575&pageseq=1", "Elements"]
```

We need to get a page from within the frame

```
Import[
 "https://web.archive.org/web/20150526181938/http://darwin-online.org.uk/
   content/frameset?viewtype=text&itemID=A575&pageseq=1", "Hyperlinks"]
```

Here is Darwin's Itinerary

```
Import[
 "https://web.archive.org/web/20150526181938/http://darwin-online.org.uk/
   content/contentblock?itemID=A575&basepage=1&hitpage=1&viewtype=text",
 "Elements"]
```

We use **StringCases** to scrape out the itinerary items and have a look at the first three entries.

```
beagleItineraryList = StringCases[Import[
    "https://web.archive.org/web/20150526181938/http://darwin-online.org.uk/
      content/contentblock?itemID=A575&basepage=1&hitpage=1&viewtype=text"
      , "Source"], "<p ><b >Day</b>" ~~ Shortest[__] ~~ "</p>"];
```

```
beagleItineraryList[[1 ;; 3]]
```

Now we want to pull out the date and latitude and longitude. Note that our scraper doesn't pull Day 1 because latitude and longitude aren't recorded.

```
Flatten@StringCases[StringReplace[#, "º" → "°"],
    Shortest["— " ~~ d : (DigitCharacter .. ~~ __ ~~ "183" ~~ DigitCharacter) ~~
       " —" ~~ __ ~~ p : (DigitCharacter .. ~~ "°" ~~ DigitCharacter .. ~~
          "' " ~~ Characters["NSEW"] ~~ " " ~~ DigitCharacter .. ~~
          "°" ~~ DigitCharacter .. ~~ "' " ~~ Characters["NSEW"])] →
     {DateObject[d], GeoPosition[p]}] & /@ beagleItineraryList[[1 ;; 3]]
```

We won't do anything with the dates right now, but let's get all the locations and plot them on a world map.

```
beagleItineraryLocations =
  Flatten@StringCases[StringReplace[#, "º" → "°"], Shortest[
      p : (DigitCharacter .. ~~ "°" ~~ DigitCharacter .. ~~ "' " ~~ Characters[
          "NSEW"] ~~ " " ~~ DigitCharacter .. ~~ "°" ~~
        DigitCharacter .. ~~ "' " ~~ Characters["NSEW"])] →
    GeoPosition[p]] & /@ beagleItineraryList;
```

```
GeoListPlot[Flatten@beagleItineraryLocations, PlotMarkers → None, Joined → True,
 GeoRange → "World", GeoBackground → "ReliefMap", ImageSize → Large]
```

Using the Open Library API

This example requires *Mathematica* 10.2 or later.

```
openLibrary = ServiceConnect["OpenLibrary"]
```

```
lyellResults =
 openLibrary["BookSearch", {"Author" → "Charles Lyell", "MaxItems" → 20}]
```

Second result has full text...

```
lyellResults[[2]]
```

Get Open Library ID

```
lyellResults[2, "EditionKey"]
```

Get information about that edition

```
openLibrary["BookInformation", {"BibKeys" → lyellResults[2, "EditionKey"][[1]]}]
```

Get the full text using Open Library ID

```
lyellManual = openLibrary["BookText", {"BibKeys" → {"OLID", "OL25467158M"}}];
```

Now we can analyze the text like any other.

```
TextWords[StringTake[lyellManual[1], 2500]]
```

Find books about Charles Lyell

```
openLibrary["BookSearch", {"Subject" → "charles lyell"}]
```

Learn more about *The Ice Finders*

```
openLibrary["BookSearch", {"Author" → "bolles", "Title" → "ice finders"}][1][
 "EditionKey"]
```

```
openLibrary["BookInformation",
  {"BibKeys" → {"OLID", "OL23243795M"}}]["Subjects"]
```

TextCases

The **TextCases** command is experimental as of *Mathematica* 10.2

```
origin = ExampleData[{"Text", "OriginOfSpecies"}];
origin20K = StringTake[origin, 20 000];
```

