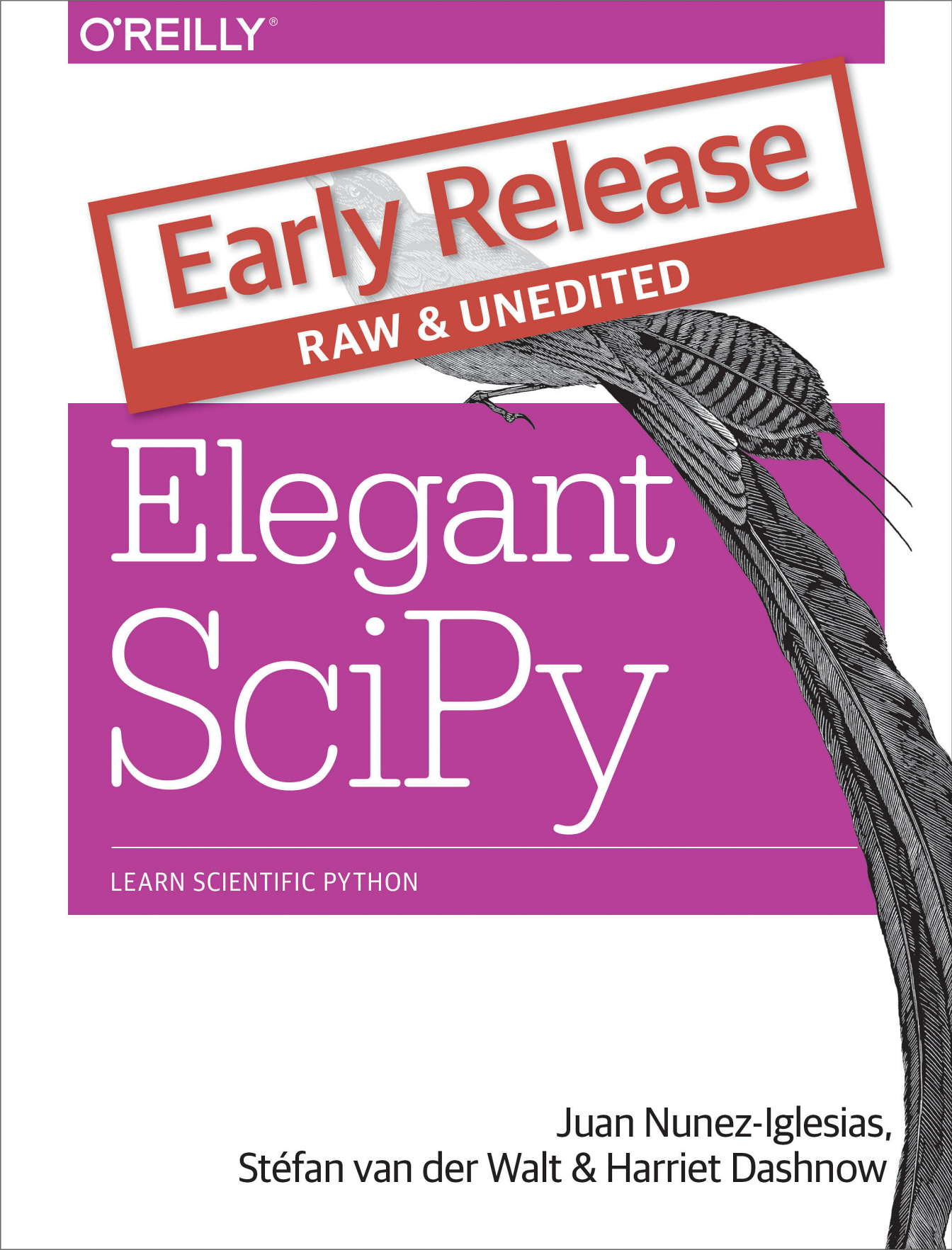
O'REILLY®

Early Release

RAW & UNEDITED



Elegant SciPy

LEARN SCIENTIFIC PYTHON

Juan Nunez-Iglesias,
Stéfan van der Walt & Harriet Dashnow

Elegant SciPy

Juan Nunez-Iglesias, Stéfan van der Walt, Harriet Dashnow

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Elegant SciPy

by Juan Nunez-Iglesias , Stéfan van der Walt , and Harriet Dashnow

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Nan Barber
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2016: First Edition

Revision History for the First Edition

2016-07-11: First Early Release
2016-07-14: Second Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491922873* for release details.

978-1-491-92287-3

[FILL IN]

Table of Contents

Preface

"Unlike the stereotypical wedding dress, it was—to use a technical term—elegant, like a computer algorithm that achieves an impressive outcome with just a few lines of code."
—-- Graeme Simsion, *The Rosie Effect*

Welcome to Elegant SciPy. We're going to spend rather a lot of time focusing on the "SciPy" bit of the title, so let's take a moment to reflect on the "Elegant" bit. There are plenty of manuals, tutorials and documentation websites out there that describe the SciPy library. Elegant SciPy goes further. More than just teaching you how to write code that works, we will inspire you to write code that rocks!

In The Rosie Effect (hilarious book; go read its prequel The Rosie Project when you're done with Elegant SciPy), Graeme Simsion twists the conventions of the word "elegant" around. Most would use it to describe the visual simplicity of something or someone stylish or graceful. Say, the first iPhone. Instead, our hero, Don Tillman, uses a computer algorithm to *define* elegance. We hope that you will know exactly what he means after reading this book. That you will read or write a piece of elegant code, and feel calmed in the glow of its beauty and grace. (Note: the authors may be prone to hyperbole.) Perhaps you too will compliment your significant other on how their outfit reminds you of that bit of code you saw in Elegant SciPy...

So, what makes code elegant? Elegant code is a pleasure to read, use and understand because it is:

- Simple
- Efficient
- Clear
- Creative

Elegant code achieves much in a few lines, through abstraction and functions, *not* through just packing in a bunch of nested function calls! Elegant code is often effi-

cient, not only in number of keystrokes but also in time and memory. Elegant code should be clear and easy to understand. It may take a minute to grok, but it should provide a crisp moment of understanding. This can be done through clear variable and function names, and carefully crafted comments, explaining not what is being done (the code does this), but *why* it is being done.

Writing this simple, efficient, clear code requires significant creativity. In the New York Times, software engineer J. Bradford Hipps recently argued that "to write better code, [one should] read Virginia Woolf." You might use a particularly efficient data structure in a new, unexpected context.

In many cases elegant code intrigues us because it does something clever, approaching a problem in a new way, or just in a way that in retrospect is obvious in its simplicity. It is the culmination of these elements of elegant code that make your code "beautiful", a pleasure to write, read and to use. This is elegant code.

Now that we've dealt with the elegant part of the title, let's bring back the SciPy.

SciPy is a library, an ecosystem, and a community. Part of what makes these great is that they have excellent online documentation and tutorials (see, e.g., https://docs.scipy.org and http://www.scipy-lectures.org/), rendering Just Another Reference book pointless; instead, Elegant SciPy wants to present the best code built using these libraries.

The code we have chosen highlights clever, elegant uses of advanced features of NumPy, SciPy, and related libraries. The beginning reader will learn to apply these libraries to real world problems using beautiful code. And we use real scientific data to motivate our examples.

When we started writing Elegant SciPy, we put out a call to the community, asking Pythonistas to nominate the most elegant code they have seen. Like SciPy itself, we wanted Elegant SciPy to be driven by the community. The code herein is the best code the community has offered up for your pleasure.

Who is this book for?

Elegant SciPy is intended to inspire you to take your Python to the next level. You will learn SciPy by example, from the very best code.

We have pitched this book towards people who have a decent beginner grounding in Python, and are now looking to do some more serious programming for their research. If you are not yet familiar with Python basics, work through a beginner tutorial before tackling this book. There are some great resources to get to get you started with Python, such as Software Carpentry (http://software-carpentry.org/).

We expect that you will be familiar with the Python programming language. You have seen Python, and know about variables, functions, loops, and maybe a bit of NumPy. Perhaps you've even honed your Python skills with some more advanced material, such as Fluent Python (http://shop.oreilly.com/product/0636920032519.do).

But you don't know whether the "SciPy stack" is a library or a menu item from International House of Pancakes, and you aren't sure about best practices. Perhaps you are a scientist who has read some Python tutorials online, and have downloaded some analysis scripts from another lab or a previous member of their own lab, and have fiddled with them. And you might think that you are more or less alone when you learn to code SciPy. You are not.

As we progress, we will teach you how to use the internet as your reference. And we will point you to the mailing lists, repositories, and conferences where you will meet like-minded scientists who are a little farther in their journey than you.

This is a book that you will read once, but may return to for inspiration (and maybe to admire some elegant code snippets!).

Why SciPy?

The NumPy and SciPy libraries make up the core of the Scientific Python ecosystem. The SciPy software library implements a set of functions for processing scientific data, such as statistics, signal processing, image processing, and function optimization. SciPy is built on top of the Python numerical array computation library NumPy. Building on NumPy and SciPy, an entire ecosystem of apps and libraries has grown dramatically over the past few years, spanning disciplines as broad as astronomy, biology, meteorology and climate science, and materials science.

This growth shows no sign of abating. For example, the Software Carpentry organization (http://software-carpentry.org/), which teaches Python to scientists, currently cannot keep up with demand, and is running "teacher training" every quarter, with a long waitlist.

In short, SciPy and related libraries will be driving much of scientific data analysis for years to come.

What is the SciPy Ecosystem?

> *"SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering."*
>
> —-- http://www.scipy.org/

The SciPy ecosystem is a loosely defined collection of Python packages. In Elegant SciPy we will see many of its main players[1]:

- **NumPy** is the foundation of scientific computing in Python. It provides efficient numeric arrays and wide support for numerical computation, including linear algebra, random numbers, and Fourier transforms. NumPy's killer feature are its ""NumPy N-dimensional arrays" on page 23", or `ndarray`. These data structures store numeric values efficiently and define a grid in any number of dimensions (more about this later). http://www.numpy.org/
- **SciPy**, the library, is a collection of efficient, user-friendly numerical algorithms. It contains toolboxes for specific domains such as signal processing, integration, optimization, and statistics. http://www.scipy.org/scipylib/index.html
- **Matplotlib** is a powerful package for plotting in two dimensions (and basic 3D). It draws its name from the syntax that it shares with Matlab. http://matplotlib.org/
- **IPython** is an interactive interface for Python, so you can quickly interact with your data and test ideas.
- **Jupyter** The Jupyter notebook runs in your browser and allows you to write code in line with text and mathematical expressions, displaying the results of computation within the text. This entire book has been written with Jupyter. Jupyter started out as an IPython extension, but now supports multiple languages, including Cython, Julia, R, Octave, Bash, Perl and Ruby. http://jupyter.org
- **pandas** provides fast, easy-to-use data structures, particularly to work with labelled data sets such as tables or relational databases, and manage time series. It also has some handy data analysis tools such as for data parsing and cleaning, sliding windows, aggregation and plotting. http://pandas.pydata.org/
- **scikit-learn** provides a unified interface to machine learning algorithms.
- **scikit-image** provides image analysis tools that integrate cleanly with the rest of SciPy.

There are others, and we will see some of them too. The takeaway here is that, although we will focus on NumPy and SciPy, it is the many surrounding packages that make Python a powerhouse for scientific computing.

Installing Python - Anaconda

Throughout this book we're going to assume that you have Python 3.5 (or a later version) and have all the major SciPy packages installed: NumPy, SciPy, Matplotlib, IPython, scikit-image, and others. We will list all of the requirements and the versions we used in the environment.yml that you can find packaged with the data for this book. The easiest way to get all of these components is to install the Anaconda Python distribution. You can download Anaconda here: http://continuum.io/downloads. You will also find detailed installation instructions.

If you find that you need to keep track of multiple versions of Python or different sets of packages, try Conda (if you've just downloaded the Anaconda Python distribution then you already have it!). That environment.yml can be passed to conda to install the right versions of everything in one go.

The Great Cataclysm: Python 2 vs. Python 3

In your Python travels, you may have already heard a few rumblings about which version of Python is better. You may wonder why it's not just the latest version.

At the end of 2008, the Python developers released a major update to the Python language, with improvements in text handling, streaming data, and overall language consistency. To quote Douglas Adams, "this has made a lot of people very angry and been widely regarded as a bad move." Python 3.0 is backwards incompatible. Python 2.6 or 2.7 code cannot be run by Python 3.x without at least minor modification. Python 2.7 was the last release of Python 2, and all development continues on the Python 3 branch, except for critical security updates. (And these will also cease after 2020.)
There will always be a tension between the march of progress and backwards compatibility. There are as many opinions on the best way to move forward as there are developers in the world. In this case, the Python core team decided that a clean break was needed to eliminate some inconsistencies in Python, and move it forward into the twenty-first century. (Python 1.0 appeared in 1994, more than 20 years ago; a lifetime in the tech world.)

Here's one way in which Python has improved in turning 3:

1. You can learn more about many of the tools mentioned here at

```
print "Hello World!"   # Python 2 print statement
print("Hello World!")  # Python 3 print function
```

So what, right? Why cause such a fuss just to add some parentheses! Well, true, but what if you want to instead print to a different *stream*, such as *standard error*, the usual place for debugging information?

```
print >>sys.stderr, "fatal error"  # Python 2
print("fatal error", file=sys.stderr)  # Python 3
```

Ah, that's looking a bit more worthwhile. What the hell is going on in that Python 2 statement? The authors don't rightly know.

Another change is the way Python 3 treats integer division, which is the way most humans treat division.

```
# Python 2
>>> 5 / 2
2
# Python 3
```

```
>>> 5 / 2
2.5
```

We were also pretty excited about the new `@` *matrix multiplication* operator introduced in Python 3.5 in 2015. Check out Chapter 5 and Chapter 6 for some examples of this operator in use!

Possibly the biggest improvement in Python 3 is its support for Unicode, a way of encoding text that allows one to use not just the English alphabet, but rather any alphabet in the world. Python 2 allowed you to define a Unicode string, like so:

```
In [1]:

beta = u"ß"
```

But in Python 3, *everything* is unicode:

```
In [2]:

ß = 0.5
print(2 * ß)

1.0
```

The Python core team decided, rightly, that it was worth supporting characters from all languages as first-class citizens in Python code. This is especially true now, when most new coders are not coming from English-speaking countries.

Nevertheless, this update broke a lot existing code, and adding insult to injury, to date, Python 3 code runs slower than the equivalent Python 2 program — even when many already complained that Python was too slow. That's certainly not motivating for someone who has to put work in to make their library work in Python 3, and it's left many continuing to use Python 2.7, refusing to upgrade.

Given the divided state of the community, many developers now write code that is compatible with both Python 2 and 3.

For new learners, the right thing to do is to use Python 3. It is the future of the language, and there is no sense in adding to the dead weight of Python 2 code littering the interwebs (including much from your authors!). In Elegant SciPy, we're going to use Python 3.5.

However, if you *must* use Python 2, you can make most of the code in this book compatible with Python 2, assuming you have the following imports at the start of your code:

```
from __future__ import division, print_function
from six.moves import zip, map, range, filter
```

A word of warning for Python 2.7 enthusiasts: Chapter 5 and Chapter 6 rely heavily on the new `@` *matrix multiplication* operator (mentioned above), which is only available in Python 3.5+. Trying to use `@` in earlier version of Python will give you a nasty

syntax error. To make the code in these chapters work with Python 2.7 you will have to use the `.dot` method of NumPy arrays and SciPy matrices, which is decidedly inelegant. More on how to do this in Chapter 5.

For more reading, you might like to check out Ed Schofield's python-future.org, and Nick Coghlan's book-length guide on the transition [^py3].

SciPy Ecosystem and Community

SciPy is a major library with a lot of functionality. Together with NumPy, it is one of Python's killer apps. And such a killer app has launched a vast number of related libraries that build on this functionality. You'll encounter many of these throughout this book.

The creators of these libraries and many of their users gather at many events and conferences around the world. These include the yearly SciPy conference in Austin, EuroSciPy, SciPy India, PyData and others. We highly recommend attending one of these, and meeting the authors of the best scientific software in the Python world. If you can't get there, or simply want a taste of these conferences, many publish their talks online.

Free and open-source software

The SciPy community embraces open source code. The source code for nearly all SciPy libraries is freely available to read, edit and reuse by anyone.

So, why open?

If you want others to use your code, one of the best ways to achieve this is to make it free. If you use closed source software, but it doesn't do exactly what you want to achieve, you're out of luck. You can email the developer and ask them to add a new feature (this often doesn't work!), or write new software yourself. If the code is open source, you can easily add or modify its functionality using the skills you learn from this book.

Similarly, if you find a bug in a piece of software, having access to the source code can make things a lot easier for both the user and the developer. Even if you don't quite understand the code, you can usually get a lot further along in diagnosing the problem, and help the developer with fixing it. It is usually a learning experience for everyone!

Open Source, Open Science

In scientific programming, all of these scenarios are extremely common and important: scientific software often builds on previous work, or modifies it in interesting

ways. And, because of the pace of scientific publishing and progress, much code is not thoroughly tested before release, resulting in minor or major bugs.

In science, another great reason for making code open source is to promote reproducible research. Many of us have had the experience of reading a really cool paper, and then downloading the code to try it out on our own data. Only we find that the executable isn't compiled for our system. Or we can't work out how to run it. Or it has bugs, missing features, or produces unexpected results. By making scientific software open source, we not only increase the quality of that software, but we make it possible to see exactly how the science was done. What assumptions were made, and even hard-coded? Open source helps to solve many of these issues. It also enables other scientists to build on the code of their peers, fostering new collaborations and speeding up scientific progress.

Open Source Licenses

If you want others to be able to use your code, then you *must* license it. If you don't license your code, it is closed by default. Even if you publish your code (for example by placing it in a public GitHub repository), without a software license, no one is allowed to use, edit or redistribute your code.

When choosing which of the many license options, you should first decide what you want people to be able to do with your code. Do you want people to be able to sell your code for profit (or sell other code that uses your code), or do you want to restrict your code to be used only in free software?

There are two broad categories of FOSS license:

- Permissive
- Copy-left

A permissive license means that you are giving anyone the write to use, edit and redistribute your code in any way that they like. This includes using your code as part of software that they are selling. Some popular choices in this category include the MIT and BSD licenses. The SciPy community has adopted the New BSD License (also called "Modified BSD" or "3-clause BSD"). Using such a license means receiving many code contributions from a wide array of people, including many in industry and start-ups.

Copy-left licenses also allow others use, edit and redistribute your code. These licenses also prescribe that derived code must also be distributed under a copy-left license. In this way, copy-left licenses restrict what users can do with the code.

The most popular copy-left license is the Gnu Public License, or GPL. The main disadvantage to using a copy-left license is that you are often putting your code off-limits to any potential users or contributors from the private sector. And this could

include your future self! This can substantially reduce your user base and thus the success of your software. In science, this could mean fewer citations.

For more help choosing a license, you might like to check out the Choose a License website http://choosealicense.com/. For licensing in a scientific context, we recommend this blog post by Jake VanderPlas, Director of Research in the Physical Sciences at the University of Washington, and all around SciPy superstar: http://www.astrobetter.com/the-whys-and-hows-of-licensing-scientific-code/. In fact we quote Jake here, to drive home the key points of software licensing.

> *...if you only take three pieces of information away from the article, let them be these:*
>
> 1. *Always license your code. Unlicensed code is closed code, so any open license is better than none (but see #2).*
> 2. *Always use a GPL-compatible license. GPL-compatible licenses ensure broad compatibility for your code, and include GPL, new BSD, MIT, and others (but see #3).*
> 3. *Always use a permissive, BSD-style license. A permissive license such as new BSD or MIT is preferable to a copyleft license such as GPL or LGPL.*
>
> —-- Jake VanderPlas http://www.astrobetter.com/the-whys-and-hows-of-licensing-scientific-code/

All the code in this book written by us is available under a BSD license. Where we have sourced code snippets from other people, the code will generally be under an open license of some variety (although not necessarily BSD).

For your own code, we recommend that you follow the practices of your community. In Scientific Python, this means 3-clause BSD, while the R language, for example, has adopted the GPL license.

GitHub: Taking Coding Social

We've talked a little about releasing your source code under an open source license. This will hopefully result in huge numbers of people downloading your code, using it, fixing bugs and adding new features. Where will you put your code so people can find it? How will those bug fixes and features get back into your code? How will you keep track of all the issues and changes? You can imagine how this could get out of control quite quickly.

Enter GitHub.

GitHub (https://github.com/) is a website for hosting, sharing and developing code. It is based on Git version control software (http://git-scm.com/). We will help you get started in GitHub land, but there are some great resources for a more in-depth experience, e.g Introducing GitHub.

GitHub has had a massive effect on open source contributions, particularly in Python. GitHub allows users to publish code. Anyone can come along and create a copy (called a *fork*) of the code and edit it to their heart's content. They can eventually contribute those changes back into the original by creating a *pull request*. There are some nice features like managing issues and change requests, as well as who can directly edit your code. You can even keep track of edits, contributors and other fun stats. There are a whole bunch of other great GitHub features, but we will leave many them for you to discover and some for you to read in later chapters. In essence, GitHub has democratized software development. It has substantially reduced the barrier to entry.