```
viewData[origin20K]
```

```
TextCases[origin20K, "Country"]
```

```
TextCases[origin20K, "ProperNoun", PerformanceGoal → "Quality"]
```

```
TextCases[origin20K, "Color"]
```

```
Sort[Tally[TextCases[origin20K, "Adjective"]], #1[[2]] > #2[[2]] &][[1 ;; 20]] //
 TableForm
```

Working with JSTOR data for research

Citations for articles with Darwin in the title

```
jstorCitations = Import[
    "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/
      master/jstor-citations/citations.xml"];
```

```
Short[jstorCitations, 20]
```

Following code adapted from method in http://mathematica.stackexchange.com/questions/58527/import-itunes-xml-data-and-convert-it-into-a-dataset-or-table

```
Clear[jstorXMLToDataset];
jstorXMLToDataset[xml_] :=
 Block[{XMLElement},
  XMLElement["citations", _, c_] := Dataset @ <|c|>;
  XMLElement["article", {id_}, c_] := Unique["id"] → <|c |>;
  XMLElement["doi", _, {c_}] := "doi" → c ;
  XMLElement["title", _, {c_}] := "title" → c ;
  XMLElement["author", _, {c_}] := "author" → c ;
  XMLElement["journaltitle", _, {c_}] := "journaltitle" → c ;
  XMLElement["volume", _, {c_}] := "volume" → c ;
  XMLElement["issue", _, {c_}] := "issue" → c ;
  XMLElement["pubdate", _, {c_}] := "pubdate" → DateList[c][[1 ;; 3]] ;
  XMLElement["pagerange", _, {c_}] := "pagerange" → c ;
  XMLElement["publisher", _, {c_}] := "publisher" → c ;
  XMLElement["type", _, {c_}] := "type" → c ;
  XMLElement["reviewed-work", _, {c_}] := "reviewed-work" → c ;
  XMLElement["abstract", _, {c_}] := "abstract" → c ;
  XMLElement[t_, _, {}] := t → Null;
  xml[[2]]
 ]
```

Dataset of all records

```
Clear[cites]; cites = jstorXMLToDataset[jstorCitations]
```

First record

```
cites[1]
```

```
Head[First[cites]]
```

```
Head[cites]
```

Number of records

```
Length[cites]
```

Retrieve DOI for a couple of records

```
cites[1, "doi"]
```

```
cites[20, "doi"]
```

Find DOIs that match a pattern

```
Select[cites[All, "doi"], StringMatchQ[#, __ ~~ ";" ~~ __] &]
```

Now want to grab all of the keywords and put into another dataset. I have only included the first five of the raw keyword files in Github to show how the technique works.

Given a DOI return keyterm file name

```
doiToKeytermFilename[doi_] :=
 "keyterms_" <> StringReplace[doi, {"/" → "_", ":" → "_"}, 1] <> ".XML"
```

```
For[i = 1, i ≤ 5, i++,
 Print[doiToKeytermFilename[cites[i, "doi"]]]]
```

Paths to keyterm files on Github

```
"https://github.com/williamjturkel/Digital-Research-Methods/raw/master/jstor-
  citations/keyterms/keyterms_10.1525_rep.2004.88.1.55.XML"
```

```
"https://github.com/williamjturkel/Digital-Research-Methods/raw/master/jstor-
  citations/keyterms/keyterms_10.2307_1639681.XML"
```

Given a dataset number get terms from a keyterm file

```
doiKeyterms[n_] :=
 Block[{XMLElement, xml},
  xml = Import[
    "https://github.com/williamjturkel/Digital-Research-Methods/raw/master/
       jstor-citations/keyterms/" <>
     doiToKeytermFilename[cites[n, "doi"]]];
  XMLElement["article", {id_}, c_] :=
   Unique["id"] → <|"doi" → id[[2]], "keyterms" → c|> ;
  XMLElement["keyterm", {w_}, {c_}] := c → w[[2]];
  xml[[2]]]
```

```
doiKeyterms[1]
```

Keyterm dataset

```
jstorXMLToKeytermDataset[dataset_] :=
 Dataset[<|doiKeyterms /@ Range[Length[dataset]]|>]
```

Create a dataset of keyterms for the first five records

```
keytermsFive = jstorXMLToKeytermDataset[cites[[1 ;; 5]]];
```

```
keytermsFive
```

Now can join information from both tables using DOI

```
cites[3, "doi"]
```

Third record

```
keytermsFive[Select[#doi == cites[3, "doi"] &]]
```

Keyterms of third record

```
keytermsFive[Select[#doi == cites[3, "doi"] &], "keyterms"]
```

First keyterm of third record

```
keytermsFive[Select[#doi == cites[3, "doi"] &], "keyterms", 1]
```

All keyterms of third record, in list form

```
Normal[Values[keytermsFive[Select[#doi == cites[3, "doi"] &], "keyterms"]][[1]]]
```

All keyterms of third record, list of keyterms only (without weighting)

```
Normal[
 Keys[Values[keytermsFive[Select[#doi == cites[3, "doi"] &], "keyterms"]][[1]]]]
```

Rather than put all of the individual keyterm files on Github, I created a *Mathematica* dataset that can be loaded directly. This has keyterms for all records.

```
keyterms = Get[
  "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/
    master/jstor-citations/keyterms-dataset.m"]
```

Try plotting a particular keyword over time

```
Normal[cites[3, "pubdate"]]
```

```
Histogram[Normal[Values[cites[All, "pubdate", 1]]]]
```

Create an association by joining information from both datasets using DOI as primary key

```
doiAssoc1 = JoinAcross[Normal@Values@cites[All, {"doi", "pubdate"}],
  Normal@Values@keyterms[All, {"doi", "keyterms"}], Key["doi"]]
```

```
Head[doiAssoc1]
```

```
Head[First[doiAssoc1]]
```

Keywords for records published in 1917

```
Cases[Normal@doiAssoc1, {_, "pubdate" → {1917, _, _}, _}]
```

```
Keys[First[doiAssoc1]["keyterms"]]
```

```
MemberQ[Keys[First[doiAssoc1]["keyterms"]], "gender"]
```

```
(MemberQ[Keys[doiAssoc1[[#]]["keyterms"]], "gender"] &)[1]
```

Can strip down the association

```
doiAssoc2 = {doiAssoc1[[#]]["pubdate"][[1]], Keys@doiAssoc1[[#]]["keyterms"]} & /@
    Range[Length[doiAssoc1]];
```

```
Short[doiAssoc2, 10]
```

Compute total number of keyterms per year - note increased numbers in 1909, 1959, 2009

```
keytermsPerYear =
 GroupBy[Sort[{#[[1]], Length[#[[2]]]} & /@ doiAssoc2], First → Last, Total]
```

```
ListLogPlot[Reverse[keytermsPerYear], Filling → Axis, PlotRange → Full]
```

Make a structure that has keyterm over time as proportion of articles in which it appears

```
doiAssoc3 = GroupBy[Flatten[Tuples[{doiAssoc2[[#, 2]], {doiAssoc2[[#, 1]]}}] & /@
      Range[Length[doiAssoc2]], 1], First → Last, Tally];

Short[doiAssoc3, 10]

doiAssoc3["gender"]

List[List[#[[1]]], #[[2]]] & /@ doiAssoc3["huxley"]
```

Get frequencies for all keyterms

```
keytermFreq = SortBy[Tally[Flatten[
      Keys[doiAssoc1[[#]]["keyterms"]] & /@ Range[Length[doiAssoc1]]]], Last];