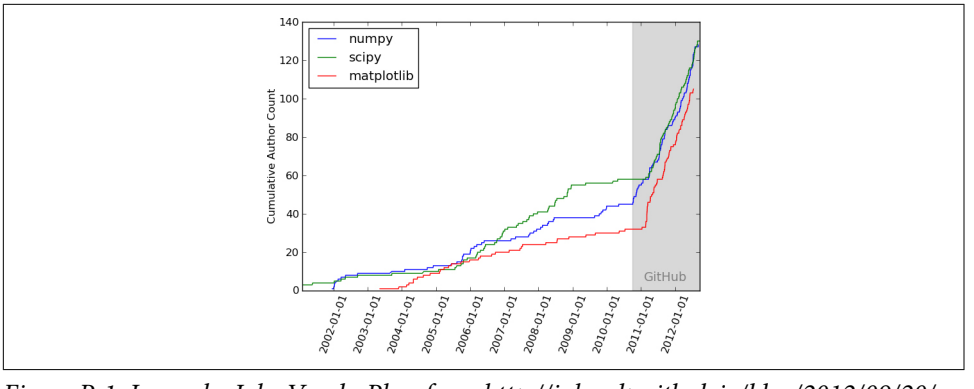


Figure P-1. Image by Jake VanderPlas, from http://jakevdp.github.io/blog/2012/09/20/why-python-is-the-last/, used with permission.

Make your Mark on the SciPy Ecosystem

Once you have become more comfortable with SciPy and started using it for your research, you may find that a particular package is lacking a feature you need, or you think that you can do something more efficiently, or perhaps you've found a bug. When you reach this point, it's time to start contributing to the SciPy Ecosystem.

We strongly encourage you to try doing this. The community lives because people are willing to share their code and improve existing code. Beyond any altruistic reasons for contributing, there are some very practical personal benefits. By engaging with the community you will become a better coder. Any code you contribute will be reviewed by others and you will receive feedback. As a side effect, you will learn how to use Git and GitHub, which are very useful tools for maintaining and sharing your own code. You may even find that interacting with the SciPy community provides you with a broader scientific network, and surprising career opportunities.

Later in Elegant SciPy we will show you how to contribute your new skills to the GitHub-hosted projects that comprise most of the scientific Python ecosystem. In the meantime we want you to start thinking about being more than just a SciPy user. You

are joining a community and we hope you will keep making it a better place for all scientific coders.

A Touch of Whimsy with your Py

In case you were worried that the SciPy community might be an imposing place to the newcomer, remember that it is made of people like you, scientists, usually with a great sense of humor.

In the land of Python, it is inevitable that you find some Monty Python references. The package Airspeed Velocity (http://spacetelescope.github.io/asv/using.html) measures your software's speed (more on this later), and references "what is the airspeed velocity of an unladen swallow?" from *Monty Python and the Holy Grail*.

Another amusingly titled package is "Sux", which allows you to use Python 2 packages from Python 3. This is a play on "six", which lets you use Python 3 syntax in Python 2, with a New Zealand accent. Sux syntax makes it less frustrating to use Python 2-only packages after you've moved to Python 3:

```
import sux
p = sux.to_use('my_py2_package')
```

In general, Python library names can be a riot, and we hope you'll enjoy your time coming up with some!

Getting Help

Our first step when we get stuck is to Google the task that we are trying to achieve, or the error message that we got. This generally leads us to Stack Overflow, an excellent question and answer site for programming. If you don't find what you're looking for immediately, try generalizing your search terms to find someone who is having a similar problem.

Sometimes, you might actually be the first person to have this specific question (this is particularly likely when you are using a brand new package), but not all is lost! As mentioned above, the SciPy community is a friendly bunch, and can be found scattered around various parts of the interwebs. Your next point of call is to Google "`<library name> mailing list`", and find an email list to ask for help. Library authors and power users read these regularly, and are very welcoming to newcomers. Note that it is common etiquette to *subscribe* to the list before posting. If you don't, it usually means someone will have to manually check that your email isn't spam before allowing it to be posted to the list. It may seem annoying to join yet another mailing list, but we highly recommend it: it is a great place to learn!

Accessing the book materials

All of the code from this book is available on our GitHub repository link.

Diving in

We've brought together some of the most elegant code offered up by the SciPy community. Along the way we are going to explore some real-world scientific problems that SciPy can solve. This book is also a glimpse into a welcoming scientific coding community that wants you to join in.

Welcome to Elegant SciPy.

Now, let's write some code!

CHAPTER 1

Elegant NumPy: The Foundation of Scientific Python

This chapter touches on some statistical functions in SciPy, but more than that, it focuses on exploring the NumPy array, a data structure that underlies almost all numerical scientific computation in Python. We will see how NumPy array operations enable concise and efficient code when manipulating numerical data.

Our use case is using gene expression data from The Cancer Genome Atlas (TCGA) project to predict mortality in skin cancer patients. We will working towards this goal throughout Chapter 1 and Chapter 2, learning about some key SciPy concepts along the way. Before we can predict mortality, we will need to normalize the expression data using a method called RPKM normalization. This allows the comparison of measurements between different samples and genes. (We will unpack what "gene expression" means in just a moment.)

Let's start with a code snippet to tantalize, and motivate the ideas in this chapter. As we will do in each chapter, we open with a code sample that we believe epitomizes the elegance and power of a particular function from the SciPy ecosystem. In this case, we want to highlight NumPy's vectorization and broadcasting rules, which allow us to manipulate and reason about data arrays very efficiently.

```
In [1]:

def rpkm(counts, lengths):
    """Calculate reads per kilobase transcript per million reads.
    RPKM = (10^9 * C) / (N * L)

    Where:
    C = Number of reads mapped to a gene
    N = Total mapped reads in the experiment
    L = Exon length in base pairs for a gene
```

```
    counts: 2D numpy ndarray (numerical)
        RNAseq (or similar) count data where columns are individual samples
        and rows are genes.
    lengths: list or 1D numpy ndarray (numerical)
        Gene lengths in base pairs in the same order
        as the rows in counts.
    """

    N = np.sum(counts, axis=0)  # sum each column to get total reads per sample
    L = lengths
    C = counts

    rpkm = ( (10e9 * C) / N[np.newaxis, :] ) / L[:, np.newaxis]

    return(rpkm)
```

This example illustrates some of the ways that NumPy arrays can make your code more elegant:

- Arrays can be one-dimensional, like lists, but they can also be two-dimensional, like matrices, and higher-dimensional still. This allows them to represent many different kinds of numerical data. In our case, we are manipulating a 2D matrix.
- Arrays can be operated on along *axes*. In the first line, we calculate the sum down each column by specifying a *different* `axis`.
- Arrays allow the expression of many numerical operations at once. For example towards the end of the function we divide the 2D array of counts (C) by the 1D array of column sums (N). This is broadcasting. More on how this works in just a moment!

Before we delve into the power of NumPy, let's spend some time to understand the biological data that we will be working with.

What is gene expression?

We will work our way through a *gene expression analysis* to demonstrate the power of NumPy and SciPy to solve a real-world biological problem. We will use the Pandas library, which builds on NumPy, to read and munge our data files, and then we manipulate our data efficiently in NumPy arrays.

The so-called central dogma of molecular biology states that all the information needed to run a cell (or an organism, for that matter) is stored in a molecule called *deoxyribonucleic acid*, or DNA. This molecule has a repetitive backbone on which lie chemical groups called *bases*, in sequence. There are four kinds of bases, abbreviated to A, C, G, and T, constituting an alphabet with which information is stored.

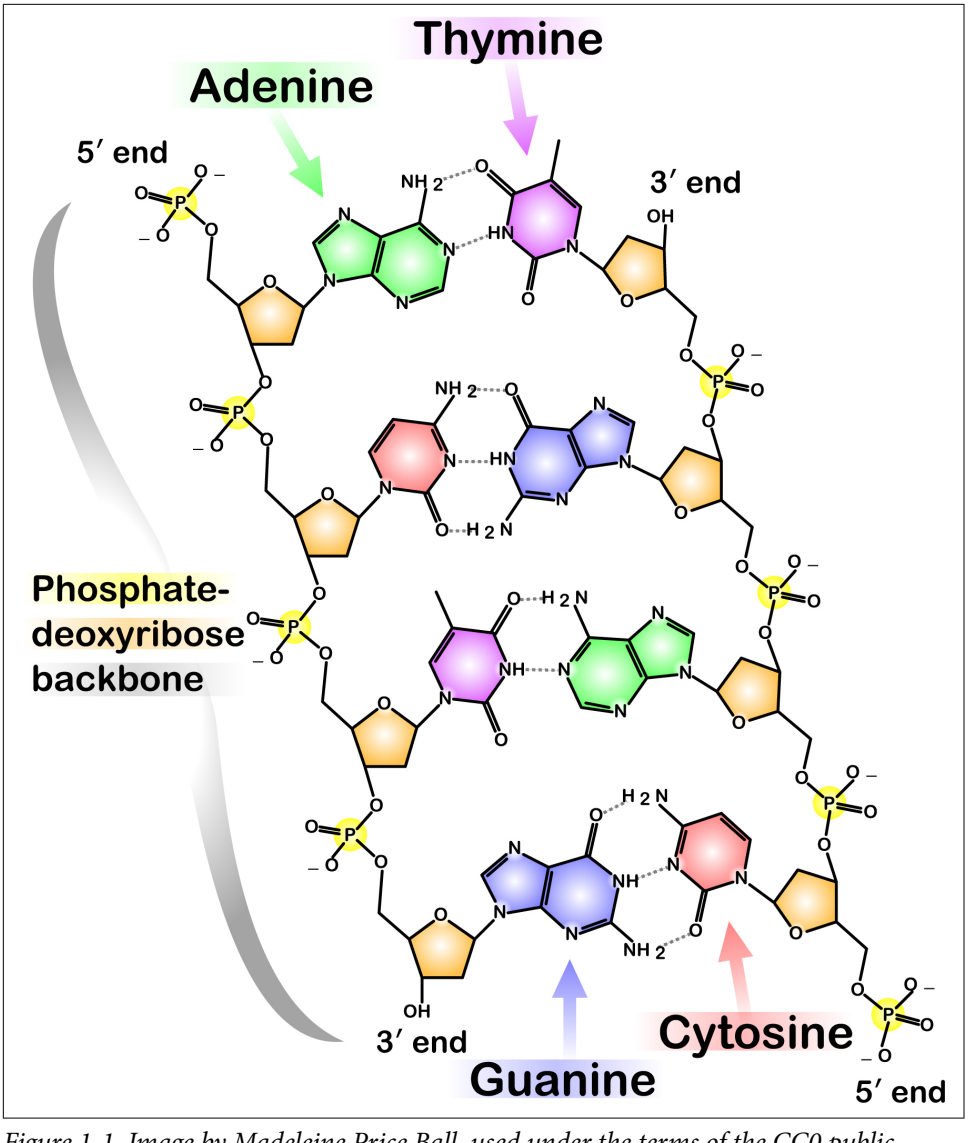
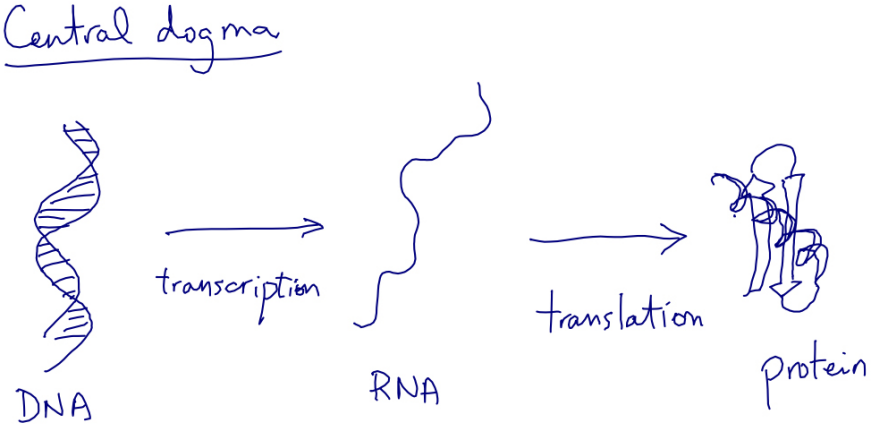


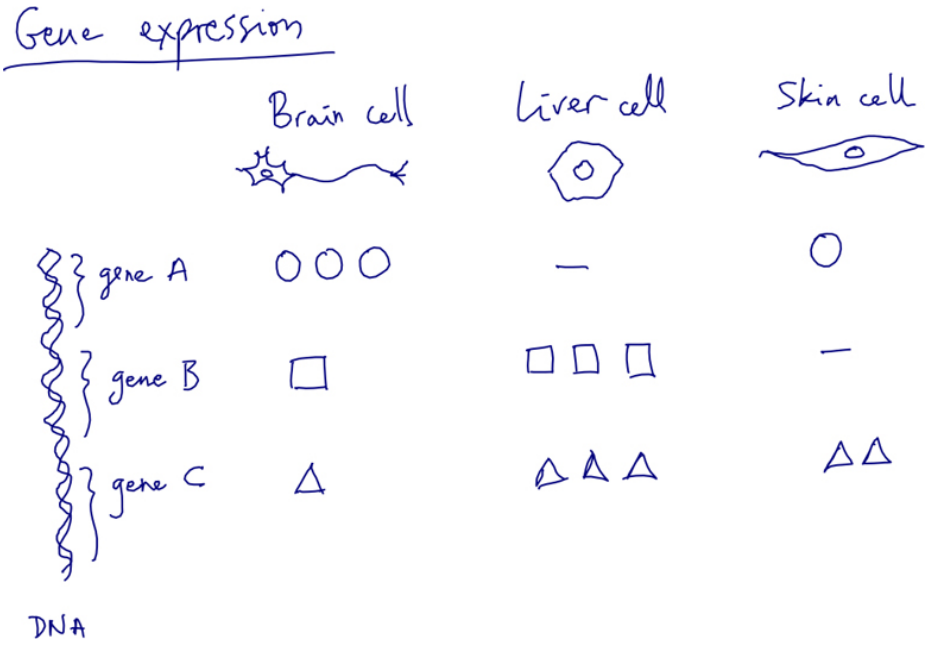
Figure 1-1. Image by Madeleine Price Ball, used under the terms of the CC0 public domain license

To access this information, the DNA is *transcribed* into a sister molecule called *messenger ribonucleic acid*, or mRNA. Finally, this mRNA is *translated* into proteins, the workhorses of the cell. A section of DNA that encodes the information to make a protein (via mRNA) is called a gene.

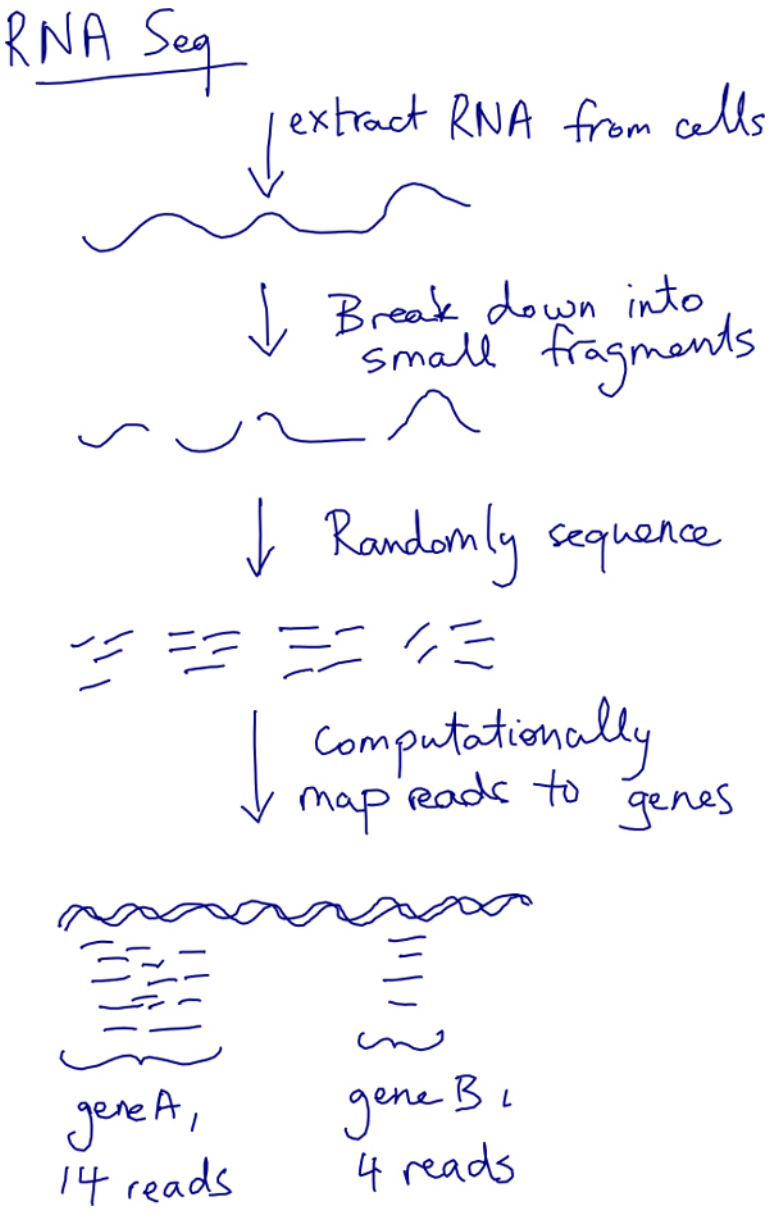
The amount of mRNA produced from a given gene is called the *expression* of that gene. Although we would ideally like to measure protein levels, this is a much harder task than measuring mRNA. Fortunately, expression levels of an mRNA and levels of its corresponding protein are usually correlated (Maier, Güell, and Serrano, 2009). Therefore, we usually measure mRNA levels and base our analyses on that. As you will see below, it often doesn't matter, because we are using mRNA levels for their power to predict biological outcomes, rather than to make specific statements about proteins.



It's important to note that the DNA in every cell of your body is identical. Thus, the differences between cells arise from *differential expression* of that DNA into RNA. Similarly, as we shall see in this chapter, differential expression can distinguish different kinds of cancer.



The state-of-the-art technology to measure mRNA is RNA sequencing (RNAseq). RNA is extracted from a tissue, for example from a biopsy from a patient, *reverse transcribed* back into DNA (which is more stable), and then read out using chemically modified bases that glow when they are incorporated into the DNA sequence. Currently, high-throughput sequencing machines can only read short fragments (approximately 100 bases is common). These short sequences are called "reads". We measure millions of reads and then based on their sequence we count how many reads came from each gene. For this chapter we'll be starting directly from this count data, but in [ch7?] we will talk more about how this type of data can be determined.



Here's an example of what this gene expression data looks like.

	Cell type A	Cell type B
Gene 0	100	200
Gene 1	50	0

	Cell type A	Cell type B
Gene 2	350	100

The data is a table of counts, integers representing how many reads were observed for each gene in each cell type. See how the counts for each gene differ between the cell types? We can use this information to tell us about the differences between these two types of cell.

One way to represent this data in Python would be as a list of lists:

```
In [2]:

gene0 = [100, 200]
gene1 = [50, 0]
gene2 = [350, 100]
expression_data = [gene0, gene1, gene2]
```

Above, each gene's expression across different cell types is stored in a list of Python integers. Then, we store all of these lists in a list (a meta-list, if you will). We can retrieve individual data points using two levels of list indexing:

```
In [3]:

expression_data[2][0]

Out[3]:
350
```

It turns out that, because of the way the Python interpreter works, this is a very inefficient way to store these data points. First, Python lists are always lists of *objects*, so that the above list `gene2` is not a list of integers, but a list of *pointers* to integers, which is unnecessary overhead. Additionally, this means that each of these lists and each of these integers end up in a completely different, random part of your computer's RAM. However, modern processors actually like to retrieve things from memory in *chunks*, so this spreading of the data throughout the RAM is very bad.

This is precisely the problem solved by the *NumPy array*.

NumPy N-dimensional arrays

One of the key NumPy data types is the N-dimensional array (ndarray, or just array). Arrays must be homogeneous; all items in an array must be the same type. In our case we will need to store integers.

Ndarrays are called N-dimensional because they can have any number of dimensions. A 1-dimesional array is roughly equivalent to a Python list:

```
In [4]:

import numpy as np
```

```
one_d_array = np.array([1,2,3,4])
print(one_d_array)
print(type(one_d_array))
```

```
[1 2 3 4]
<class 'numpy.ndarray'>
```

Arrays have particular attributes and methods, that you can access by placing a dot after the array name. For example, you can get the array's *shape*:

```
In [5]:
```

```
print(one_d_array.shape)
```

```
(4,)
```

Here, it's just a tuple with a single number. You might wonder why you wouldn't just use `len`, as you would for a list. That will work, but it doesn't extend to *two-dimensional* arrays.