Take[keytermFreq, -100]
```

Sparklines http://mathematica.stackexchange.com/questions/9095/adding-lines-to-sparklines-plots-w-o-frames-axes-etc

Further Digital Research Methods

These are things that I am considering adding at some point...

- **3D Models**. Working with heritage artifact scans
- **APIs**. Using Wolfram Cloud to create application programming interfaces to share research results
- **Approximate pattern matching**. EditDistance, HammingDistance, DamerauLevenshteinDistance, Jaccard (dis)similarity, etc.
- **Audio / sound**.
- **Bibliography and bibliometrics**. Citation databases and citation analysis. Deciphering journal abbreviations.
  http://journal.code4lib.org/articles/1758
- **Biography**. Historical biographies of natural historians (OCR is unusable in Internet Archive book). Scrape names from Darwin Online and/or Wallace Online sites?
  https://archive.org/details/Taxidermywithbi00Swai
  http://www.hps.cam.ac.uk/research/nhbnc.html
- **Captain Cook**.
  Select[histevents, StringContainsQ[#[[1]], "Cook"] && 1760 <= DateList[#[[2]]][[1]] <= 1780 &]
- **Cloud deployment**. Sharing research results in manipulable form
- **Comics**. Here is a nice source about 'evolution'
  http://digitalcomicmuseum.com/preview/index.php?did=17840&page=3
- **Computable data**. 'Freezing' things in a form that can be quickly reloaded and rerun.
- **Computational journalism**. Implementation of techniques from Jonathan Stray course or ProPublica website.
  http://courses.jmsc.hku.hk/jmsc6041spring2013/2013/02/14/introduction-computer-science-and-journalism/
- **Correspondence networks**. Text of 7500 Darwin letters and information about 7500 more.
  http://www.darwinproject.ac.uk
  http://www.darwinproject.ac.uk/all-darwins-correspondents
- **CrossRef metadata search**. Resolve free-form citations
  http://labs.crossref.org/resolving-citations-we-dont-need-no-stinkin-parser/
- **Date Histogram**.
- **DBPedia**.
  http://wiki.dbpedia.org

Digital Humanities. Points of contact with other projects
http://programminghistorian.org
http://docs.voyant-tools.org/about/examples-gallery/
http://nbviewer.ipython.org/github/sgsinclair/alta/blob/master/ipynb/ArtOfLiteraryTextAnalysis.ipynb
http://www.themacroscope.org
http://benschmidt.org/dighist13/syllabus.pdf
http://dh-r.lincolnmullen.com
http://douglasduhaime.com
http://mariandoerk.de/wordwanderer/corpora2015.pdf

- **DOI resolving**.
- **DPLA**. API
  http://dp.la/info/developers/codex/
- **Dropbox access**. Accessible from *Mathematica*
- **Entropy browser**. Use his method to explore WARC files? Identify pictures, stretches of text / code, etc.?
  http://yurichev.com/blog/entropy/
- **External programs and file system**. Shell access. FindList.
- **Facial recognition**. Eigenfaces example - details in Evernote notebook
- **Genealogy**. Reconstruct Darwin, Wedgwood, Galton family tree. Possibly by spidering Wikipedia Infobox person template, or possibly with DBPedia data. WolframAlpha has nice graphics for kinship relations; possible to use those generatively? Could also scrape "CD's cousin", "CD's son" labels from Darwin Project correspondence page. GEDCOM file format. GedML
  https://en.wikipedia.org/wiki/Template:Infobox_person
  https://en.wikipedia.org/wiki/Charles_Darwin
  https://en.wikipedia.org/wiki/Darwin–Wedgwood_family
  <table class="infobox vcard" style="width:22em">
  <th scope="row">Born</th>
  http://www.darwinproject.ac.uk/all-darwins-correspondents
  http://library.wolfram.com/infocenter/Demos/4215/
  http://homepages.rootsweb.ancestry.com/~pmcbride/gedcom/55gctoc.htm
  http://www.gedcomx.org/About.html
- **Geographical networks**. Late Medieval Trade Routes. ARPANET over time.
  http://upload.wikimedia.org/wikipedia/commons/e/e1/Late_Medieval_Trade _Routes.jpg
  http://som.csudh.edu/cis/lpress/history/arpamaps/
- **Geolocation**. I have an example of geolocating places of newspaper publication using Trove data
- **Google n-gram data**.
- **Grammars**. Use rewriting rules to define grammar for limited domain (e.g., historical currencies, extracting capitalized phrases of arbitrary length). Simple grammars and transformation rules. Getting WordData for sentence understanding. Little languages or EDSLs.
  http://reference.wolfram.com/language/guide/ProgrammableLinguisticInterface.html
- **Handwriting**.
- **Hathi Trust**. Data for mining
  http://www.hathitrust.org/feature_extraction_alpha _release
  https://sharc.hathitrust.org/features
- **Historical photos**. Train machine learner to distinguish different eras of photography: daguerreotypes, etc. Machine learning of gender with historical fashions. Draw images from British Library million or Flickr Commons? Possible to automate search for image details hidden in shadows of historical photos?
  http://britishlibrary.typepad.co.uk/digital-scholarship/2013/12/a-million-first-steps.html
  https://github.com/BL-Labs/imagedirectory

- **Image averaging**.
- **Image repair**. Inpainting examples from *Mathematica* documentation (repair old photos, remove time stamps and watermarks). Adapt ImageMagick technique for automatically increasing legibility of scans
  http://journal.code4lib.org/articles/5385
  http://darwin-online.org.uk/graphics/RN_Illustrations.html
  http://darwin-online.org.uk/content/frameset?pageseq=1&itemID=CUL-DAR209.14.172&viewtype=image
  http://darwin.amnh.org/files/images/large/79949_MS-DAR-00049-000-00015.jpg
  http://www.amnh.org/our-research/darwin-manuscripts-project/journal-pocket-diary/1838-1881
- **Image tampering**. e.g., Hany Farid's work. Comparing images with retouched or hoax versions?
- **ISBN services**.
  http://journal.code4lib.org/articles/8715
- **Latent Dirichlet allocation and latent semantic indexing**. Example of LSI using DimensionReduce, DimensionReduction? (It is Experimental as of *Mathematica* v10.1)
- **Library APIs**. e.g., OCLC developer, LibraryThing. Getting more information about the books mentioned in *Origin*: *Vestiges*, *Principles of Geology*, Hooker's work, etc. Alternately, do something with the books that were carried on the *Beagle*. Closest library with item. WorldCat search API can do SRU but requires developer wskey. European Library has about 35K items for Darwin.
  http://www.darwinproject.ac.uk/books-on-the-beagle
  http://darwin-online.org.uk/BeagleLibrary/Beagle_Library_Introduction.htm
  Find your library's code at http://www.worldcat.org/libraries (Western libraries 66428, oclcsymbol UWO)
  https://platform.worldcat.org/api-explorer/wcapi
  https://www.oclc.org/developer/develop/web-services/worldcat-search-api/bibliographic-resource.en.html
  http://www.theeuropeanlibrary.org/tel4/search?query=darwin
- **Linked open data for cultural heritage**. Some mixture of DBPedia, LC/NAF and VIAF.
- **Machine learning**. My Weiss et al *Text Mining* notebook has an example of Naive Bayesian that could be easily adapted.
  https://mathematicaforprediction.wordpress.com
  https://github.com/antononcube/MathematicaForPrediction
- **Manuscripts**. Catalogue of all Darwin mss in the world (76K records)
  http://test.darwin-online.org.uk/MScatintro.html
- **Moving images**. Visualizing film, TV, video. Doing OCR on subtitles and news tickers
- **Network analysis**. Adapt Paul Revere example?
- **New *Mathematica* commands**. Most later than version 10.1
  TextCases, TextPosition, Containing
  TextSearch, TextSearchReport, SearchIndices, CreateSearchIndex, DeleteSearchIndex, ContentObject, Snippet
  CharacterName
  GoogleCustomSearch
  NearestNeighborGraph
  NestGraph (use to find nearby meanings for semantic data? or in spidering task?)