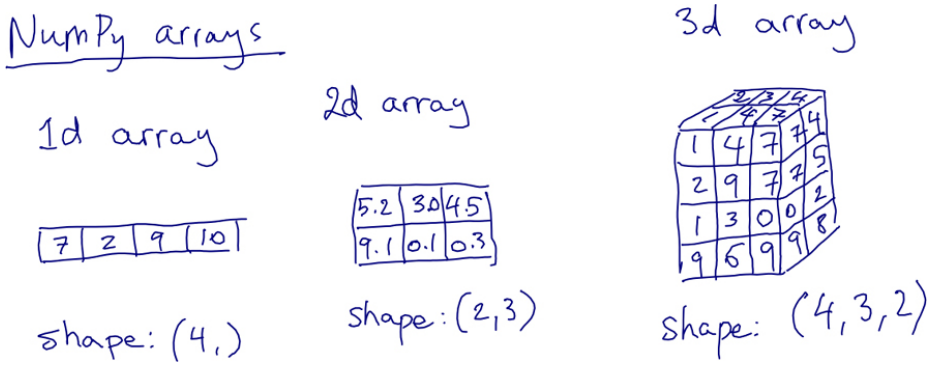
This is what we use to represent our mini gene expression table from above:

```
In [6]:
```

```
two_d_array = np.array(expression_data)
print(two_d_array.shape)
```

```
(3, 2)
```

Now you can see that the `shape` attribute generalises `len` to account for the size of multiple dimensions of an array of data.



Arrays have other attributes, such as `ndim`, the number of dimensions:

```
In [7]:
```

```
print(two_d_array.ndim)
```

```
2
```

You'll become familiar with all of these as you start to use NumPy more for your own data analysis.

NumPy arrays can represent data that has even more dimensions, such as magnetic resonance imaging (MRI) data, which includes measurements within a 3D volume. If we store MRI values over time, we might need a 4D NumPy array.

For now, we'll stick to 2D data. Later chapters will introduce higher-dimensional data and will teach you to write code that works for data of any number of dimensions!

Why use ndarrays as opposed to Python lists?

Arrays are fast because they enable vectorized operations, written in the low-level language C, that act on the whole array. Say you have a list and you want to multiply every element in the list by 5. A standard Python approach would be to write a loop that iterates over the elements of the list and multiply each one by 5. However, if your data were instead represented as an array, you can multiply every element in the array by 5 in a single bound. Behind the scenes, the highly-optimized NumPy library is doing the iteration as fast as possible.

```
In [8]:

import numpy as np

# Create an ndarray of integers in the range
# 0 up to (but not including) 10,000,000
nd_array = np.arange(1e6)
# Convert arr to a list
list_array = nd_array.tolist()

In [9]:

%%timeit -n10
# Time how long it takes to multiply each element in the list by 5
for i, val in enumerate(list_array):
    list_array[i] = val * 5

10 loops, best of 3: 239 ms per loop

In [10]:

%%timeit -n10
# Use the IPython "magic" command timeit to time how
# long it takes to multiply each element in the ndarray by 5
x = nd_array * 5

10 loops, best of 3: 2.76 ms per loop
```

More than 100 times faster, and more concise, too!

Arrays are also size efficient. In Python, each element in a list is an object and is given a healthy memory allocation (or is that unhealthy?). In contrast, in arrays, each element takes up just the necessary amount of memory. For example, an array of 64-bit

integers takes up exactly 64-bits per element, plus some very small overhead for array metadata, such as the `shape` attribute we discussed above. This is generally much less than would be given to objects in a python list. (If you're interested in digging into how Python memory allocation works, check out Jake VanderPlas' blog post Why Python is Slow: Looking Under the Hood.)

Plus, when computing with arrays, you can also use *slices* that subset the array *without copying the underlying data*.

```
In [11]:

# Create an ndarray x
x = np.array([1, 2, 3], np.int32)
print(x)

[1 2 3]

In [12]:

# Create a "slice" of x
y = x[:2]
print(y)

[1 2]

In [13]:

# Set the first element of y to be 6
y[0] = 6
print(y)

[6 2]
```

Notice that although we edited `y`, `x` has also changed, because `y` was referencing the same data!

```
In [14]:

# Now the first element in x has changed to 6!
print(x)

[6 2 3]
```

This does mean you have to be careful with array references. If you want to manipulate the data without touching the original, it's easy to make a copy:

```
In [15]:
y = np.copy(x[:2])
```

Vectorization

Earlier we talked about the speed of operations on arrays. Once of the tricks Numpy uses to speed things up is *vectorization*. Vectorization is where you apply a calculation to each element in an array, without having to use a for loop. In addition to speeding things up, this can result in more natural, readable code. Let's look at some examples.

```
In [16]:
x = np.array([1, 2, 3, 4])
print(x * 2)
[2 4 6 8]
```

Here, we have `x`, an array of 4 values, and we have implicitly multiplied every element in `x` by 2, a single value.

```
In [17]:

y = np.array([0, 1, 2, 1])
print(x + y)

[1 3 5 5]
```

Now, we have added together each element in `x` to its corresponding element in `y`, an array of the same shape.

Both of these operations are simple and, we hope, intuitive examples of vectorization. NumPy also makes them very fast, much faster than iterating over the arrays manually. (Feel free to play with this yourself using the `%%timeit` IPython magic.)

Broadcasting

One of the most powerful and often misunderstood features of ndarrays is broadcasting. Broadcasting is a way of performing implicit operations between two arrays. It allows you to perform operations on arrays of *compatible* shapes, to create arrays bigger than either of the starting ones. For example, we can compute the outer product of two vectors, by reshaping them appropriately:

```
In [18]:

x = np.array([1, 2, 3, 4])
x = np.reshape(x, (len(x), 1))
print(x)

[[1]
 [2]
 [3]
 [4]]

In [19]:

y = np.array([0, 1, 2, 1])
y = np.reshape(y, (1, len(y)))
print(y)

[[0 1 2 1]]
```

In order to do broadcasting, the two arrays have to have the same number of dimensions and the sizes of the dimensions need to match (or be equal to 1). Let's check the shapes of these two arrays.

```
In [20]:
```

```
print(x.shape)
print(y.shape)
```

```
(4, 1)
(1, 4)
```

Both arrays have two dimensions and the inner dimensions of both arrays are 1, so the dimensions are compatible!

```
In [21]:
```

```
outer = x * y
print(outer)
```

```
[[0 1 2 1]
 [0 2 4 2]
 [0 3 6 3]
 [0 4 8 4]]
```

The outer dimensions tell you how size of the resulting array. In our case we expect a (4, 4) array:

```
In [22]:
```

```
print(outer.shape)
```

```
(4, 4)
```

You can see for yourself that `outer[i, j] = x[i] * y[j]` for all `(i, j)`.

This was accomplished by NumPy's broadcasting rules, which implicitly expand dimensions of size 1 in one array to match the corresponding dimension of the other array. Don't worry, we will talk about these rules in more detail later in this chapter.

As we will see in the rest of the chapter, as we explore real data, broadcasting is extremely valuable to perform real-world calculations on arrays of data. It allows us to express complex operations concisely and efficiently.

Exploring a gene expression data set

The data set that we'll be using is an RNAseq experiment of skin cancer samples from The Cancer Genome Atlas (TCGA) project (http://cancergenome.nih.gov/). In Chapter 2 we will be using this gene expression data to predict mortality in skin cancer patients, reproducing a simplified version of Figures 5A and 5B of a paper from the TCGA consortium. But first we need to get our heads around the biases in our data, and think about how we could improve it.

Downloading the data

[Links to data!]

We're first going to use Pandas to read in the table of counts. Pandas is a Python library for data manipulation and analysis, with particular emphasis on tabular and time series data. Here, we will use it here to read in tabular data of mixed type. It uses the DataFrame type, which is a flexible tabular format based on the data frame object in R. For example the data we will read has a column of gene names (strings) and multiple columns of counts (integers), so it doesn't make sense to read this data in directly as an ndarray. By reading the data in as a Pandas DataFrame we can let Pandas do all the parsing, then extract out the relevant information and store it in a more efficient data type. Here we are just using Pandas briefly to import data. In later chapters we will give you some more insight into the world of Pandas.

In [23]:

```
import numpy as np
import pandas as pd

# Import TCGA melanoma data
filename = 'data/counts.txt'
with open(filename, 'rt') as f:
    data_table = pd.read_csv(f, index_col=0) # Parse file with pandas

print(data_table.iloc[:5, :5])
```

```
       00624286-41dd-476f-a63b-d2a5f484bb45  TCGA-FS-A1Z0  TCGA-D9-A3Z1   \
A1BG                                1272.36        452.96        288.06
A1CF                                   0.00          0.00          0.00
A2BP1                                  0.00          0.00          0.00
A2LD1                                164.38        552.43        201.83
A2ML1                                 27.00          0.00          0.00

       02c76d24-f1d2-4029-95b4-8be3bda8fdbe  TCGA-EB-A51B
A1BG                                 400.11        420.46
A1CF                                   1.00          0.00
A2BP1                                  0.00          1.00
A2LD1                                165.12         95.75
A2ML1                                  0.00          8.00
```

We can see that Pandas has kindly pulled out the header row and used it to name the columns. The first column gives the name of the gene, and the remaining columns represent individual samples.

We will also needs some corresponding metadata, including the sample information and the gene lengths.

In [24]:

```
# Sample names
samples = list(data_table.columns)
```

We will need some information about the lengths of the genes for our normalization. So that we can take advantage of some fancy pandas indexing, we're going to set the index of the pandas table to be the gene names in the first column.

```
In [25]:
```

```
# Import gene lengths
filename = 'data/genes.csv'
with open(filename, 'rt') as f:
    gene_info = pd.read_csv(f, index_col=0) # Parse file with pandas, index by GeneSymbol
print(gene_info.iloc[:5, :])
```

```
            GeneID  GeneLength
GeneSymbol
CPA1          1357        1724
GUCY2D        3000        3623
UBC           7316        2687
C11orf95     65998        5581
ANKMY2       57037        2611
```

Let's check how well our gene length data matches up with our count data.

```
In [26]:
```

```
print("Genes in data_table: ", data_table.shape[0])
print("Genes in gene_info: ", gene_info.shape[0])
```

```
Genes in data_table: 20500
Genes in gene_info: 20503
```

There are more genes in our gene length data than were actually measured in the experiment. Let's filter so we only get the relevant genes, and we want to make sure they are in the same order as in our count data. This is where pandas indexing comes in handy! We can get the intersection of the gene names from our our two sources of data and use these to index both data sets, ensuring they have the same genes in the same order.

```
In [27]:
```

```
#Subset gene info to match the count data
matched_index = pd.Index.intersection(data_table.index, gene_info.index)
```

Now let's use the intersection of the gene names to index our count data.

```
In [28]:
```

```
# 2D ndarray containing expression counts for each gene in each individual
counts = np.asarray(data_table.loc[matched_index], dtype=int)

# Check how many genes and individuals were measured
print("{0} genes measured in {1} individuals".format(counts.shape[0], counts.shape[1]))
```

```
20500 genes measured in 375 individuals
```

And our gene lengths.

```
In [29]:

# 1D ndarray containing the lengths of each gene
gene_lengths = np.asarray(gene_info.loc[matched_index]['GeneLength'],
                          dtype=int)
```

And let's check the dimensions of our objects.

```
In [30]:

print(counts.shape)
print(gene_lengths.shape)

(20500, 375)
(20500,)
```

As expected, they now match up nicely!

Normalization

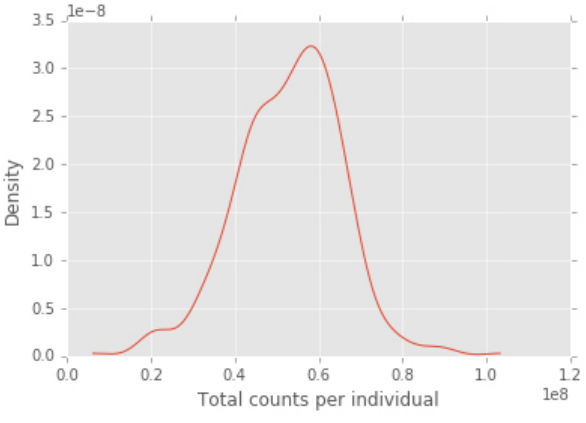
Before we do any kind of analysis with our data, it is important to take a look at it and determine if we need to normalize it first.

Between samples

For example, the number of counts for each individual can vary substantially in RNAseq experiments. Let's take a look at the distribution of expression counts over all the genes. First we will sum the rows to get the total counts of expression of all genes for each individual, so we can just look at the variation between individuals. To visualize the distribution of total counts, we will use a kernel density estimation (KDE) function. KDE is commonly used to smooth out histograms, which gives a clearer picture of the underlying distribution.

```
In [31]:

%matplotlib inline
# Make all plots appear inline in the Jupyter notebook from now onwards

import matplotlib.pyplot as plt
plt.style.use('ggplot') # Use ggplot style graphs for something a little prettier

In [32]:

total_counts = counts.sum(axis=0) # sum each column (axis=1 would sum rows)

from scipy import stats
density = stats.kde.gaussian_kde(total_counts) # Use gaussian smoothing to estimate the densit
y
x = np.arange(min(total_counts), max(total_counts), 10000) # create ndarray of integers from m
in to max in steps of 10,000
plt.plot(x, density(x))
plt.xlabel("Total counts per individual")
plt.ylabel("Density")
```

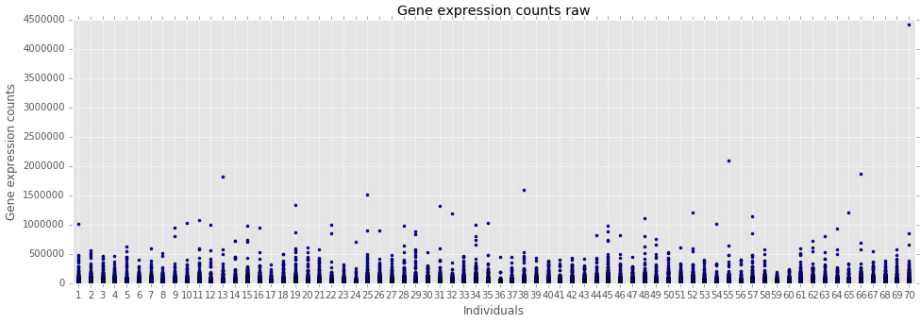
```
plt.show()

print("Min counts: {0}, Mean counts: {1}, Max counts: {2}".format(total_counts.min(), total_co
unts.mean(), total_counts.max()))
```




```
Min counts: 6231205, Mean counts: 52995255.33866667, Max counts: 103219262
```

We can see that there is an order of magnitude difference in the total number of counts between the lowest and the highest individual. This means that a different number of RNAseq reads were generated for each individual. We say that these individuals have different library sizes.

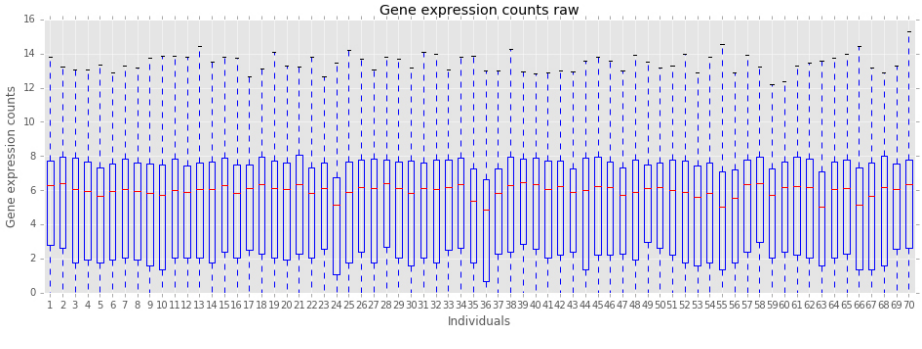
```
In [33]:

# Subset data for plotting
np.random.seed(seed=7) # Set seed so we will get consistent results
samples_index = np.random.choice(range(counts.shape[1]), size=70, replace=False) # Randomly se
lect 70 samples
counts_subset = counts[:,samples_index]

In [34]:

# Bar plot of expression counts by individual
plt.figure(figsize=(16,5))
plt.boxplot(counts_subset, sym=".")
plt.title("Gene expression counts raw")
plt.xlabel("Individuals")
plt.ylabel("Gene expression counts")
plt.show()
```



There are obviously a lot of outliers at the high expression end of the scale and a lot of variation between individuals, but pretty hard to see because everything is clustered around zero. So let's do log(n + 1) of our data so it's a bit easier to look at. Both the log function and the n + 1 step can be done using broadcasting to simplify our code and speed things up.

In [35]:

```
# Bar plot of expression counts by individual
plt.figure(figsize=(16,5))
plt.boxplot(np.log(counts_subset + 1), sym=".")
plt.title("Gene expression counts raw")
plt.xlabel("Individuals")
plt.ylabel("Gene expression counts")
plt.show()
```

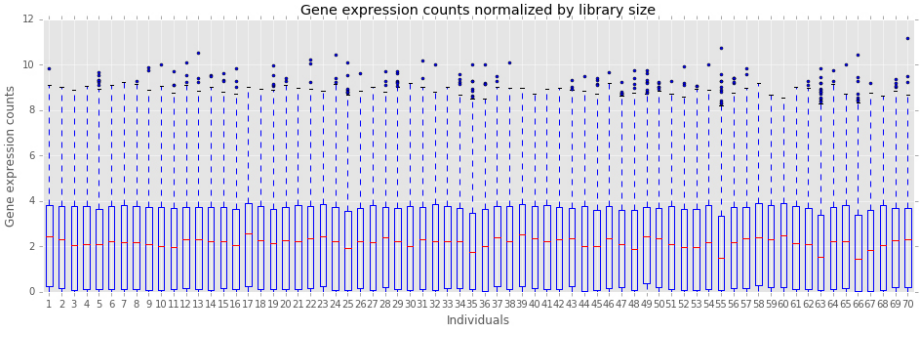


Now let's see what happens when we normalize by library size.

In [36]:

```
# normalize by library size
# Divide the expression counts by the total counts for that individual
counts_lib_norm = counts / total_counts * 1000000 # Multiply by 1 million to get things back i
n a similar scale
```

```
# Notice how we just used broadcasting twice there!
counts_subset_lib_norm = counts_lib_norm[:,samples_index]

# Bar plot of expression counts by individual
plt.figure(figsize=(16,5))
plt.boxplot(np.log(counts_subset_lib_norm + 1), sym=".")
plt.title("Gene expression counts normalized by library size")
plt.xlabel("Individuals")
plt.ylabel("Gene expression counts")
plt.show()
```



Much better! Also notice how we used broadcasting twice there. Once to divide all the gene expression counts by the total for that column, and then again to multiply all the values by 1 million.

[ED'S NOTE]: the following function will probably be replaced by Seaborn's new boxplot function, which supports exactly this use case.

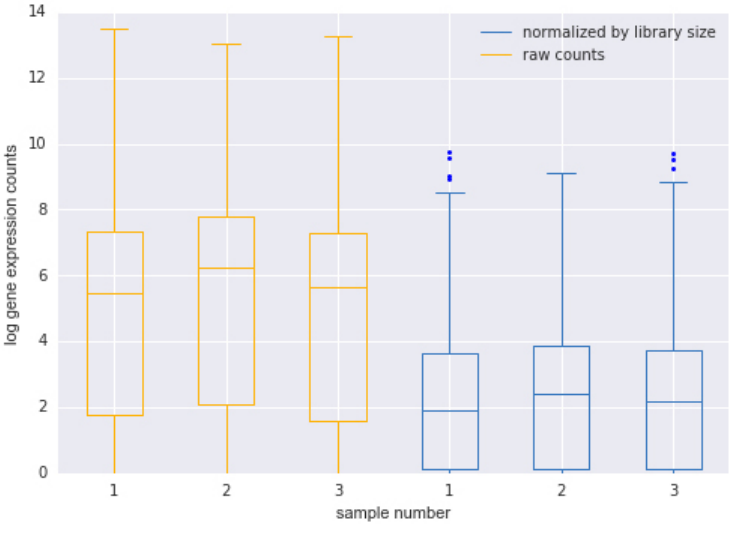
```
In [37]:
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import itertools as it

def class_boxplot(data, classes, colors=None, **kwargs):
    """Make a boxplot with boxes colored according to the class they belong to.

    Parameters
    ----------
    data : list of array-like of float
        The input data. One boxplot will be generated for each element
        in `data`.
    classes : list of string, same length as `data`
        The class each distribution in `data` belongs to.
    colors : list of matplotlib colorspecs
        The color corresponding to each class. These will be cycled in
        the order in which the classes appear in `classes`. (So it is
        ideal to provide as many colors as there are classes! The
```

```
        default palette contains five colors.)

    Other parameters
    ----------------
    kwargs : dict
        Keyword arguments to pass on to `plt.boxplot`.
    """
    # default color palette
    if colors is None:
        colors = sns.xkcd_palette(["windows blue", "amber", "greyish",
                                   "faded green", "dusty purple"])
    # default boxplot parameters; only updated if not specified
    kwargs['sym'] = kwargs.get('sym', '.')
    kwargs['whiskerprops'] = kwargs.get('whiskerprops', {'linestyle': '-'})

    all_classes = sorted(set(classes))
    class2color = dict(zip(all_classes, it.cycle(colors)))
    # create a dictionary containing data of same length but only data
    # from that class
    class2data = {}
    for i, (distrib, cls) in enumerate(zip(data, classes)):
        for c in all_classes:
            class2data.setdefault(c, []).append([]) # empty dataset at first
        class2data[cls][-1] = distrib
    # then, do each boxplot in turn with the appropriate color
    lines = []
    for cls in all_classes:
        # set color for all elements of the boxplot
        for key in ['boxprops', 'whiskerprops', 'capprops',
                    'medianprops', 'flierprops']:
            kwargs.setdefault(key, {}).update(color=class2color[cls])
        # draw the boxplot
        box = plt.boxplot(class2data[cls], **kwargs)
        lines.append(box['caps'][0])
    plt.legend(lines, all_classes)
```

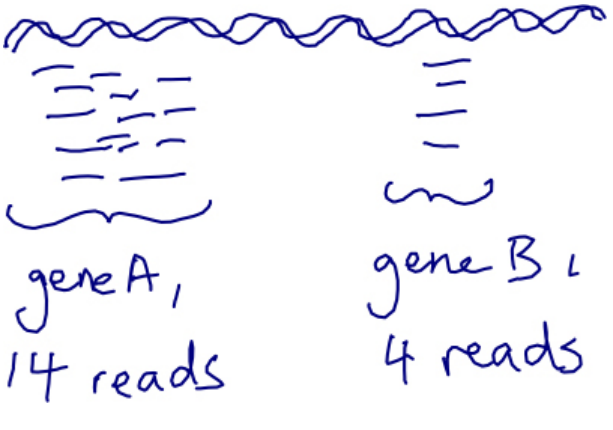
Now we can plot a colored boxplot according to normalized vs unnormalized samples. We show only three samples from each class for illustration:

```
In [38]:

log_counts_3 = list(np.log(counts.T[:3] + 1))
log_ncounts_3 = list(np.log(counts_lib_norm.T[:3] + 1))
class_boxplot(log_counts_3 + log_ncounts_3,
              ['raw counts'] * 3 + ['normalized by library size'] * 3,
              labels=[1, 2, 3, 1, 2, 3])
plt.xlabel('sample number')
plt.ylabel('log gene expression counts')
plt.show()
```



Between genes

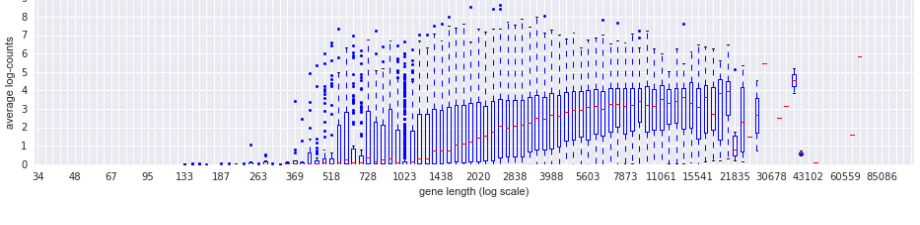
The number of counts for a gene, is related to the gene length. Let's say we have gene A and gene B. Gene B is twice as long as gene A. Both are expressed at similar levels in the sample, i.e. both produce a similar number of mRNA molecules. Therefore you would expect that gene B would have about twice as many counts as gene A. Remember, that when we do an RNAseq experiment, we are fragmenting the transcript, and sampling reads from that pool of fragments. The counts are the number of reads from that gene in a given sample. So if a gene is twice as long, we are twice as likely to sample it.



Let's see if the relationship between gene length and counts plays out in our data set.

In [39]:

```
def binned_boxplot(x, y):
    """
    x: x axis, values to be binned. Should be a 1D ndarray.
    y: y axis. Should be a 1D ndarray.
    """
    # get the "optimal" bin size using astropy's histogram function
    from astropy.stats import histogram
    gene_len_hist, gene_len_bins = histogram(log_gene_lengths, bins='knuth')
    # np.digitize tells you which bin an observation belongs to.
    # we don't use the last bin edge because it breaks the right-open assumption
    # of digitize. The max observation correctly goes into the last bin.
    gene_len_idxs = np.digitize(log_gene_lengths, gene_len_bins[:-1])
    # Use those indices to create a list of arrays, each containing the log
    # counts corresponding to genes of that length. This is the input expected
    # by plt.boxplot
    binned_counts = [mean_log_counts[gene_len_idxs == i]
                     for i in range(np.max(gene_len_idxs))]
    plt.figure(figsize=(16,3))
    # Make the x-axis labels using real gene length
    gene_len_bin_centres = (gene_len_bins[1:] + gene_len_bins[:-1]) / 2
    gene_len_labels = np.round(np.exp(gene_len_bin_centres)).astype(int)
    # use only every 5th label to prevent crowding on x-axis ticks
    labels = []
    for i, lab in enumerate(gene_len_labels):
        if i % 5 == 0:
            labels.append(str(lab))
        else:
            labels.append('')
    # make the boxplot
    plt.boxplot(binned_counts, labels=labels, sym=".")
    # Adjust the axis names
    plt.xlabel('gene length (log scale)')
    plt.ylabel('average log-counts')
    plt.show()

log_counts = np.log(counts_lib_norm + 1)
mean_log_counts = np.mean(log_counts, axis=1)
log_gene_lengths = np.log(gene_lengths)

binned_boxplot(x=log_gene_lengths, y=mean_log_counts)
```



We can see a positive relationship between the length of a gene and the counts!

Normalizing over samples and genes: RPKM

One of the simplest normalization methods for RNAseq data is RPKM: reads per kilobase transcript per million reads. RPKM puts together the ideas of normalising by sample and by gene. When we calculate RPKM, we are normalizing for both the library size (the sum of each column) and the gene length.

Working through how RPKM is derived:

Let's say:

- C = Number of reads mapped to a gene
- L = exon length in base-pairs for a gene
- N = Total mapped reads in the experiment

First, let's calculate reads per kilobase.

Reads per base would be:

$$\frac{C}{L}$$

The formula asks for reads per kilobase instead of reads per base. One kilobase = 1000 bases, so we'll need to divide length (L) by 1000.

Reads per kilobase would be:

$$\frac{C}{L/1000} = \frac{10^3 C}{L}$$

Next, we need to normalize by library size. If we just divide by the number of mapped reads we get:

$$\frac{10^3 C}{LN}$$

But biologists like thinking in millions of reads so that the numbers don't get too big. Counting per million reads we get:

$$\frac{10^3 C}{L\left(N/10^6\right)} = \frac{10^9 C}{LN}$$

In summary, to calculate reads per kilobase transcript per million reads:

$$RPKM = \frac{10^9 C}{LN}$$

Now let's implement RPKM over the entire counts array.

```
In [40]:

# Make our variable names the same as the RPKM formula so we can compare easily
C = counts
N = counts.sum(axis=0)  # sum each column to get total reads per sample
L = gene_lengths  # lengths for each gene, matching rows in `C`
```

First, we multiply by 10^9. Because counts (C) is an ndarray, we can use broadcasting. If we multiple an ndarray by a single value, that value is broadcast over the entire array.

```
In [41]:

# Multiply all counts by 10^9
C_tmp = 10^9 * C
```

Next we need to divide by the gene length. Broadcasting a single value over a 2D array was pretty clear. We were just multiplying every element in the array by the value. But what happens when we need to divide a 2D array by a 1D array?

Broadcasting rules

Broadcasting allows calculations between ndarrays that have a different number of dimensions.

If the input arrays do not have the same number of dimensions, then then an additional dimension is added to the start of the first array, with a value of 1. Once the two arrays have the same number of dimensions, broadcasting can only occur if the sizes of the dimensions match, or one of them is equal to 1.

For example, let's say we have two ndarrays, A and B:

A. shape = (1, 2)

B. shape = (2,)

If we performed the operation `A * B` then broadcasting would occur. B has fewer dimension than A, so during the calculation a new dimension is prepended to B with value 1.

B.shape = (1, 2)

Now A and B have the same number of dimensions, so broadcasting can proceed.

Now let's say we have another array, C:

C.shape = (2, 1)

B.shape = (2,)

Now, if we were to do the operation `C * B`, a new dimension needs to be prepended to B.

B.shape = (1, 2)

However, the dimensions of the two ndarrays do not match, so broadcasting will fail.

Let's say that we know that it is appropriate to broadcast B over C. We can explicitly add a new dimension to B using `np.newaxis`. Let's see this in our normalization by RPKM.

Let's have a look at the dimensions of our two arrays.

```
In [42]:

# Check the shapes of C_tmp and L
print('C_tmp.shape', C_tmp.shape)
print('L.shape', L.shape)

C_tmp.shape (20500, 375)
L.shape (20500,)
```

We can see that `C_tmp` has 2 dimensions, while L has one. So during broadcasting, an additional dimension will be prepended to L. Then we will have:

```
C_tmp.shape (20500, 375)
L.shape (1, 20500)
```

The dimensions won't match! We want to broadcast L over the first dimension of `C_tmp`, so we need to adjust the dimensions of L ourselves.

```
In [43]:

L = L[:, np.newaxis] # append a dimension to L, with value 1
print('C_tmp.shape', C_tmp.shape)
print('L.shape', L.shape)

C_tmp.shape (20500, 375)
L.shape (20500, 1)
```

Now that our dimensions match or are equal to 1, we can broadcast.

```
In [44]:

# Divide each row by the gene length for that gene (L)
C_tmp = C_tmp / L
```

Finally we need to normalize by the libaray size, the total number of counts for that column.

Remember that we have already calculated N.

```
N = counts.sum(axis=0) # sum each column to get total reads per sample
```

```
In [45]:
```

```
# Check the shapes of C_tmp and N
print('C_tmp.shape', C_tmp.shape)
print('N.shape', N.shape)
```

```
C_tmp.shape (20500, 375)
N.shape (375,)
```

Once we trigger broadcasting, an additional dimension will be prepended to N: N.shape (1, 375)

The dimensions will match so we don't have to do anything. However, for readability, it can be useful to add the extra dimension to N anyway.

```
In [46]:
```

```
# Divide each column by the total counts for that column (N)
N = N[np.newaxis, :]
print('C_tmp.shape', C_tmp.shape)
print('N.shape', N.shape)
```

```
C_tmp.shape (20500, 375)
N.shape (1, 375)
```

```
In [47]:
```

```
# Divide each column by the total counts for that column (N)
rpkm_counts = C_tmp / N
```

Let's put this in a function so we can reuse it.

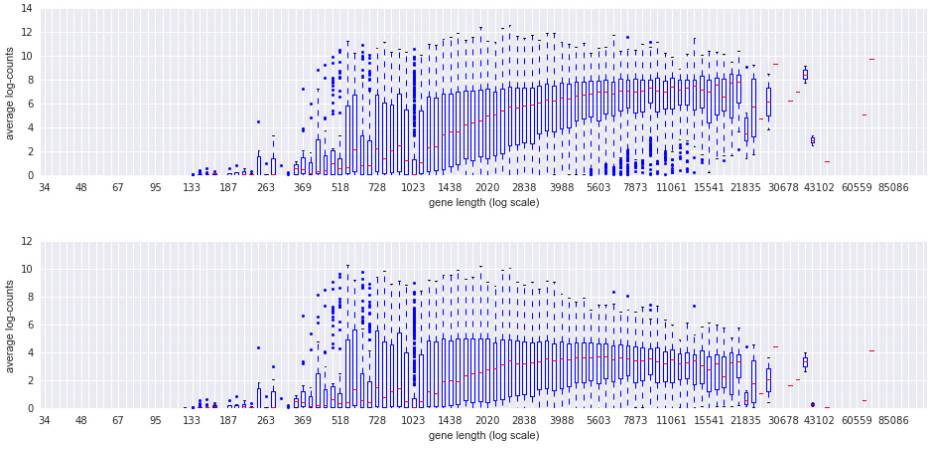
```
In [48]:
```

```
def rpkm(counts, lengths):
    """Calculate reads per kilobase transcript per million reads.
    RPKM = (10^9 * C) / (N * L)

    Where:
    C = Number of reads mapped to a gene
    N = Total mapped reads in the experiment
    L = Exon length in base pairs for a gene

    counts: 2D numpy ndarray (numerical)
        RNAseq (or similar) count data where columns are individual samples
        and rows are genes.
    lengths: list or 1D numpy ndarray (numerical)
        Gene lengths in base pairs in the same order
        as the rows in counts.
    """

    N = np.sum(counts, axis=0) # sum each column to get total reads per sample
    L = lengths
```

```
    C = counts

    rpkm = ( (10e9 * C) / N[np.newaxis, :] ) / L[:, np.newaxis]

    return(rpkm)

counts_rpkm = rpkm(counts, gene_lengths)
```

In [49]:

```
# Repeat binned boxplot with raw values
log_counts = np.log(counts + 1)
mean_log_counts = np.mean(log_counts, axis=1)
log_gene_lengths = np.log(gene_lengths)

binned_boxplot(x=log_gene_lengths, y=mean_log_counts)

# Repeat binned boxplot with RPKM values
log_counts = np.log(counts_rpkm + 1)
mean_log_counts = np.mean(log_counts, axis=1)
log_gene_lengths = np.log(gene_lengths)

binned_boxplot(x=log_gene_lengths, y=mean_log_counts)
```

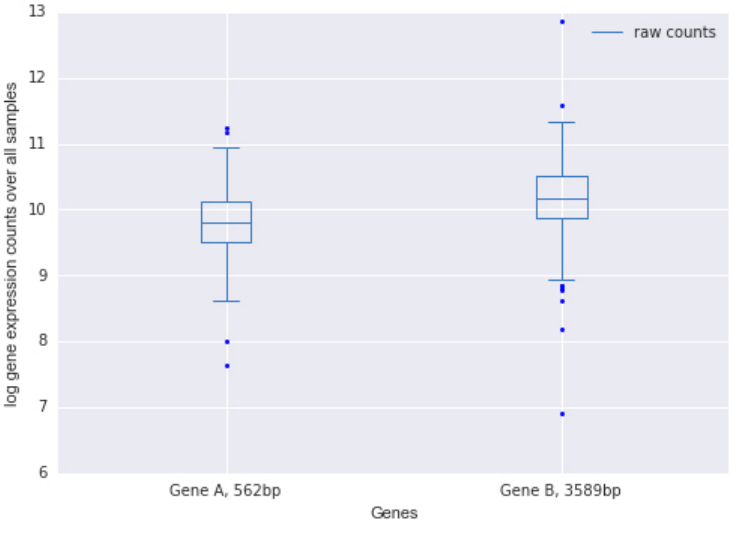


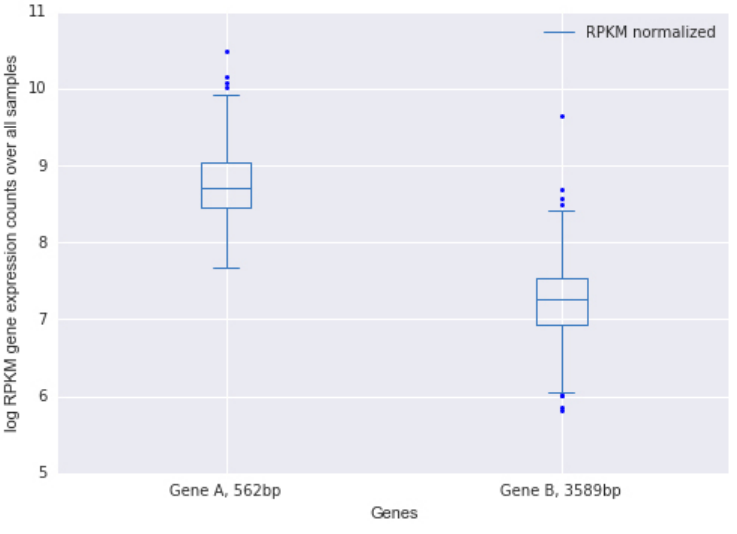
RPMK normalization can be particularly useful comparing the expression profile of two different genes. We've already seen that longer genes have higher counts, but this doesn't mean their expression level is actually higher. Let's choose a short gene and a long gene and compare their counts before and after RPKM normalization to see what we mean.

In [50]:

```
# Boxplot of expression from a short gene vs. a long gene
# showing how normalization can influence interpretation
```

```
genes2_idx = [80, 186]
genes2_lengths = gene_lengths[genes2_idx]
genes2_labels = ['Gene A, {}bp'.format(genes2_lengths[0]), 'Gene B,
{}bp'.format(genes2_lengths[1])]

log_counts_2 = list(np.log(counts[genes2_idx] + 1))
log_ncounts_2 = list(np.log(counts_rpkm[genes2_idx] + 1))

class_boxplot(log_counts_2,
              ['raw counts'] * 3,
              labels=genes2_labels)
plt.xlabel('Genes')
plt.ylabel('log gene expression counts over all samples')
plt.show()

class_boxplot(log_ncounts_2,
              ['RPKM normalized'] * 3,
              labels=genes2_labels)
plt.xlabel('Genes')
plt.ylabel('log RPKM gene expression counts over all samples')
plt.show()
```





Just looking at the raw counts, it looks like the longer gene B is expressed slightly more than gene A. Yet once we normalize to RPKM values, the story changes substantially. Now it looks like gene A is actually expressed at a much higher level than gene B. This is because RPKM includes normalization for gene length, so we can now directly compare between genes of dramatically different lengths.

Taking stock

So far we have, imported data using Pandas, gotten to know the key NumPy data type: the ndarray, and used the power of broadcasting to make our calculations more elegant.

In Chapter 2 we will continue working with the same data set, implementing a more sophisticated normalization technique, and then using clustering to make some predictions about mortality in skin cancer patients.

CHAPTER 2

Quantile normalization with NumPy and SciPy

Our use case is using gene expression data to predict mortality in skin cancer patients. We will reproduce a simplified version of Figures 5A and 5B from this paper, which comes from The Cancer Genome Atlas (TCGA) project.

The code we will work to understand is an implementation of *quantile normalization*, a technique that ensures measurements fit a specific distribution. This requires a strong assumption: if the data are not distributed according to a bell curve, we just make it fit! But it turns out to be simple and useful in many cases where the specific distribution doesn't matter, but the relative changes of values within a population are important. For example, Bolstad and colleagues showed that it performs admirably in recovering known expression levels in microarray data.

Using NumPy indexing tricks and the `scipy.stats.mstats.rankdata` function, quantile normalization in Python is fast, efficient, and elegant.

```
In [1]:
```

```
import numpy as np
from scipy import stats

def quantile_norm(X):
    """Normalize the columns of X to each have the same distribution.

    Given an expression matrix (microarray data, read counts, etc) of M genes
    by N samples, quantile normalization ensures all samples have the same
    spread of data (by construction).

    The input data is log-transformed, then the data across each row are
    averaged to obtain an average column. Each column quantile is replaced
    with the corresponding quantile of the average column.
```

```
    The data is then transformed back to counts.

    Parameters
    ----------
    X : 2D array of float, shape (M, N)
        The input data, with M rows (genes/features) and N columns (samples).

    Returns
    -------
    Xn : 2D array of float, shape (M, N)
        The normalized data.
    """
    # log-transform the data
    logX = np.log2(X + 1)

    # compute the quantiles
    log_quantiles = np.mean(np.sort(logX, axis=0), axis=1)

    # compute the column-wise ranks. Each observation is replaced with its
    # rank in that column: the smallest observation is replaced by 0, the
    # second-smallest by 1, ..., and the largest by M, the number of rows.
    ranks = stats.mstats.rankdata(X, axis=0).astype(int) - 1

    # index the quantiles for each rank with the ranks matrix
    logXn = log_quantiles[ranks]

    # convert the data back to counts (casting to int is optional)
    Xn = np.round(2**logXn - 1).astype(int)
    return(Xn)
```

We'll unpack that example throughout the chapter, but for now note that it illustrates many of the things that make NumPy powerful (you will remember the first three of these moves from Chapter 1):

- Arrays can be one-dimensional, like lists, but they can also be two-dimensional, like matrices, and higher-dimensional still. This allows them to represent many different kinds of numerical data. In our case, we are representing a 2D matrix.
- Arrays allow the expression of many numerical operations at once. In the first line of the function, we take log $(x + 1)$ for every value in the array.
- Arrays can be operated on along *axes*. In the second line, we sort the data along each column just by specifying an `axis` parameter to `np.sort`. We then take the mean along each row by specifying a *different* `axis`.
- Arrays underpin the scientific Python ecosystem. The `scipy.stats.mstats.rankdata` function operates not on Python lists, but on NumPy arrays. This is true of many scientific libraries in Python.

- Arrays support many kinds of data manipulation through *fancy indexing*: `logXn = log_quantiles[ranks]`. This is possibly the trickiest part of NumPy, but also the most useful. We will explore it further in the text that follows.

In [2]:

```
%matplotlib inline

import matplotlib.pyplot as plt
plt.style.use('ggplot') # Use ggplot style graphs for something a little prettier
```

Get the data

As in Chapter 1, we will be working with the The Cancer Genome Atlas (TCGA) skin cancer RNAseq data set. Our goal is to predict mortality in skin cancer patients using their RNA expression data. By the end of this chapter we will have reproduced a simplified version of Figures 5A and 5B of a paper from the TCGA consortium.

As in Chapter 1, first we will use Pandas to make our job of reading in the data much easier. First we will read in our counts data as a pandas table.