  Nothing
  NumberDecompose, MixedRadix (currency, time)
  ConformImages, Thumbnail
  WolframLanguageData
- **Newspapers**. In addition to Library of Congress' *Chronicling America* and Trove, there are 11 million pages of European newspapers at
  http://www.theeuropeanlibrary.org/tel4/newspapers

- **Notebooks**. Using code inside the notebook to solve various problems that may come up in research (easy things that come to mind would be automatically creating section headings for list of topics or dates; programmatic uses of Manipulate; hard things include searching and indexing; manipulating outlines; citation management)
- **OAI-PMH metadata**.
  http://www.openarchives.org/Register/BrowseSites
  http://journal.code4lib.org/articles/7818
- **OCLC million**. One million most widely held works in WorldCat
  http://www.oclc.org/news/releases/2012/201252.en.html
- **OCR**. Assess OCR quality by looking at ratio of 'words' to dictionary words. Build a Manipulate to explore parameters that make recognition better. StackExchange example of pulling material from different parts of the page.
- **Open Calais**. If this is still easy to call, I have an example using Trove
- **Overview project**. Details of the use of k-means clustering and TF-IDF in creating folders
  https://blog.overviewdocs.com/2013/04/30/how-overview-can-organize-thousands-of-documents-for-a-reporter/
- **PageRank**. Weiss et al *Text Mining* 103-104; *Mathematica* help file for PageRankCentrality.
  http://arxiv.org/abs/cs/0601030
- **Panoramas**. Use ImageAlign to do an example with historical and contemporary photos (rephotography). Darwin's Down House examples? Can Inpaint borders after transformations.
  http://www.english-heritage.org.uk/visit/places/home-of-charles-darwin-down-house/
  https://www.darwinproject.ac.uk/darwin-and-down
  http://www.lib.cam.ac.uk/Newsletters/nl07/downhouse.html
  https://www.darwinproject.ac.uk/dining-at-down-house
  https://en.wikipedia.org/wiki/Down_House#/media/File:Salón_Down _House.jpg
  http://www.english-heritage.org.uk/content/properties/the-home-of-charles-darwin-down-house/portico/old-study-1882
- **PDF**. Extracting text and page images. Bursting a PDF. (*Mathematica* seems to have a lot of difficulty with PDFs, so may need to call on *ghostscript*)
  http://mathematica.stackexchange.com/questions/2781/difficulties-with-importing-pdfs-in-mathematica
- **RDF**.
- **Real time monitoring**. Information trapping. Wikipedia edits. Google Trends. ScheduledTasks run with CloudDeploy (guide/TimedEvaluations in documentation).
  http://www.google.com/trends/explore#q=geocities&geo=PE
- **Sequence alignments**. Analyze successive edits of *Origin*. Look at use of Overscript and Overlay in keyboard example. Compare with Ben Fry visualization...
  http://www.wolfram.com/language/gallery/correct-and-grade-keyboard-practice/
  http://benfry.com/traces/
- **Shape analysis**. Barnacles and orchids. Might also do something with fractal dimension. Possible to use TimeWarpingCorrespondence to compare shapes?
  http://darwin-online.org.uk/graphics/Living_Cirripedia_illustrations.html
  http://darwin-online.org.uk/graphics/FertilisationofOrchids_Illustrations.html
  http://darwin-online.org.uk/graphics/illustrations.html
- **Sparklines**.
  http://mathematica.stackexchange.com/questions/9095/adding-lines-to-sparklines-plots-w-o-frames-axes-etc
- **SPARQL**.
- **Stereoscopic images**. Reconstructing 3D geometry

- **Stream methods**. For working with very large files
  http://reference.wolfram.com/language/tutorial/StreamMethods.html
- **Structured data and SQL**. Querying databases. Descriptive statistics. Dataset data structure. Selections and transformations. Relational data and normalization. Grabbing stuff from SQL databases. New dataset manipulation commands allow you to do SQL-like things with structured data (JoinAcross etc.)