In [3]:

```
import numpy as np
import pandas as pd

# Import TCGA melanoma data
filename = 'data/counts.txt'
with open(filename, 'rt') as f:
    data_table = pd.read_csv(f, index_col=0) # Parse file with pandas

print(data_table.iloc[:5, :5])
```

```
        00624286-41dd-476f-a63b-d2a5f484bb45  TCGA-FS-A1Z0  TCGA-D9-A3Z1  \
A1BG                                 1272.36        452.96        288.06
A1CF                                    0.00          0.00          0.00
A2BP1                                   0.00          0.00          0.00
A2LD1                                 164.38        552.43        201.83
A2ML1                                  27.00          0.00          0.00

        02c76d24-f1d2-4029-95b4-8be3bda8fdbe  TCGA-EB-A51B
A1BG                                 400.11        420.46
A1CF                                   1.00          0.00
A2BP1                                  0.00          1.00
A2LD1                                165.12         95.75
A2ML1                                  0.00          8.00
```

Looking at the first 5 rows and columns of `data_table` you we can see that the columns are the samples and the rows are the genes. Now let's put our counts in an ndarray.

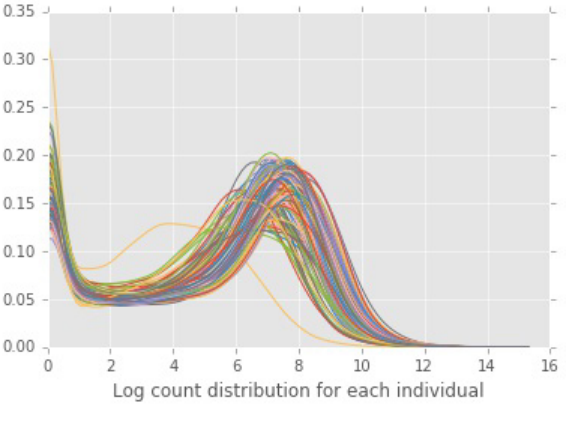
In [4]:

```
# 2D ndarray containing expression counts for each gene in each individual
counts = np.asarray(data_table, dtype=int)
```

Now, let's get a feel for our counts data by plotting the distribution of counts for each individual. We will use a gaussian kernel to smooth out bumps in our data so we can get a better idea of the overall shape.

```
In [5]:
```

```
def plot_col_density(data, xlabel=None):
    """For each column (individual) produce a density plot over all rows (genes).

    data : 2d nparray
    xlabel : x axis label
    """

    density_per_col = [stats.kde.gaussian_kde(col) for col in data.T] # Use gaussian smoothing
to estimate the density
    x = np.linspace(np.min(data), np.max(data), 100)

    plt.figure()
    for density in density_per_col:
        plt.plot(x, density(x))
    plt.xlabel(xlabel)
    plt.show()

# Before normalization
log_counts = np.log(counts + 1)
plot_col_density(log_counts, xlabel="Log count distribution for each individual")
```



We can see that while the distributions of counts are broadly similar, some individuals have flatter distributions and a few are pushed right over to the left. When doing our analysis of the counts data later in this chapter, we will be assuming that changes in gene expression are due to biological differences between our samples. But a major

distribution shift like this suggests that the differences are technical. So we will try to normalise out these global differences between individuals.

To do this, we will be performing quantile normalisation. The idea is that we assume all our samples should have a similar distribution, so any differences in the shape are due to some technical variation. We can fix this by forcing all the samples to have the same distribution. More formally, given an expression matrix (microarray data, read counts, etc) of ngenes by nsamples, quantile normalization ensures all samples have the same spread of data (by construction). It involves:

- (optionally) log-transforming the data
- sorting all the data points column-wise
- averaging the rows and
- replacing each column quantile with the quantile of the average column.

With NumPy and SciPy, this can be done easily and efficiently.

Let's assume we've read in the input matrix as X:

```
In [6]:
```

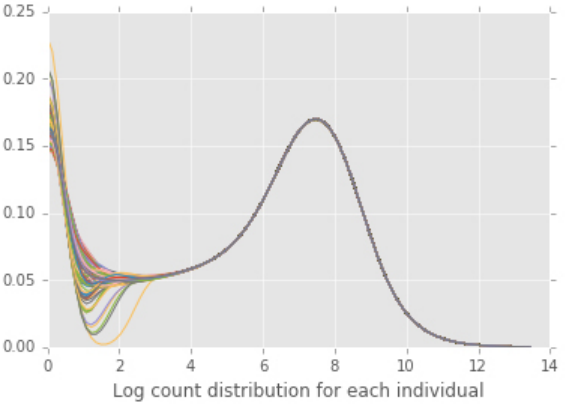
```
import numpy as np
from scipy import stats

def quantile_norm(X):
    """Normalize the columns of X to each have the same distribution.

    Given an expression matrix (microarray data, read counts, etc) of M genes
    by N samples, quantile normalization ensures all samples have the same
    spread of data (by construction).

    The input data is log-transformed, then the data across each row are
    averaged to obtain an average column. Each column quantile is replaced
    with the corresponding quantile of the average column.
    The data is then transformed back to counts.

    Parameters
    ----------
    X : 2D array of float, shape (M, N)
        The input data, with M rows (genes/features) and N columns (samples).

    Returns
    -------
    Xn : 2D array of float, shape (M, N)
        The normalized data.
    """
    # log-transform the data
    logX = np.log2(X + 1)

    # compute the quantiles
```

```
    log_quantiles = np.mean(np.sort(logX, axis=0), axis=1)

    # compute the column-wise ranks
    ranks = stats.mstats.rankdata(X, axis=0).astype(int) - 1
    # alternative: ranks = np.argsort(np.argsort(X, axis=0), axis=0)

    # index the quantiles for each rank with the ranks matrix
    logXn = log_quantiles[ranks]

    # convert the data back to counts (casting to int is optional)
    Xn = np.round(2**logXn - 1).astype(int)
    return(Xn)
```

Now, let's see what our distributions look like after quantile normalization.

```
In [7]:
```

```
# After normalization
log_quant_norm_counts = np.log(quantile_norm(counts)+1)

plot_col_density(log_quant_norm_counts, xlabel="Log count distribution for each individual")
```



As you might expect, the distributions now look virtually identical!

Now that we have normalized our counts, we can start using our gene expression data to predict mortality

Biclustering the counts data

Now that the data are normalized, we can cluster the genes (rows) and samples (columns) of the expression matrix. Clustering the rows tells us which genes' expression values are linked, which is an indication that they work together in the process being studied. Clustering the samples tells us which samples have similar gene expression profiles, which may indicate similar characteristics of the samples on other scales.

Because clustering can be an expensive operation, we will limit our analysis to the 1,500 genes that are most variable, since these will account for most of the correlation signal in either dimension.

```
In [8]:
```

```
def most_variable_rows(data, n=1500):
    """Subset data to the n most variable rows

    In this case, we want the n most variable genes.

    Parameters
    ----------
    data : 2D array of float
        The data to be subset
    n : int, optional
        Number of rows to return.
    method : function, optional
        The function with which to compute variance. Must take an array
        of shape (nrows, ncols) and an axis parameter and return an
        array of shape (nrows,).
    """
    # compute variance along the columns axis
    rowvar = np.var(data, axis=1)
    # Get sorted indices (ascending order), take the last n
    sort_indices = np.argsort(rowvar)[-n:]
    # use as index for data
    variable_data = data[sort_indices, :]
    return variable_data
```

Next, we need a function to *bicluster* the data. This means clustering along both the rows (to find out with genes are working together) and the columns (to find out which samples are similar).

Normally, you would use a sophisticated clustering algorithm from the scikit-learn library for this. In our case, we want to use hierarchical clustering for simplicity and ease of display. The SciPy library happens to have a perfectly good hierarchical clustering module, though it requires a bit of wrangling to get your head around its interface.

As a reminder, hierarchical clustering is a method to group observations using sequential merging of clusters: initially, every observation is its own cluster. Then, the two nearest clusters are repeatedly merged, and then the next two, and so on, until every observation is in a single cluster. This sequence of merges forms a *merge tree*. By cutting the tree at a specific height, we can get a finer or coarser clustering of observations.

The `linkage` function in `scipy.cluster.hierarchy` performs a hierarchical clustering of the rows of a matrix, using a particular metric (for example, Euclidean distance, Manhattan distance, or others) and a particular linkage method, the distance

between two clusters (for example, the average distance between all the observations in a pair of clusters).

It returns the merge tree as a "linkage matrix", which contains each merge operation along with the distance computed for the merge and the number of observations in the resulting cluster. From the `linkage` documentation:

> *A cluster with an index less than n corresponds to one of the n original observations. The distance between clusters `Z[i, 0]` and `Z[i, 1]` is given by `Z[i, 2]`. The fourth value `Z[i, 3]` represents the number of original observations in the newly formed cluster.*

Whew! So that's a lot of information, but let's dive right in and hopefully you'll get the hang of it rather quickly. First, we define a function, `bicluster`, that clusters both the rows *and* the columns of a matrix:

```
In [9]:
```

```
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage

def bicluster(data, linkage_method='average', distance_metric='correlation'):
    """Cluster the rows and the columns of a matrix.

    Parameters
    ----------
    data : 2D ndarray
        The input data to bicluster.
    linkage_method : string, optional
        Method to be passed to `linkage`.
    distance_metric : string, optional
        Distance metric to use for clustering. See the documentation
        for ``scipy.spatial.distance.pdist`` for valid metrics.

    Returns
    -------
    y_rows : linkage matrix
        The clustering of the rows of the input data.
    y_cols : linkage matrix
        The clustering of the cols of the input data.
    """
    y_rows = linkage(data, method=linkage_method, metric=distance_metric)
    y_cols = linkage(data.T, method=linkage_method, metric=distance_metric)
    return y_rows, y_cols
```

Simple: we just call `linkage` for the input matrix and also for the transpose of that matrix, in which columns become rows and rows become columns.

Visualizing clusters

Next, we define a function to visualize the output of that clustering. We are going to rearrange the rows and columns of the input data so that similar rows are together

and similar columns are together. And we are additionally going to show the merge tree for both rows and columns, displaying which observations belong together for each. The merge trees are presented as dendograms, with the branch-lengths indicating how similar the obvservations are to each other (shorter = more similar).

As a word of warning, there is a fair bit of hard-coding of parameters going on here. This is difficult to avoid for plotting, where design is often a matter of eyeballing to find the correct proportions.

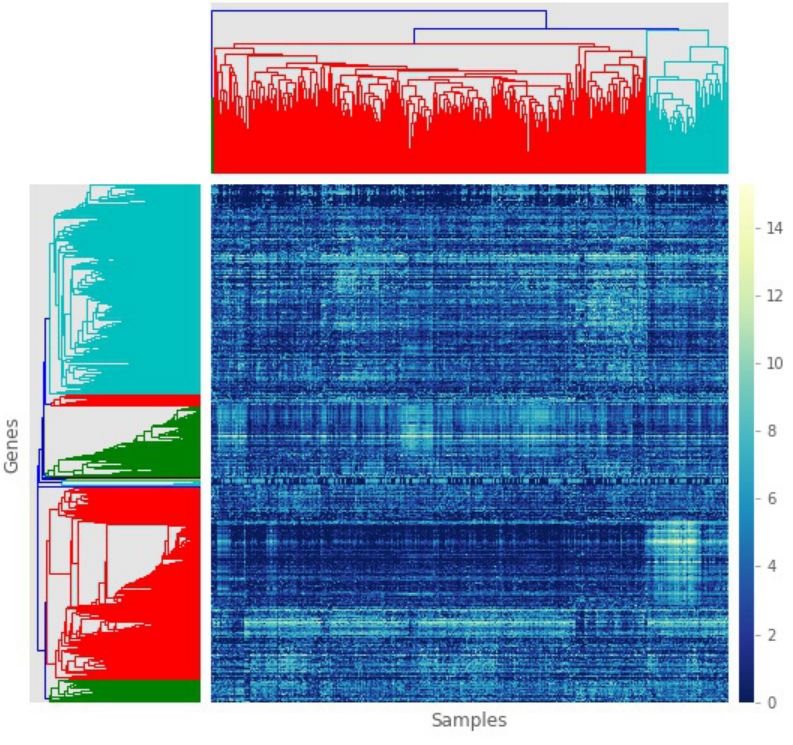
```
In [10]:
```

```
from scipy.cluster.hierarchy import dendrogram, leaves_list
def plot_bicluster(data, row_linkage, col_linkage,
                   row_nclusters=10, col_nclusters=3):
    """Perform a biclustering, plot a heatmap with dendrograms on each axis.

    Parameters
    ----------
    data : array of float, shape (M, N)
        The input data to bicluster.
    row_linkage : array, shape (M-1, 4)
        The linkage matrix for the rows of `data`.
    col_linkage : array, shape (N-1, 4)
        The linkage matrix for the columns of `data`.
    n_clusters_r, n_clusters_c : int, optional
        Number of clusters for rows and columns.
    """
    fig = plt.figure(figsize=(8, 8))

    # Compute and plot row-wise dendrogram
    # `add_axes` takes a "rectangle" input to add a subplot to a figure.
    # The figure is considered to have side-length 1 on each side, and its
    # bottom-left corner is at (0, 0).
    # The measurements passed to `add_axes` are the left, bottom, width, and
    # height of the subplot. Thus, to draw the left dendrogram (for the rows),
    # we create a rectangle whose bottom-left corner is at (0.09, 0.1), and
    # measuring 0.2 in width and 0.6 in height.
    ax1 = fig.add_axes([0.09, 0.1, 0.2, 0.6])
    # For a given number of clusters, we can obtain a cut of the linkage
    # tree by looking at the corresponding distance annotation in the linkage
    # matrix.
    threshold_r = (row_linkage[-row_nclusters, 2] +
                   row_linkage[-row_nclusters+1, 2]) / 2
    dendrogram(row_linkage, orientation='left', color_threshold=threshold_r)

    # Compute and plot column-wise dendogram
    # See notes above for explanation of parameters to `add_axes`
    ax2 = fig.add_axes([0.3, 0.71, 0.6, 0.2])
    threshold_c = (col_linkage[-col_nclusters, 2] +
                   col_linkage[-col_nclusters+1, 2]) / 2
    dendrogram(col_linkage, color_threshold=threshold_c)
```

```
    # Hide axes labels
    ax1.set_xticks([])
    ax1.set_yticks([])
    ax2.set_xticks([])
    ax2.set_yticks([])

    # Plot data heatmap
    ax = fig.add_axes([0.3, 0.1, 0.6, 0.6])

    # Sort data by the dendogram leaves
    idx_rows = leaves_list(row_linkage)
    data = data[idx_rows, :]
    idx_cols = leaves_list(col_linkage)
    data = data[:, idx_cols]

    im = ax.matshow(data, aspect='auto', origin='lower', cmap='YlGnBu_r')
    ax.set_xticks([])
    ax.set_yticks([])

    # Axis labels
    plt.xlabel('Samples')
    plt.ylabel('Genes', labelpad=125)

    # Plot legend
    axcolor = fig.add_axes([0.91, 0.1, 0.02, 0.6])
    plt.colorbar(im, cax=axcolor)

    # display the plot
    plt.show()
```

Now we apply these functions to our normalized counts matrix to display row and column clusterings.

```
In [11]:

counts_log = np.log(counts + 1)
counts_var = most_variable_rows(counts_log, n=1500)
yr, yc = bicluster(counts_var)
plot_bicluster(counts_var, yr, yc)
```



Predicting survival

We can see that the sample data naturally falls into at least 2 clusters. Are these clusters meaningful? To answer this, we can access the patient data, available from the data repository for the paper. After some preprocessing, we get the patients table (TODO: LINK TO FINAL PATIENTS TABLE), which contains survival information for each patient. We can then match these to the counts clusters, and understand whether the patients' gene expression can predict differences in their pathology.

```
In [12]:
```

```
patients = pd.read_csv('data/patients.csv', index_col=0)
patients.head()
```

```
Out[12]:
```

	UV-signature	original-clusters	melanoma-survival-time	melanoma-dead
TCGA-BF-A1PU	UV signature	keratin	NaN	NaN
TCGA-BF-A1PV	UV signature	keratin	13.0	0.0
TCGA-BF-A1PX	UV signature	keratin	NaN	NaN
TCGA-BF-A1PZ	UV signature	keratin	NaN	NaN

	UV-signature	original-clusters	melanoma-survival-time	melanoma-dead
TCGA-BF-A1Q0	not UV	immune	17.0	0.0

For each patient (the rows) we have:

- UV signature: Ultraviolet light tends to cause specific DNA mutations. By looking for this mutation signature they can infer that UV light likely caused the mutation(s) that lead to cancer in these patients.
- original clusters: In the paper, the patients were clustered using gene expression data. These clusters were classified according to the types of genes that typified that cluster. The main clusters were "immune" (n = 168; 51%), "keratin" (n = 102; 31%), and "MITFlow" (n = 59; 18%).
- melanoma survival time: Number of days that the patient survived.
- melanoma dead: 1 if the patient died of melanoma, 0 if they are alive or died of something else.

Now we need to draw *survival curves* for each group of patients defined by the clustering. This is a plot of the fraction of a population that remains alive over a period of time. Note that some data is *right-censored*, which means that in some cases, we don't actually know when the patient died, or the patient might have died of causes unrelated to the melanoma. We count these patients as "alive" for the duration of the survival curve, but more sophisticated analyses might try to estimate their likely time of death.

To obtain a survival curve from survival times, we create a step function that decreases by $1/n$ at each step, where n is the number of patients in the group. We then match that function against the non-censored survival times.

```
In [13]:
```

```
def survival_distribution_function(lifetimes, right_censored=None):
    """Return the survival distribution function of a set of lifetimes.

    Parameters
    ----------
    lifetimes : array of float or int
        The observed lifetimes of a population. These must be non-negative.
    right_censored : array of bool, same shape as `lifetimes`
        A value of `True` here indicates that this lifetime was not
        observed. Values of `np.nan` in `lifetimes` are also considered
        to be right-censored.

    Returns
    -------
    sorted_lifetimes : array of float
        The
    sdf : array of float
```

```
        Values starting at 1 and progressively decreasing, one level
        for each observation in `lifetimes`.

    Examples
    --------

    In this example, of a population of four, two die at time 1, a
    third dies at time 2, and a final individual dies at an unknown
    time. (Hence, ``np.nan``.)

    >>> lifetimes = np.array([2, 1, 1, np.nan])
    >>> survival_distribution_function(lifetimes)
    (array([ 0., 1., 1., 2.]), array([ 1. , 0.75, 0.5 , 0.25]))
    """
    n_obs = len(lifetimes)
    rc = np.isnan(lifetimes)
    if right_censored is not None:
        rc |= right_censored
    observed = lifetimes[~rc]
    xs = np.concatenate(([0], np.sort(observed)))
    ys = np.concatenate((np.arange(1, 0, -1/n_obs), [0]))
    ys = ys[:len(xs)]
    return xs, ys
```

Now that we can easily obtain survival curves from the survival data, we can plot them. We write a function that groups the survival times by cluster identity and plots each group as a different line:

```
In [14]:
```

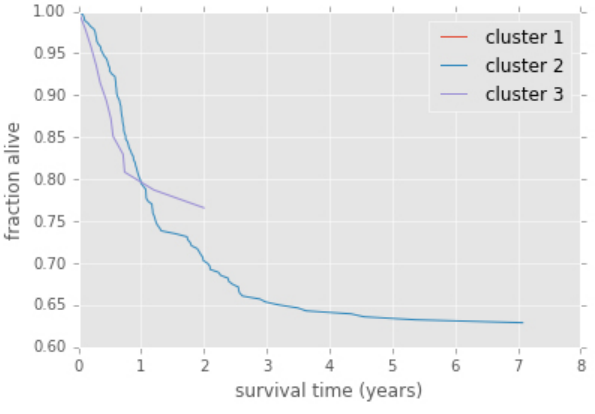
```
def plot_cluster_survival_curves(clusters, sample_names, patients,
                                 censor=True):
    """Plot the survival data from a set of sample clusters.

    Parameters
    ----------
    clusters : array of int or categorical pd.Series
        The cluster identity of each sample, encoded as a simple int
        or as a pandas categorical variable.
    sample_names : list of string
        The name corresponding to each sample. Must be the same length
        as `clusters`.
    patients : pandas.DataFrame
        The DataFrame containing survival information for each patient.
        The indices of this DataFrame must correspond to the
        `sample_names`. Samples not represented in this list will be
        ignored.
    censor : bool, optional
        If `True`, use `patients['melanoma-dead']` to right-censor the
        survival data.
    """
    plt.figure()
    if type(clusters) == np.ndarray:
```

```
        cluster_ids = np.unique(clusters)
        cluster_names = ['cluster {}'.format(i) for i in cluster_ids]
    elif type(clusters) == pd.Series:
        cluster_ids = clusters.cat.categories
        cluster_names = list(cluster_ids)
    n_clusters = len(cluster_ids)
    for c in cluster_ids:
        clust_samples = np.flatnonzero(clusters == c)
        # discard patients not present in survival data
        clust_samples = [sample_names[i] for i in clust_samples
                         if sample_names[i] in patients.index]
        patient_cluster = patients.loc[clust_samples]
        survival_times = np.array(patient_cluster['melanoma-survival-time'])
        if censor:
            censored = ~np.array(patient_cluster['melanoma-dead']).astype(bool)
        else:
            censored = None
        stimes, sfracs = survival_distribution_function(survival_times,
                                                        censored)
        plt.plot(stimes / 365, sfracs)

    plt.xlabel('survival time (years)')
    plt.ylabel('fraction alive')
    plt.legend(cluster_names)
```

Now we can use the `fcluster` function to obtain cluster identities for the samples (columns of the counts data), and plot each survival curve separately. The `fcluster` function takes a linkage matrix, as returned by `linkage`, and a threshold, and returns cluster identities. It's difficult to know a-priori what the threshold should be, but we can obtain the appropriate threshold for a fixed number of clusters by checking the distances in the linkage matrix.

```
In [15]:
```

```
from scipy.cluster.hierarchy import fcluster
n_clusters = 3
threshold_distance = (yc[-n_clusters, 2] + yc[-n_clusters+1, 2]) / 2
clusters = fcluster(yc, threshold_distance, 'distance')

plot_cluster_survival_curves(clusters, data_table.columns, patients)
```



The clustering of gene expression profiles has identified a higher-risk subtype of melanoma, which constitutes the majority of patients. This is indeed only the latest study to show such a result, with others identifying subtypes of leukemia (blood cancer), gut cancer, and more. Although the above clustering technique is quite fragile, there are other ways to explore this dataset and similar ones that are more robust.