  http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/
- **SRU**. Search/Retrieval via URL.
  http://www.loc.gov/standards/sru/misc/simple.html
  http://www.loc.gov/standards/sru/resources/lcServers.html
  http://stackoverflow.com/questions/13667361/how-to-retrieve-books-information-in-xml-json-from-library-of-congress-by-isbn
- **Temporal expression grammar**. Chang & Manning 2012. Stanford SUTime tagger.
  https://github.com/stanfordnlp/CoreNLP/blob/master/src/edu/stanford/nlp/time/rules/english.sutime.txt
- **Text classifier**. I have one for images but not for text. Maybe something similar to the prose or poetry or authorship examples. Also check Classify documentation for built-in classifiers that may be of interest (Spam, Sentiment, NameGender, FacebookTopic, etc.)
  http://www.wolfram.com/language/gallery/determine-if-a-text-is-prose-or-poetry/
  http://www.wolfram.com/language/gallery/determine-the-author-of-a-text/
- **TF-IDF**. Other ways of calculating TF-IDF. Using IDF or TF-IDF to weight scores in document vector model. My Weiss et al *Text Mining* notebook has an example in section 4.4.2. Cosine similarity example in section 4.4.3.
  http://en.wikipedia.org/wiki/Tf–idf
- **Topic modeling**. Implement from scratch.
- **Trigram frequencies**. Function in this example
  http://www.wolfram.com/language/gallery/generate-random-pronounceable-words/
- **Tweets**. Sentiment example? Seriating matrix of mentions
  https://www.miskatonic.org/2013/02/24/seriation-and-kayiwa-yobj-vortex/
- **Unicode**.
- **Unsupervised clustering**. PeakDetect, FindPeaks. Classify. DimensionReduce.
  tutorial/PartitioningDataIntoClusters
- **VIAF**. OCLC Virtual International Authority File
  http://inkdroid.org/journal/2012/05/15/diving-into-viaf/
- **Visualization**. SectorChart for something? Try adapting some examples from D3. Using ArrayPlot to plot cooccurrence relations.
  https://github.com/mbostock/d3/wiki/Gallery
  http://prcweb.co.uk/circularheatchart/
  http://bl.ocks.org/mbostock/4062006
  http://bl.ocks.org/mbostock/4063269
  http://orbitingfrog.com/2012/07/27/more-astronomy-data-mining-its-word-matrix-time/
  http://deliveryimages.acm.org/10.1145/1750000/1743567/figs/f5c.jpg
  http://cacm.acm.org/magazines/2010/6/92482-a-tour-through-the-visualization-zoo/fulltext#figures
- **WARC file analysis**. Use secure hash of images in WARC files to see how images travel from one website to another.
  https://archive.org/details/ExampleArcAndWarcFiles
  https://bitbucket.org/hanzo/warc-tools
- **Web Crawling**. Example of building up a complicated program step-by-step.

- **Wikipedia**. WikipediaData and WikipediaSearch. Possible to get links into and out of articles for spidering or citation analysis. Also possible to do stuff with Geolocation / GeoDisk and with image thumbnails. And Wikipedia entries in other languages (Title Translation Rules). Categories and subcategories. A very interesting example that uses cross-correlation between time series for different celebrities to cluster them into groups (cf computational history examples): http://www.wolfram.com/language/gallery/visualize-celebrity-gossip/
- **Wolfram Data Drop**. http://datadrop.wolframcloud.com
- **Zenodo**.
- **Zipf's Law**. Idea that it is a consequence of having whitespace. Discussion in Manning & Schutze, *Foundations*. http://mathworld.wolfram.com/ZipfsLaw.html
- **Zotero API**. Have *Mathematica* code for earlier version of API but need to test with v.3

appA

Appendix A: Sources and Code

This contains sources and code from chapters 01-03 in a form that can be easily evaluated if you want to restart *Mathematica* or copy them into another notebook.

Sources

Darwin's *Origin of Species*