Exercise: Do our clusters do a better job of predicting survival than the original clusters in the paper? What about UV signature? Plot survival curves using the original clusters and UV signature columns of the patient data. How do they compare to our clusters?

Exercise: We leave you the exercise of implementing the approach described in the paper:

1. Take bootstrap samples (random choice with replacement) of the genes used to cluster the samples;
2. For each sample, produce a hierarchical clustering;
3. In a `(n_samples, n_samples)`-shaped matrix, store the number of times a sample pair appears together in a bootstrapped clustering.
4. Perform a hierarchical clustering on the resulting matrix.

This identifies groups of samples that frequently occur together in clusterings, regardless of the genes chosen. Thus, these samples can be considered to robustly cluster together.

Hint: use `np.random.choice` with `replacement=True` to create bootstrap samples of row indices.

CHAPTER 3

Networks of Image Regions with ndimage

This chapter gets a special mention because it inspired the whole book. Vighnesh Birodkar wrote this as an undergraduate while participating in Google Summer of Code (GSoC) 2014. When I saw this bit of code, it blew me away, and over a year later, I still haven't seen anything like it. For the purposes of this book, it touches on many aspects of scientific Python. By the time you're done with this chapter, you should be able to process arrays of *any* dimension, rather than thinking of them only as 1D lists or 2D tables. More than that, you'll understand the basics of image filtering and network processing.

You probably know that digital images are made up of *pixels*. These are the light signal *sampled on a regular grid*. When computing on images, we often deal with objects much larger than individual pixels. In a landscape, the sky, earth, trees, rocks each span many pixels. A common structure to represent these is the Region Adjacency Graph, or RAG. Its *nodes* hold properties of each region in the image, and its *links* hold the spatial relationships between the regions. Two nodes are linked whenever their corresponding regions touch each other in the input image.

Building such a structure could be a complicated affair, and even more difficult when images are not two-dimensional but 3D and even 4D, as is common in microscopy, materials science, and climatology, among others. But here we will show you how to produce a RAG in a few lines of code using NetworkX (a Python library to analyze graphs and networks), and a filter from SciPy's N-dimensional image processing submodule, `ndimage`.

```
In [1]:
```

```
import networkx as nx
import numpy as np
from scipy import ndimage as nd
```

```
def add_edge_filter(values, graph):
    center = values[len(values) // 2]
    for neighbor in values:
        if neighbor != center and not graph.has_edge(center, neighbor):
            graph.add_edge(center, neighbor)
    return 0.0

def build_rag(labels, image):
    g = nx.Graph()
    footprint = ndi.generate_binary_structure(labels.ndim, connectivity=1)
    _ = ndi.generic_filter(labels, add_edge_filter, footprint=footprint,
                           mode='nearest', extra_arguments=(g,))
    return g
```

There are a few things going on here: images being represented as numpy arrays, *filtering* of these images using `scipy.ndimage`, and building of the image regions into a graph (network) using the NetworkX library. We'll go over these in turn.

Images are numpy arrays

In the previous chapter, we saw that numpy arrays can efficiently represent tabular data, and are a convenient way to perform computations on it. It turns out that arrays are equally adept at representing images.

Here's how to create an image of white noise using just numpy, and display it with matplotlib. First, we import the necessary packages, and use the `matplotlib inline` IPython magic to make our images appear below the code:

```
In [2]:
```

```
%matplotlib inline
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import cm # colormap module
```

Next, we set the default matplotlib colormap and interpolation method:

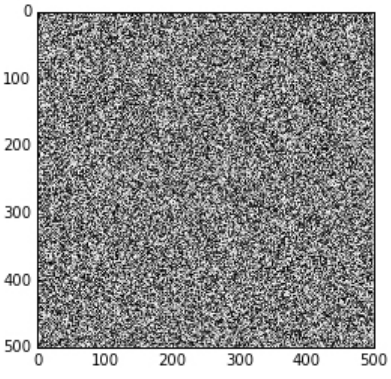
```
In [3]:
```

```
mpl.rcParams['image.cmap'] = 'gray'
mpl.rcParams['image.interpolation'] = None
```

Finally, "make some noise" and display it as an image:

```
In [4]:
```

```
random_image = np.random.rand(500, 500)
plt.imshow(random_image);
```



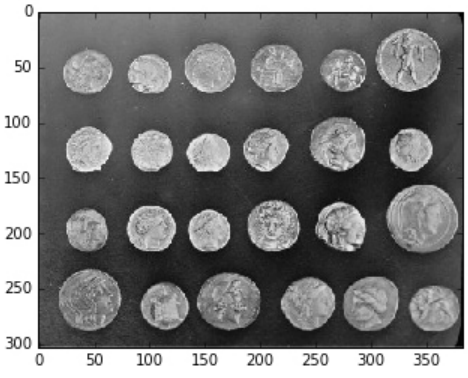
This displays a numpy array as an image. The converse is also true: an image can be considered "as" a numpy array. For this example we use the scikit-image library, a collection of image processing tools built on top of NumPy and SciPy.

Here is PNG image from the scikit-image repository. It is a black and white (sometimes called "grayscale") picture of some ancient Roman coins from Pompeii, obtained from the Brooklyn Museum [^coins-source]:

Here is the coin image loaded with scikit-image:

```
In [5]:

from skimage import io
url_coins = 'https://raw.githubusercontent.com/scikit-image/scikit-image/v0.10.1/skimage/data/coin
coins = io.imread(url_coins)
print("Type:", type(coins), "Shape:", coins.shape, "Data type:", coins.dtype)
plt.imshow(coins);
```

```
Type: <class 'numpy.ndarray'> Shape: (303, 384) Data type: uint8
```



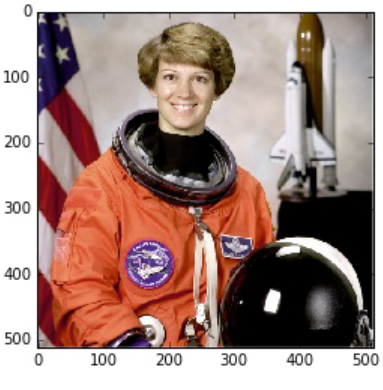
A grayscale image can be represented as a *2-dimensional array*, with each array element containing the grayscale intensity at that position. So, **an image is just a numpy array**.

Color images are a *3-dimensional* array, where the first two dimensions represent the spatial positions of the image, while the final dimension represents color channels, typically the three primary additive colors of red, green, and blue. To show what we can do with these dimensions, let's play with this photo of an astronaut:

```
In [6]:
```

```
url_astronaut = 'https://raw.githubusercontent.com/scikit-image/scikit-image/master/skimage/data/a
astro = io.imread(url_astronaut)
print("Type:", type(astro), "Shape:", astro.shape, "Data type:", astro.dtype)
plt.imshow(astro);
```

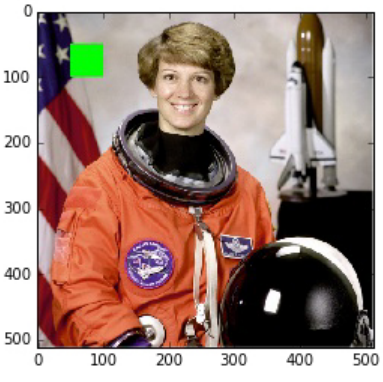
```
Type: <class 'numpy.ndarray'> Shape: (512, 512, 3) Data type: uint8
```



This image is *just numpy arrays*. Adding a green square to the image is easy once you realize this, using simple numpy slicing:

```
In [7]:
```

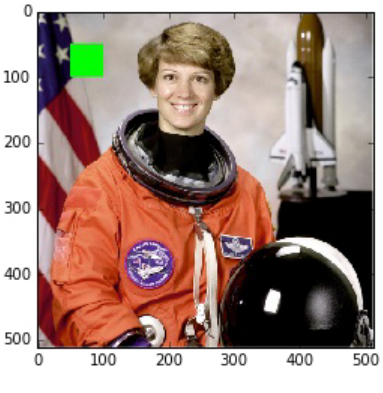
```
astro_sq = np.copy(astro)
astro_sq[50:100, 50:100] = [0, 255, 0] # red, green, blue
plt.imshow(astro_sq);
```



You can also use a boolean *mask*, an array of `True` or `False` values. We saw these in Chapter 2 as a way to select rows of a table. In this case, we can use an array of the same shape as the image to select pixels:

```
In [8]:
```

```
astro_sq = np.copy(astro)
sq_mask = np.zeros(astro.shape[:2], bool)
sq_mask[50:100, 50:100] = True
astro_sq[sq_mask] = [0, 255, 0]
plt.imshow(astro_sq);
```



Exercise: We just saw how to select a square and paint it red. Can you extend that to other shapes and colors? Create a function to draw a blue grid onto a color image, and apply it to the `astronaut` image of Eileen Collins (above). Your function should take two parameters: the input image, and the grid spacing. Use the following template to help you get started.

```
In [9]:
```

```
def overlay_grid(image, spacing=128):
    """Return an image with a grid overlay, using the provided spacing.

    Parameters
    ----------
    image : array, shape (M, N, 3)
        The input image.
    spacing : int
        The spacing between the grid lines.

    Returns
    -------
    image_gridded : array, shape (M, N, 3)
        The original image with a blue grid superimposed.
    """
    image_gridded = image.copy()
    pass # replace this line with your code...
    return image_gridded

# plt.imshow(overlay_grid(astro, 128)); # ... and uncomment this line to test your function
```

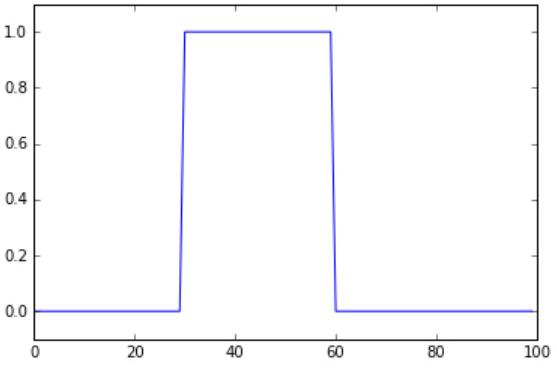
Filters in signal processing

Filtering is one of the most fundamental and common operations in image processing. You can filter an image to remove noise, to enhance features, or to detect edges between objects in the image.

To understand filters, it's easiest to start with a 1D signal, instead of an image. For example, you might measure the light arriving at your end of a fiber-optic cable. If you *sample* the signal every millisecond (ms) for 100ms, you end up with an array of length 100. Suppose that after 30ms the light signal is turned on, and 30ms later, it is switched off. You end up with a signal like this:

```
In [10]:
```

```
sig = np.zeros(100, np.float) #
sig[30:60] = 1  # signal = 1 during the period 30-60ms because light is observed
plt.plot(sig);
plt.ylim(-0.1, 1.1);
```



To find *when* the light is turned on, you can *delay* it by 1ms, then *subtract* the original from delayed signal. This way, when the signal is unchanged from one millisecond to the next, the subtraction will give zero, but when the signal *increases*, you will get a positive signal.

When the signal *decreases*, we will get a negative signal. If we are only interested in pinpointing the time when the light was turned on, we can *clip* the difference signal, so that any negative values are converted to 0.

```
In [11]:

sigdelta = sig[1:]  # sigdelta[0] equals sig[1], and so on
sigdiff = sigdelta - sig[:-1]
sigon = np.clip(sigdiff, 0, np.inf)
print(1 + np.flatnonzero(sigon)[0], 'ms')

30 ms
```

It turns out that this can be accomplished by an signal processing operation called *convolution*. At every point of the signal, we compute the dot-product between the values surrounding it and a *kernel* or *filter*, which is a predetermined vector of values. Depending on the kernel, then, the convolution shows a different feature of the signal.

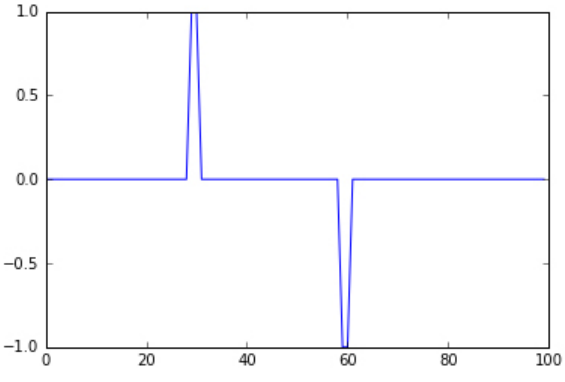
Now, think of what happens when the kernel is (1, 0, -1), the difference filter, for a signals. At any position `i`, the convolution result is `1*s[i+1] + 0*s[i] - 1*s[i-1]`, that is, `s[i+1] - s[i-1]`. Thus, when adjacent values are identical, the convolution gives 0, but when `s[i+1] > s[i-1]` (the signal is increasing), it gives a positive value, and, conversely, when `s[i+1] < s[i-1]`, it gives a negative value. You can think of this as an estimate of the derivative of the input function.

In general, the formula for convolution is: $s'(t) = \Sigma_{j=t-\iota}^{t} s(j) f(t-j)$ where s is the signal, s' is the filtered signal, f is the filter, and τ is the length of the filter.

In scipy, you can use the `scipy.ndimage.convolve` to work on this.

```
In [12]:
```

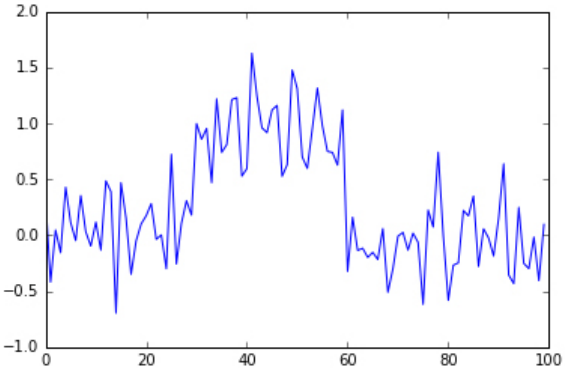
```
diff = np.array([1, 0, -1])
from scipy import ndimage as ndi
dsig = ndi.convolve(sig, diff)
plt.plot(dsig);
```



Signals are usually *noisy* though, not perfect as above:

```
In [13]:
```

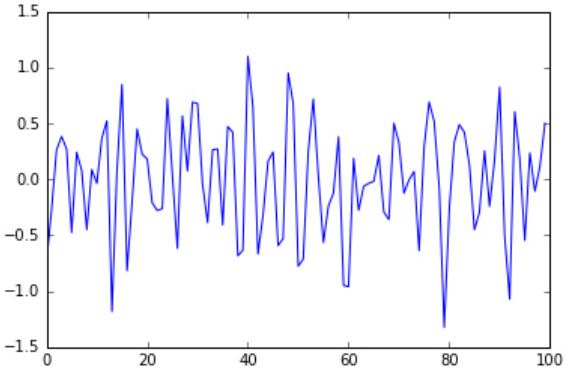
```
sig = sig + np.random.normal(0, 0.3, size=sig.shape)
plt.plot(sig);
```



The plain difference filter can amplify that noise:

```
In [14]:
```

```
plt.plot(ndi.convolve(sig, diff));
```

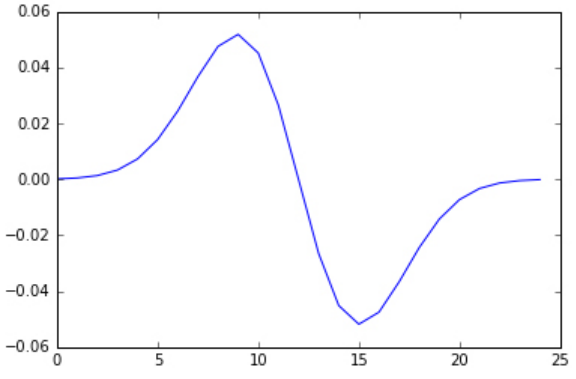


In such cases, you can add smoothing to the filter. The most common form of smoothing is *Gaussian* smoothing, which takes the weighted average of neighboring points in the signal using the Gaussian function. We can write a function to make a Gaussian smoothing kernel as follows:

```
In [15]:

def gaussian_kernel(size, sigma):
    """Make a 1D Gaussian kernel of the specified size and standard deviation.

    The size should be an odd number and at least ~6 times greater than sigma
    to ensure sufficient coverage.
    """
    positions = np.arange(size) - size // 2
    kernel_raw = np.exp(-positions**2 / (2 * sigma**2))
    kernel_normalized = kernel_raw / np.sum(kernel_raw)
    return kernel_normalized
```

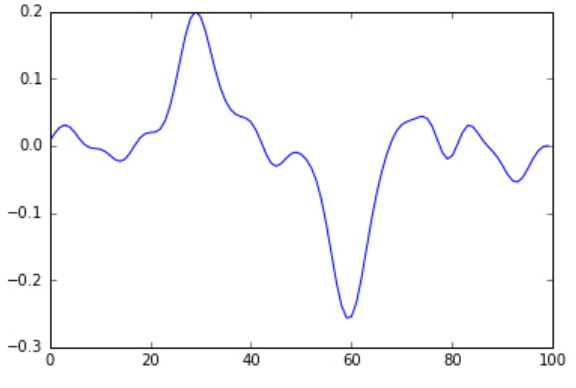
A really nice feature feature of convolution is that it's *associative*, meaning if you want to find the derivative of the smoothed signal, you can equivalently convolve the signal with the smoothed difference filter! This can save a lot of computation time, because you can smooth just the filter, which is usually much smaller than the data.

```
In [16]:

smooth_diff = ndi.convolve(gaussian_kernel(25, 3), diff)
plt.plot(smooth_diff)

Out[16]:

[<matplotlib.lines.Line2D at 0x112bc04a8>]
```



This smoothed difference filter looks for an edge in the central position, but also for that difference to continue. This continuation happens in the case of a true edge, but not in "spurious" edges caused by noise. Check out the result:

```
In [17]:

sdsig = ndi.convolve(sig, smooth_diff)
plt.plot(sdsig);
```



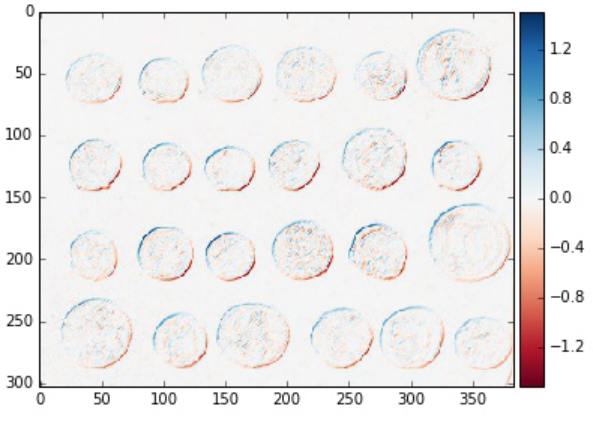
Although it still looks wobbly, the *signal-to-noise ratio* (SNR), is much greater in this version than when using the simple difference filter.

(Note: this operation is called filtering because, in physical electrical circuits, many of these operations are implemented by hardware that lets certain kinds of current through, but not others; these components are called filters. For example, a common filter that removes high-frequency voltage fluctuations from a current is called a *low-pass filter*.)

Filtering images (2D filters)

Now that you've seen filtering in 1D, I hope you'll find it straightforward to extend these concepts to 2D. Here's a 2D difference filter finding the edges in the coins image:

```
In [18]:

coins = coins.astype(float) / 255 # prevents overflow errors
diff2d = np.array([[0, 1, 0], [1, 0, -1], [0, -1, 0]])
coins_edges = ndi.convolve(coins, diff2d)
io.imshow(coins_edges);
```



The principle is the same as the 1D filter: at every point in the image, place the filter, compute the dot-product of the filter's values with the image values, and place the result at the same location in the output image. And, as with the 1D difference filter, when the filter is placed on a location with little variation, the dot-product cancels out to zero, whereas, placed on a location where the image brightness is changing, the values multiplied by 1 will be different from those multiplied by -1, and the filter's output will be a positive or negative value (depending on whether the image is brighter towards the bottom-right or top-left at that point).

Just as with the 1D filter, you can get more sophisticated and smooth out noise right within the filter. The *Sobel* filter is designed to do just that. It comes in horizontal and vertical varieties, to find edges with that orientation in the data. Let's start with the horizontal filter first. To find a horizontal edge in a picture, you might try the following filter:

```
In [19]:

# column vector (vertical) to find horizontal edges
hdiff = np.array([[1], [0], [-1]])
```

However, as we saw with 1D filters, this will result in a noisy estimate of the edges in the image. But rather than using Gaussian smoothing, which can cause blurry edges, the Sobel filter uses the property that edges in images tend to be continuous: a picture of the ocean, for example, will contain a horizontal edge along an entire line, not just at specific points of the image. So the Sobel filter smooths the vertical filter horizontally: it looks for a strong edge at the central position that is corroborated by the adjacent positions:

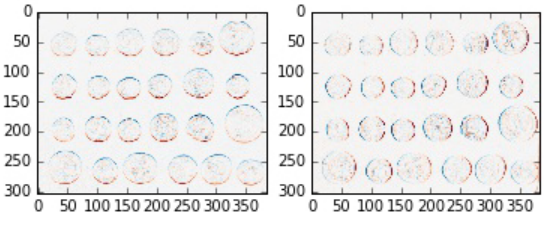
```
In [20]:

hsobel = np.array([[ 1,  2,  1],
                   [ 0,  0,  0],
                   [-1, -2, -1]])
```

The vertical Sobel filter is simply the transpose of the horizontal:

```
In [21]:

vsobel = hsobel.T
```

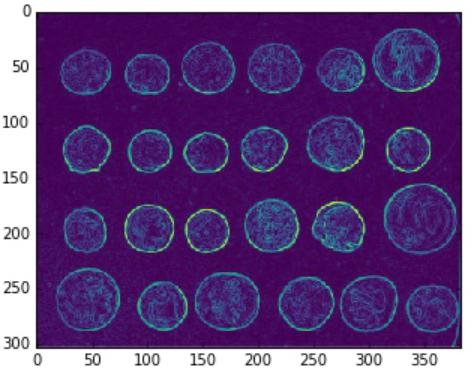
We can then find the horizontal and vertical edges in the coins image:

```
In [22]:

coins_h = ndi.convolve(coins, hsobel)
coins_v = ndi.convolve(coins, vsobel)

fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(coins_h, cmap=plt.cm.RdBu)
axes[1].imshow(coins_v, cmap=plt.cm.RdBu)

Out[22]:

<matplotlib.image.AxesImage at 0x10f624780>
```



And finally, just like the Pythagorean theorem, you can argue that the edge magnitude in *any* direction is equal to the square root of the sum of squares of the horizontal and vertical components:

```
In [23]:

coins_sobel = np.sqrt(coins_h**2 + coins_v**2)
plt.imshow(coins_sobel, cmap=plt.cm.viridis);
```



Generic filters

In addition to dot-products, implemented by `ndi.convolve`, SciPy lets you define a filter that is an *arbitrary function* of the points in a neighborhood, implemented in `ndi.generic_filter`. This can let you express arbitrarily complex filters.

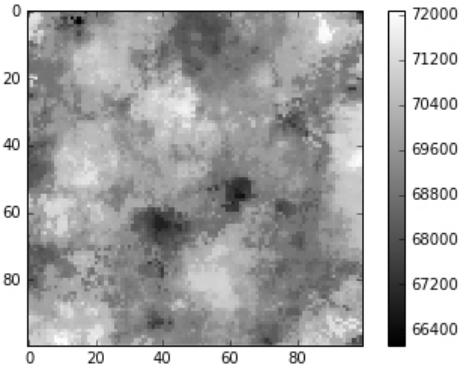
For example, suppose an image represents median house values in a county, with a 100m x 100m resolution. The local council decides to tax house sales as $10,000 plus 5% of the 90th percentile of house prices in a 1km radius. (So, selling a house in an expensive neighborhood costs more.) With `generic_filter`, we can produce the map of the tax rate everywhere in the map:

```
In [24]:
```

```
from skimage import morphology
def tax(prices):
    return 10 + 0.05 * np.percentile(prices, 90)
house_price_map = (0.5 + np.random.rand(100, 100)) * 1e6
footprint = morphology.disk(radius=10)
tax_rate_map = ndi.generic_filter(house_price_map, tax, footprint=footprint)
plt.imshow(tax_rate_map)
plt.colorbar()
```

```
Out[24]:
```

```
<matplotlib.colorbar.Colorbar at 0x114aafd30>
```



Exercise: Conway's Game of Life.

Suggested by Nicolas Rougier.

Conway's Game of Life is a seemingly simple construct in which "cells" on a regular square grid live or die according to the cells in their immediate surroundings. At every timestep, we determine the state of position (i, j) according to its previous state and that of its 8 neighbors (above, below, left, right, and diagonals):

- a live cell with only one live neighbor or none dies.
- a live cell with two or three live neighbors lives on for another generation.
- a live cell with four or more live neighbors dies, as if from overpopulation.
- a dead cell with exactly three live neighbors becomes alive, as if by reproduction.

Although the rules sound like a contrived math problem, they in fact give rise to incredible patterns, starting with gliders (small patterns of live cells that slowly move in each generation) and glider guns (stationary patterns that sprout off gliders), all the way up to prime number generator machines (see, for example, this page), and even simulating Game of Life itself!

Can you implement the Game of Life using `ndi.generic_filter`?

Exercise: Sobel gradient magnitude.

Above, we saw how we can combine the output of two different filters, the horizontal Sobel filter, and the vertical one. Can you write a function that does this in a single pass using `ndi.generic_filter`?

Graphs and the NetworkX library

To introduce you to graphs, we will reproduce some results from the paper "Structural properties of the *Caenorhabditis elegans* neuronal networks", by Varshney *et al*, 2011. Note that in this context the term "graph" is synonymous with "network", but not with "plot". Mathematicians and computer scientists invented slightly different words to discuss these: graph = network, vertex = node, and edge = link. As most people do, we will be using these terms interchangeably.

You might be slightly more familiar with the network terminology: a network consists of *nodes* and *links* between the nodes. Equivalently, a graph consists of *vertices* and *edges* between the vertices. In NetworkX, you have `Graph` objects consisting of `nodes` and `edges` between the nodes. Oh well.

Graphs are a natural representation for a bewildering variety of data. Pages on the world wide web, for example, can comprise nodes, while links between those pages can be, well, links. Or, in so-called *transcriptional networks*, nodes represent genes and edges connect genes that have a direct influence on each other's expression.

In our example, we will represent neurons in the nematode worm's nervous system as nodes, and place an edge between two nodes when a neuron makes a synapse with another. (*Synapses* are the chemical connections through which neurons communicate.) The worm is an awesome example of neural connectivity analysis because every worm (of this species) has the same number of neurons (302), and the connections between them are all known. This has resulted in the fantastic Openworm project [^openworm], which I encourage you to follow.

You can download the neuronal dataset in Excel format (yuck) from the WormAtlas database at http://www.wormatlas.org/neuronalwiring.html#Connectivitydata. The direct link to the data is: http://www.wormatlas.org/images/NeuronConnect.xls Let's start by getting a list of rows out of the file. An elegant pattern from Tony Yu [^fileurl] enables us to open a remote URL as a local file. It uses a context manager to download a remote file to a local temporary file. (Your operating system provides Python with a place to put temporary files.)

The funny `@something` syntax might be new to you. This is a Python decorator, a function that modifies another function. We won't go over decorators just yet, as they are a side point here. In Chapter 8, we will discuss a particular decorator in more detail.

We then use the `xlrd` library to read the contents of the Excel file into a connectivity matrix.

```
In [25]:
```

```
import os
import xlrd # Excel-reading library in Python
```

```
from urllib.request import urlopen  # getting files from the web, Py3

import tempfile
from contextlib import contextmanager

@contextmanager
def url2filename(url):
    base_filename, ext = os.path.splitext(url)
    with tempfile.NamedTemporaryFile(delete=False, suffix=ext) as f:
        remote = urlopen(url)
        f.write(remote.read())
    try:
        yield f.name
    finally:
        os.remove(f.name)

connectome_url = "http://www.wormatlas.org/images/NeuronConnect.xls"

with url2filename(connectome_url) as fin:
    sheet = xlrd.open_workbook(fin).sheet_by_index(0)
    conn = [sheet.row_values(i) for i in range(1, sheet.nrows)]
```

`conn` now contains a list of connections of the form:

[Neuron1, Neuron2, connection type, strength]

We are only going to examine the connectome of chemical synapses, so we filter out other synapse types as follows:

```
In [26]:

conn_edges = [(n1, n2, {'weight': s}) for n1, n2, t, s in conn
              if t.startswith('S')]
```

(Look at the WormAtlas page for a description of the different connection types.)

We use `weight` in a dictionary above because it is a special keyword for edge properties in NetworkX. We then build the graph using NetworkX's `DiGraph` class:

```
In [27]:

import networkx as nx
wormbrain = nx.DiGraph()
wormbrain.add_edges_from(conn_edges)
```

We can now examine some of the properties of this network. One of the first things researchers ask about directed networks is which nodes are the most critical to information flow within it. Nodes with high *betweenness centrality* are those that belong the shortest path between many different pairs of nodes. Think of a rail network: certain stations will connect to many lines, so that you will be forced to change lines there for many different trips. They are the ones with high betweenness centrality.

With networkx, we can find similarly important neurons with ease. In the networkx API documentation [^nxdoc], under "centrality", the docstring for `betweenness_centrality` [^bwcdoc] specifies a function that takes a graph as input and returns a dictionary mapping node IDs to betweenness centrality values (floating point values).

```
In [28]:

centrality = nx.betweenness_centrality(wormbrain)
```

Now we can find the neurons with highest centrality using the Python built-in function `sorted`:

```
In [29]:

central = sorted(centrality, key=centrality.__getitem__, reverse=True)
print(central[:5])

['AVAR', 'AVAL', 'PVCR', 'PVT', 'PVCL']
```

This returns the neurons AVAR, AVAL, PVCR, PVT, and PVCL, which have been implicated in how the worm responds to prodding: the AVA neurons link the worm's front touch receptors (among others) to neurons responsible for backward motion, while the PVC neurons link the rear touch receptors to forward motion.

These neurons' high centrality feels like a bit of an artifact of their placement controlling a large number of motor neurons. Yes, they are in many routes from sensory neurons to motor neurons. But all of the motor neurons do essentially the same thing, as hinted at by their generic names, VA 1-12. If we were to collapse them into one, the high centrality of the "command" neurons AVA R and L, and PVC R and L, might vanish. Returning to the rail lines example, suppose trains between Grand Central Station in New York City and Washington DC's Union Station could end up at one of 12 different platforms, *and we counted each of those as a separate train line*. The betweenness centrality of Grand Central would be inflated because from it you could get to Union Station platform 1, platform 2, etc. That's not necessarily very interesting.

Varshney *et al* study the properties of a *strongly connected component* of 237 neurons, out of a total of 279. In graphs, a *connected component* is a set of nodes that are reachable by some path through all the links. The connectome is a *directed* graph, meaning the edges *point* from one node to the other, rather than merely connecting them. In this case, a strongly connected component is one where all nodes are reachable from each other by traversing links *in the correct direction*. So A -> B -> C is not strongly connected, because there is no way to get to A from B or C. but A -> B -> C -> A *is* strongly connected.

In a neuronal circuit, you can think of the strongly connected component as the "brain" of the circuit, where the processing happens, while nodes upstream of it are inputs, and nodes downstream are outputs.

Box

The idea of cyclical neuronal circuits dates back to the 1950s. Here's a lovely paragraph about this idea from an article in *Nautilus*, "The Man Who Tried to Redeem the World With Logic", by Amanda Gefter:

If one were to see a lightning bolt flash on the sky, the eyes would send a signal to the brain, shuffling it through a chain of neurons. Starting with any given neuron in the chain, you could retrace the signal's steps and figure out just how long ago the lightning struck. Unless, that is, the chain is a loop. In that case, the information encoding the lightning bolt just spins in circles, endlessly. It bears no connection to the time at which the lightning actually occurred. It becomes, as McCulloch put it, "an idea wrenched out of time." In other words, a memory.

NetworkX makes straightforward work out of getting the largest strongly connected component from our `wormbrain` network:

```
In [30]:

sccs = nx.strongly_connected_component_subgraphs(wormbrain)
sccs = sorted(sccs, key=len, reverse=True)
giantscc = sccs[0]
print('The largest strongly connected component has %i nodes,' %
      giantscc.number_of_nodes(), 'out of %i total.' %
      wormbrain.number_of_nodes())
```

```
The largest strongly connected component has 237 nodes, out of 279 total.
```

As noted in the paper, the size of this component is *smaller* than expected by chance, demonstrating that the network is segregated into input, central, and output layers.

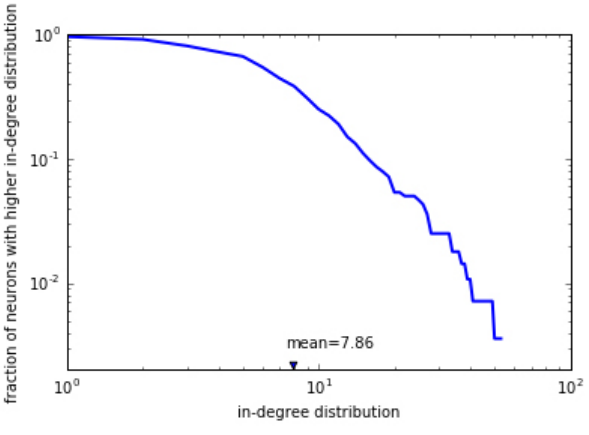
Now we reproduce figure 6B from the paper, the survival function of the in-degree distribution. First, compute the relevant quantities:

```
In [31]:

in_degrees = list(wormbrain.in_degree().values())
in_deg_distrib = np.bincount(in_degrees)
avg_in_degree = np.mean(in_degrees)
cumfreq = np.cumsum(in_deg_distrib) / np.sum(in_deg_distrib)
survival = 1 - cumfreq
```

Then, plot using Matplotlib:

```
In [32]:
```

```
plt.loglog(np.arange(1, len(survival) + 1), survival, c='b', lw=2)
plt.xlabel('in-degree distribution')
plt.ylabel('fraction of neurons with higher in-degree distribution')
plt.scatter(avg_in_degree, 0.0022, marker='v')
plt.text(avg_in_degree - 0.5, 0.003, 'mean=%.2f' % avg_in_degree, )
plt.ylim(0.002, 1.0)
plt.show()
```



Exercise: Use `scipy.optimize.curve_fit` to fit the tail of the in-degree survival function to a power-law, $f(d) \sim d^{-\gamma}$, $d > d_0$, for $d_0 = 10$ (the red line in Figure 6B of the paper), and modify the plot to include that line.

Region adjacency graphs

I hope that the previous section gave you an idea of the power of graphs as a scientific abstraction, and also how Python makes it easy to manipulate and analyse them. Now we will study a special kind of graph, the region adjacency graph, or RAG. This is a representation of an image that is useful for *segmentation*, the division of images into meaningful regions (or *segments*). If you've seen Terminator 2, you've seen segmentation:

Segmentation is one of those problems that humans do trivially, all the time, without thinking, whereas computers have a really hard time of it. To understand this difficulty, look at this image:

While you see a face, a computer only sees a bunch of numbers:

```
586888888888888999988989888888666532121
668888868889989999998999998888888865421
666655665666899999999999998888888888653
666688999986556889998999988888668665554
668888999988888888899888886656666666543
668888888868688688899988886666688888865
666664433345566888899888666666666668866
668842352214465888899886656446444444666
868644862336646668898866554643212423 45
866666583336855888888666555659381366324
888666866886668668888866585884224854 34
888888888886886888888665666866666565444
888888888686668888888665566886666686555
888889888888888888888866568868886866666
888899999899988888888866666888888868886
888899988888888888888865668888888888866
888889988888886888886665668688688888888
688889998888888888688665688888888888866
688889999988888886888886556888888888866
688889998866866688868886565668888888886
888888888866688888888886565588888888886
688888866656688888898885555556888888886
868688686586688686888865555555888866866
666888664688668555666554455556568888866
666886548888868686665555545566666666865
```

```
886886586888888888866666555566866886 65
688888866668888889888888666666566866 65
668888888456868889998888866666556866 655
666888888624566688666666544312686866 55
686888988866896966666556553136686886 55
688888988886689989989988853568889866 55
686888898888668999999998666666689866 55
688888888888666668888666666666888666 55
568888888886868899868686555666888865 55
366688888888688888686886666866886866 55
266868888888888888888886666886888658 654
286888888888888888886686666686866665 55
286666888888888888686686686888866655 48
```

(Yes, your visual system is tuned enough to find faces that it sees the face even in this blob of numbers! But I hope you get my point. Also, check out the "Faces In Things" Tumblr.)

So the challenge is to make sense of those numbers, and where the boundaries lie that divide the different parts of the image. A popular approach is to find small regions (called superpixels) that you're *sure* belong in the same segment, and then merge those according to some more sophisticated rule.

As a simple example, suppose you want to segment out the tiger in this picture, from the Berkeley Segmentation DataSet (BSDS) [^bsds-tiger]:

A clustering algorithm, simple linear iterative clustering (SLIC) [^slic], can give us a decent starting point. It is available in the scikit-image library.

```
In [33]:
```

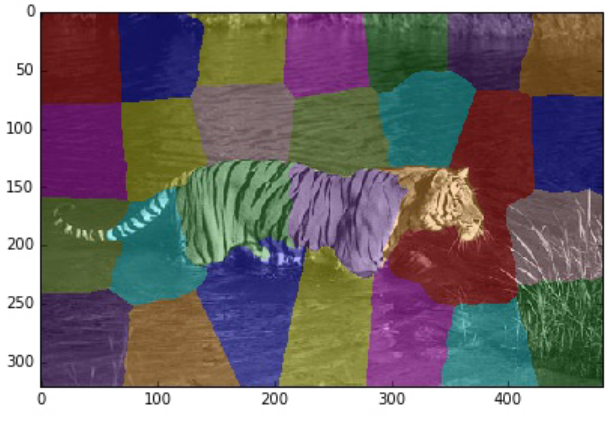
```
url =
'http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300/html/images/plain/normal/co
tiger = io.imread(url)
from skimage import segmentation
seg = segmentation.slic(tiger, n_segments=30, compactness=40.0,
                        enforce_connectivity=True, sigma=3)
```

Scikit-image also has a function to *display* segmentations, which we use to visualize the result of SLIC:

```
In [34]:
```

```
from skimage import color
io.imshow(color.label2rgb(seg, tiger))
```

```
Out[34]:

<matplotlib.image.AxesImage at 0x1162732b0>
```



This shows that the tiger has been split in three parts, with the rest of the image in the remaining segments.

A region adjacency graph (RAG) is a graph in which every node represents one of the above regions, and an edge connects two nodes when they touch. For a taste of what it looks like before we build one, we'll use the `draw_rag` function from scikit-image — indeed, the library that contains this chapter's code snippet!

```
In [35]:
```

```
from skimage.future import graph
g = graph.rag_mean_color(tiger, seg)
graph.draw_rag(seg, g, tiger)
```

```
Out[35]:
array([[[ 0.19607843, 0.21568627, 0.2 ],
        [ 0.14901961, 0.16862745, 0.15294118],
```

```
 [ 0.10980392, 0.12941176, 0.11372549],
 ...,
 [ 0.23921569, 0.42352941, 0.21176471],
 [ 0.23921569, 0.43137255, 0.21568627],
 [ 0.24705882, 0.41568627, 0.24705882]],

[[ 0.2       , 0.21960784, 0.20392157],
 [ 0.14117647, 0.16078431, 0.14509804],
 [ 0.09019608, 0.10980392, 0.09411765],
 ...,
 [ 0.22745098, 0.41960784, 0.20392157],
 [ 0.15686275, 0.34901961, 0.13333333],
 [ 0.20392157, 0.38431373, 0.20392157]],

[[ 0.18823529, 0.20784314, 0.19215686],
 [ 0.1254902 , 0.14509804, 0.12941176],
 [ 0.0745098 , 0.09803922, 0.08235294],
 ...,
 [ 0.20784314, 0.4 , 0.18431373],
 [ 0.11372549, 0.30588235, 0.09019608],
 [ 0.16470588, 0.34509804, 0.16470588]],

 ...,
[[ 0.25490196, 0.2745098 , 0.14901961],
 [ 0.26666667, 0.28627451, 0.16078431],
 [ 0.27843137, 0.29803922, 0.17254902],
 ...,
 [ 0.29803922, 0.33333333, 0.22745098],
 [ 0.23921569, 0.2745098 , 0.16862745],
 [ 0.29019608, 0.32941176, 0.19607843]],

[[ 0.35294118, 0.36470588, 0.24313725],
 [ 0.35294118, 0.36470588, 0.24313725],
 [ 0.34901961, 0.36078431, 0.23921569],
 ...,
 [ 0.30588235, 0.34117647, 0.22745098],
 [ 0.25490196, 0.29019608, 0.17647059],
 [ 0.23529412, 0.2745098 , 0.1372549 ]],

[[ 0.32156863, 0.3254902 , 0.2 ],
 [ 0.36862745, 0.37254902, 0.24705882],
 [ 0.36470588, 0.36862745, 0.24313725],
 ...,
 [ 0.33333333, 0.35686275, 0.22352941],
 [ 0.32941176, 0.35294118, 0.21960784],
 [ 0.30980392, 0.33333333, 0.2       ]]])
```

Here, you can see the nodes corresponding to each segment, and the edges between adjacent segments. These are colored with the YlGnBu (yellow-green-blue) colormap from matplotlib, according to the difference in color between the two nodes.