```
origin = ExampleData[{"Text", "OriginOfSpecies"}];
```

Stopwords

```
stopwords = WordData[All, "Stopwords"];
```

All words in order

```
originWords = TextWords[origin];
```

Unique terms

```
originTerms = Union[TextWords[origin]];
```

Word frequencies

```
originWordFreqs = WordCounts[origin, IgnoreCase → True];
```

Bigram frequencies

```
originBigrams = WordCounts[origin, 2];
```

Code

Predicates

```
capitalizedQ[w_] := UpperCaseQ[StringTake[w, 1]]
```

```
nonStopwordQ[w_] :=
 Not[MemberQ[stopwords, w]]
```

These all search WordData

```
nounQ[str_] :=
 Cases[WordData[str], {_, "Noun", ___}] ≠ {}
```

```
adjectiveQ[str_] :=
 Cases[WordData[str], {_, "Adjective", ___}] ≠ {}
```

```
personTermQ[str_] :=
 If[nounQ[str] && Not[adjectiveQ[str]], MemberQ[
   Union[Flatten[NestList[Flatten[WordData[#, "BroaderTerms", "List"] & /@ #] &,
       {str}, 3]]], "person"], False]
```

Get *Wolfram Alpha* Person Entity

```
getWAPerson[str_] :=
 Module[{entity},
  entity = WolframAlpha[str, "WolframResult"];
  If[EntityTypeName[entity] == "Person", Return[entity]]]
```

Bag of Words

```
bagOfWords[str_] :=
 Union@DeleteStopwords@TextWords@ToLowerCase[str]
```

View Data

```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Keyword in Context

Best to set window size to 3 words on either side

```
kwic[text_String, keyword_String, win_Integer] :=
 Module[{wordpattern, window, resultlist, formatted},
  wordpattern = WordCharacter .. ~~ Except[WordCharacter] ..;
  window = Repeated[wordpattern, {win}];
  resultlist = StringCases[text,
    window ~~ keyword ~~ Except[WordCharacter] .. ~~ window, IgnoreCase → True];
  formatted = Style[TableForm[StringSplit[resultlist]], Medium];
  Return[Framed[Pane[formatted, {Full, Automatic}, Scrollbars → True]]]]
```

Text Search

```
textSearch[txt_, str_] :=
 TabView[Map[StringTake[txt, {#[[1]] - 100, #[[2]] + 100}] &,
   StringPosition[txt, str ~~ WordBoundary]]]
```

Finding One String Near Another

```
stringFindNear[txt_, str1_, str2_, within_] :=
 TabView[Map[StringTake[txt, {#[[1]] - 100, #[[2]] + 100}] &,
   Select[DeleteCases[Partition[Sort[Join[Map[List[str1, #] &, StringPosition[
             txt, str1 ~~ WordBoundary][[All, 1]]], Map[List[str2, #] &,
          StringPosition[txt, str2 ~~ WordBoundary][[All, 1]]]], #1[[2]] < #2[[2]] &],
       2, 1], {{x_, _}, {x_, _}}][[All, 1 ;; 2, 2]], (#[[2]] - #[[1]]) ≤ within &]]]
```

Who's Who

This version shows text matches

```
formatWAPersonData[assoc_] :=
 Grid[{{assoc[[Key[EntityProperty["Person", "Image"]]]],
    Column[{Text@Style[assoc[[Key[EntityProperty["Person", "FullName"]]]], Bold],
      Text["b. " <> DateString@
          assoc[[Key[EntityProperty["Person", "BirthDate"]]]] <> ", d. " <>
        DateString@assoc[[Key[EntityProperty["Person", "DeathDate"]]]]],
      Column[Text[Style[#, Medium]] & /@
        assoc[[Key[EntityProperty["Person", "NotableFacts"]]]]]}]},
  Frame → All, ItemSize → {{Scaled[.25], Scaled[.65]}},
  Alignment → {{Center, Left}, {Top, Top}}]

whosWho2[assoc_, txt_] :=
 MenuView[Sort[Table[Keys[assoc][[i]] → Column[{
       formatWAPersonData[assoc[[Keys[assoc][[i]]]]],
       Text@
        SlideView[With[{srch = Map[StringTake[txt, {#[[1]] - 200, #[[2]] + 200}] &,
             StringPosition[txt, StringSplit[Keys[assoc][[i]],
                 Except[WordCharacter]][[1]] ~~ WordBoundary]]},
          If[srch ≠ {}, srch, {Style["No match found", Italic]}]]],
        ImageSize → Scaled[0.9], ControlPlacement → Bottom,
        AppearanceElements → {"FirstSlide", "PreviousSlide",
          "NextSlide", "LastSlide", "SlideNumber", "SlideTotal"}]}],
    {i, Length[Keys[assoc]]}]], ImageSize → Automatic]
```

Capitalized Words and Bigrams

Both take a text string

```
capitalizedWords[textstr_] :=
 Union[Select[
   Flatten[Map[Rest, Map[TextWords, TextSentences[textstr]]]], capitalizedQ]]

capitalizedBigrams[txtstr_] :=
 Cases[Keys[WordCounts[
    StringRiffle[Flatten[Map[Rest, Map[TextWords, TextSentences[txtstr]]]]],
    2]], {_?capitalizedQ, _?capitalizedQ}]
```