The figure also shows the magic of thinking of segmentations as graphs: you can see that edges between nodes within the tiger and those outside of it are darker (higher-valued) than edges within the same object. Thus, if we can cut the graph along those edges, we will get our segmentation! (Yes, I have chosen an easy example for color-based segmentation, but the same principles hold true for graphs with more complicated pairwise relationships!)

Elegant ndimage

All the pieces are in place: you know about numpy arrays, image filtering, generic filters, graphs, and region adjacency graphs. Let's build one to pluck the tiger out of that picture!

The obvious approach is to use two nested for-loops to iterate over every pixel of the image, look at the neighboring pixels, and checking for different labels:

```
In [36]:
```

```
import networkx as nx
def build_rag(labels, image):
    g = nx.Graph()
    nrows, ncols = labels.shape
    for row in range(nrows):
        for col in range(ncols):
            current_label = labels[row, col]
            if not current_label in g:
                g.add_node(current_label)
                g.node[current_label]['total color'] = np.zeros(3, dtype=np.float)
                g.node[current_label]['pixel count'] = 0
            if row < nrows - 1 and labels[row + 1, col] != current_label:
                g.add_edge(current_label, labels[row + 1, col])
            if col < ncols - 1 and labels[row, col + 1] != current_label:
                g.add_edge(current_label, labels[row, col + 1])
            g.node[current_label]['total color'] += image[row, col]
            g.node[current_label]['pixel count'] += 1
    return g
```

This works, but if you want to segment a 3D image, you'll have to write a different version:

```
In [37]:
```

```
import networkx as nx
def build_rag_3d(labels, image):
    g = nx.Graph()
    nplns, nrows, ncols = labels.shape
    for pln in range(nplns):
        for row in range(nrows):
            for col in range(ncols):
                current_label = labels[pln, row, col]
                if not current_label in g:
```

```
                    g.add_node(current_label)
                    g.node[current_label]['total color'] = np.zeros(3, dtype=np.float)
                    g.node[current_label]['pixel count'] = 0
                if pln < nplns - 1 and labels[pln + 1, row, col] != current_label:
                    g.add_edge(current_label, labels[pln + 1, row, col])
                if row < nrows - 1 and labels[pln, row + 1, col] != current_label:
                    g.add_edge(current_label, labels[pln, row + 1, col])
                if col < ncols - 1 and labels[pln, row, col + 1] != current_label:
                    g.add_edge(current_label, labels[pln, row, col + 1])
                g.node[current_label]['total color'] += image[pln, row, col]
                g.node[current_label]['pixel count'] += 1
    return g
```

Both of these are pretty ugly and unwieldy, too. And difficult to extend: if we want to count diagonally neighboring pixels as adjacent (that is, [row, col] is "adjacent to" [row + 1, col + 1]), the code becomes even messier. And if we want to analyze 3D video, we need yet another dimension, and another level of nesting. It's a mess!

Enter Vighnesh's insight: SciPy's `generic_filter` function already does this iteration for us! We used it above to compute an arbitrarily complicated function on the neighborhood of every element of a numpy array. Only now we don't want a filtered image out of the function: we want a graph. It turns out that `generic_filter` lets you pass additional arguments to the filter function, and we can use that to build the graph:

```
In [38]:
```

```
import networkx as nx
import numpy as np
from scipy import ndimage as nd

def add_edge_filter(values, graph):
    center = values[len(values) // 2]
    for neighbor in values:
        if neighbor != center and not graph.has_edge(center, neighbor):
            graph.add_edge(center, neighbor)
    # float return value is unused but needed by `generic_filter`
    return 0.0

def build_rag(labels, image):
    g = nx.Graph()
    footprint = ndi.generate_binary_structure(labels.ndim, connectivity=1)
    _ = ndi.generic_filter(labels, add_edge_filter, footprint=footprint,
                           mode='nearest', extra_arguments=(g,))
    for n in g:
        g.node[n]['total color'] = np.zeros(3, np.double)
        g.node[n]['pixel count'] = 0
    for index in np.ndindex(labels.shape):
        n = labels[index]
        g.node[n]['total color'] += image[index]
        g.node[n]['pixel count'] += 1
    return g
```

There's a few things to notice here:

- we return "0.0" from the filter function because `generic_filter` requires the filter function to return a float. However, we will ignore the filter output, and only use it for its "side effect" of adding edges to the graph.
- the loops are not nested several levels deep. This makes the code more compact, easier to take in in one go.
- the code works identically for 1D, 2D, 3D, or even 8D images!
- if we want to add support for diagonal connectivity, we just need to change the `connectivity` parameter to `ndi.generate_binary_structure`
- `ndi.generic_filter` iterates over array elements *with their neighbors*; use `numpy.ndindex` to simply iterate over array indices.

Overall, I think this is just a brilliant piece of code.

Putting it all together: mean color segmentation

Now, we can use it to segment the tiger in the image above:

```
In [39]:
```

```
g = build_rag(seg, tiger)
for n in g:
    node = g.node[n]
    node['mean'] = node['total color'] / node['pixel count']
for u, v in g.edges_iter():
    d = g.node[u]['mean'] - g.node[v]['mean']
    g[u][v]['weight'] = np.linalg.norm(d)
```

Each edge holds the difference between the average color of each segment. We can now threshold the graph:

```
In [40]:
```

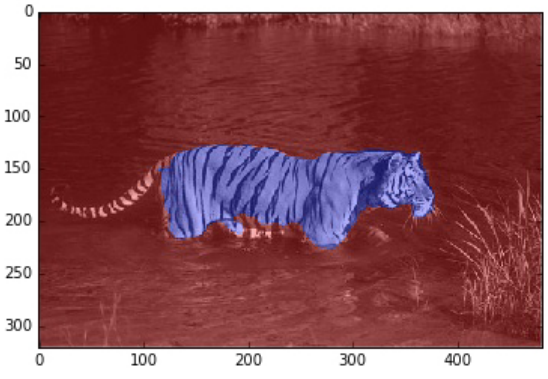
```
def threshold_graph(g, t):
    to_remove = ((u, v) for (u, v, d) in g.edges(data=True)
                 if d['weight'] > t)
    g.remove_edges_from(to_remove)
threshold_graph(g, 80)
```

Finally, we use the numpy index-with-an-array trick we learned in Chapter 2:

```
In [41]:
```

```
map_array = np.zeros(np.max(seg) + 1, int)
for i, segment in enumerate(nx.connected_components(g)):
    for initial in segment:
        map_array[int(initial)] = i
```

```
segmented = map_array[seg]
plt.imshow(color.label2rgb(segmented, tiger));
```



Oops! Looks like the cat lost its tail!

Still, we think that's a nice demonstration of the capabilities of RAGs... And the beauty with which SciPy and NetworkX make it feasible!

Many of these functions are available in the scikit-image library. If you are interested in image analysis, check it out!

CHAPTER 4
Frequency and the Fast Fourier Transform

This chapter was written in collaboration with SW's father, PW van der Walt.

This chapter will depart slightly from the format of the rest of the book. In particular, you may find the *code* in the chapter quite modest. Instead, we want to illustrate an elegant *algorithm*, the Fast Fourier Transform (FFT), that is endlessly useful, implemented in SciPy, and works, of course, on NumPy arrays.

We'll start by setting up some plotting styles and importing the usual suspects:

```
In [1]:
```

```
%matplotlib inline

import seaborn as sns
sns.set_style('white')
sns.despine()

import matplotlib.pyplot as plt
import numpy as np
```

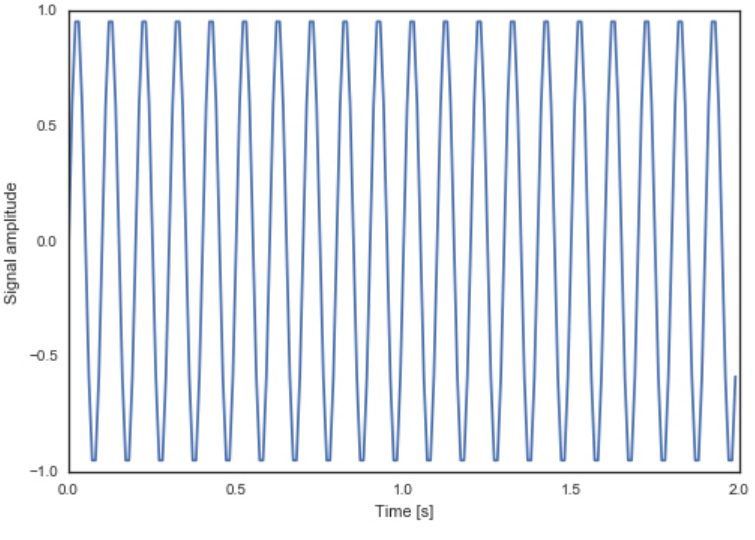
```
<matplotlib.figure.Figure at 0x10414b2b0>
```

The discrete[^discrete] Fourier Transform is a mathematical technique to convert temporal or spatial data into *frequency domain* data. *Frequency* is a familiar concept, due to its colloquial occurrence in the English language: the lowest notes your headphones can rumble out are around 20 Hertz, whereas middle C on a piano lies around 261.6 Hertz. Hertz (Hz), or oscillations per second, in this case literally refers to the number of times per second at which the membrane inside the headphone moves to-and-fro. That, in turn, creates compressed pulses of air which, upon arrival at your eardrum, induces a vibration at the same frequency. So, if you take a simple periodic function, $\sin(10 * 2\pi t)$, you can view it as a wave:

```
In [2]:
```

```
f = 10  # Frequency, in cycles per second, or Herz
f_s = 100  # Sampling rate, or number of measurements per second

t = np.linspace(0, 2, 2 * f_s, endpoint=False)
x = np.sin(f * 2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, x)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Signal amplitude')
```

```
Out[2]:

<matplotlib.text.Text at 0x10f8c12b0>
```

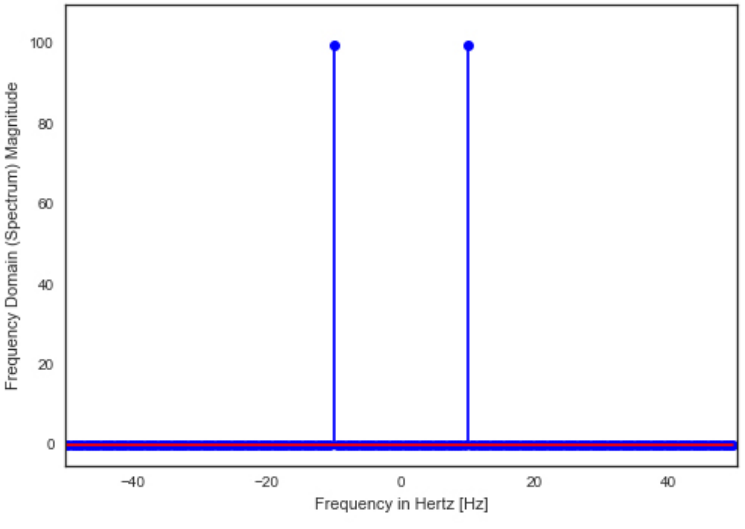


Or you can equivalently think of it as an repeating signal of *frequency* 10 Hertz (it repeats once every 1/10 seconds—a length of time we call its *period*):

```
In [3]:
```

```
from scipy import fftpack

X = fftpack.fft(x)
freqs = fftpack.fftfreq(len(x)) * f_s

fig, ax = plt.subplots()

ax.stem(freqs, np.abs(X))
ax.set_xlabel('Frequency in Hertz [Hz]')
ax.set_ylabel('Frequency Domain (Spectrum) Magnitude')
ax.set_xlim(-f_s / 2, f_s / 2)
ax.set_ylim(-5, 110)
```

```
Out[3]:

(-5, 110)
```

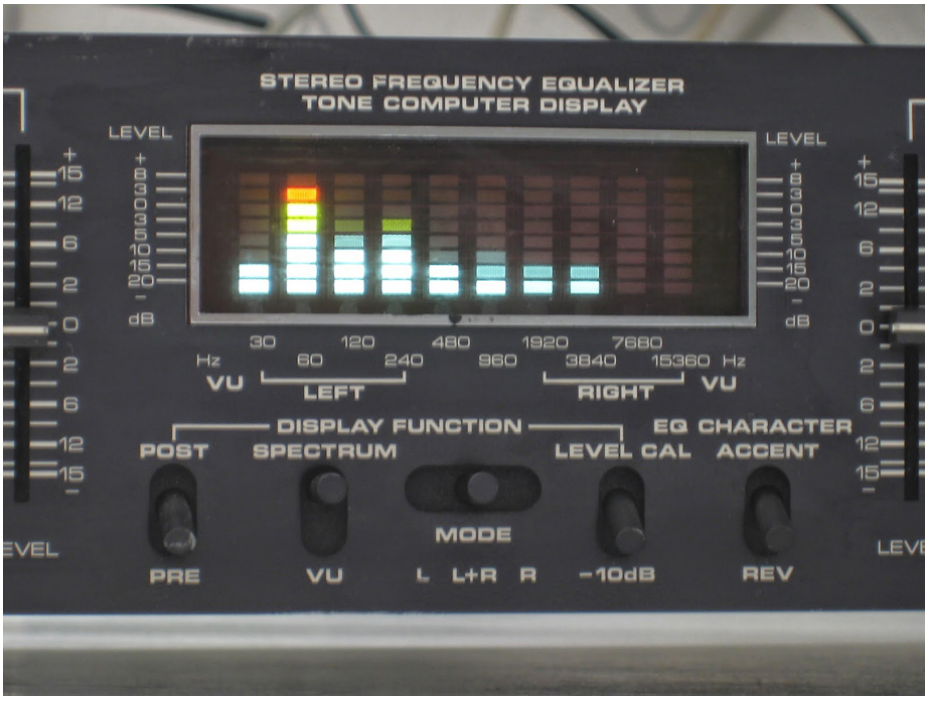


(We'll explain why you see positive and negative frequencies later on.)

The Fourier transform takes us from the *time* to the *frequency* domain, and this turns out to have a massive number of applications. The *Fast Fourier Transform (FFT)* is an algorithm for computing the discrete Fourier Transform; it achieves its high speed by storing and re-using results of computations as it progresses.

In this chapter, we examine a few applications of the Discrete Fourier Transform to demonstrate that the FFT can be applied to multidimensional data (not just 1D measurements) to achieve a variety of goals.

Let's start with one of the most common applications, converting a sound signal (consisting of variations of air pressure over time) to a *spectrogram*. (You might have seen spectrograms on your music player's equalizer view, or even on an old-school stereo.)



[ED NOTE: Used with permission from the author, Sergey Gerasimuk, http://sgerasimuk.blogspot.com/2014/06/numark-eq-2600-10-band-stereo-graphic.html]

Listen to the following snippet of nightingale birdsong:

```
In [4]:
```

```
from IPython.display import Audio
Audio('data/nightingale.wav')
```

```
Out[4]:
```

Your browser does not support the audio element.

If you are reading the paper version of this book, you'll have to use your imagination! It goes something like this: chee-chee-woorrrr-hee-hee cheet-wheet-hoorrr-chi rrr-whi-wheo-wheo-wheo-wheo-wheo-wheo.

Since we realise that not everyone is fluent in bird-speak, perhaps it's best if we visualize the measurements—better known as "the signal"—instead.

We load the audio file (released under CC BY 4.0 at http://www.orangefreesounds.com/nightingale-sound/), which gives us the sampling rate (number of measurements per second) as well as audio data as an `(N, 2)` array—two columns because this is a stereo recording.

```
In [5]:
```

```
from scipy.io import wavfile

rate, audio = wavfile.read('data/nightingale.wav')
```

We convert to mono by averaging the left and right channels.

```
In [6]:
```

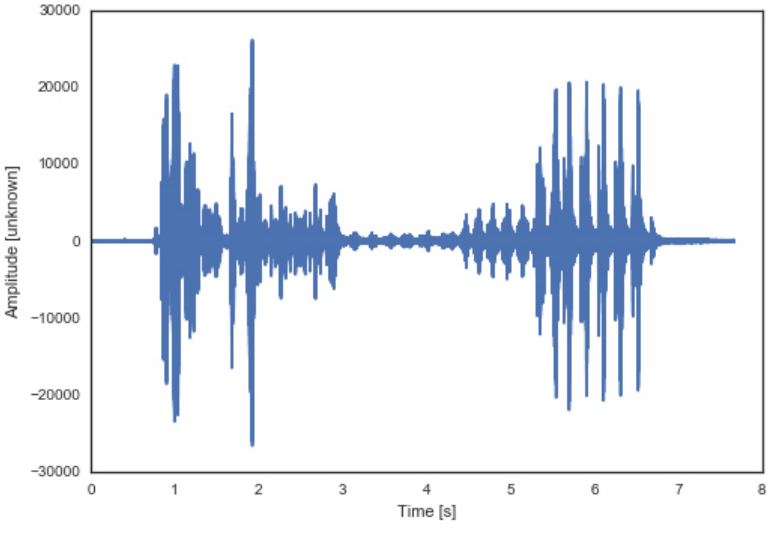
```
audio = np.mean(audio, axis=1)
```

Then, calculate the length of the snippet and plot the audio.

```
In [7]:
```

```
N = audio.shape[0]
L = N / rate

print('Audio length: {:.2f} seconds'.format(L))

f, ax = plt.subplots()
ax.plot(np.arange(N) / rate, audio)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Amplitude [unknown]')
plt.show()
```

```
Audio length: 7.67 seconds
```



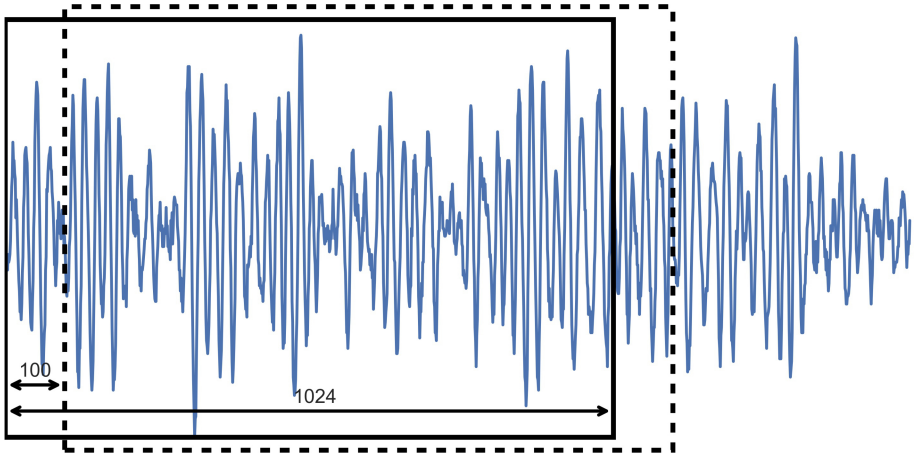
Well, that's not very satisfying, is it! If I sent this voltage to a speaker, I might hear a bird chirping, but I can't very well imagine how it would sound like in my head. Is there a better way of *seeing* what is going on?

There is, and it is called the Discrete Fast Fourier Transform (discrete, because our recording consists of discrete measurements, and fast—because we're in a hurry!). The Fast Fourier Transform (FFT) tells us which frequencies or "notes" to expect in our signal.

Of course, a bird sings many notes throughout the song, so we'd also like to know *when* each note occurs. The Fourier transform takes a signal in the time domain (i.e., a set of measurements over time) and turns it into a spectrum—a set of frequencies with corresponding (complex[1]) values. The spectrum does not contain any information about time![2]

So, to find both the frequencies and the time at which they were sung, we'll need to be somewhat clever. Our strategy will be as follows: take the audio signal, split it into small, overlapping slices, and apply the Fourier transform to each (a technique known as the Short Time Fourier Transform).

We'll split the signal into slices of 1024 samples—that's about 0.02 seconds of audio. Why we choose 1024 and not 1000 we'll explain in a second when we examine performance. The slices will overlap by 100 samples as shown here:



Start by chopping up the signal into slices of 1024 samples, each slice overlapping the previous by 100 samples. The resulting `slices` object contains one slice per row.

1. The Fourier transform essentially tells us how to combine a set of sinusoids of varying frequency to form the input signal. The spectrum consists of complex numbers—one for each sinusoid. A complex number encodes two things: a magnitude and an angle. The magnitude is the strength of the sinusoid in the signal, and the angle how much it is shifted in time. At this point, we only care about the magnitude, which we calculate using `np.abs`.

2. For more on techniques for calculating both (approximate) frequencies and time of occurrence, read up on wavelet analysis.

```
In [8]:

from skimage import util

M = 1024

slices = util.view_as_windows(audio, window_shape=(M,), step=100)
print('Audio shape: {}, Sliced audio shape: {}'.format(audio.shape, slices.shape))

Audio shape: (338081,), Sliced audio shape: (3371, 1024)
```

Generate a windowing function and multiply it with the signal—more on this later:

```
In [9]:

win = np.hanning(M + 1)[:-1]
slices = slices * win
```

It's more convenient to have one slice per column, so we take the transpose:

```
In [10]:

slices = slices.T
print('Shape of `slices`:', slices.shape)

Shape of `slices`: (1024, 3371)
```

For each slice, calculate the Fourier transform. The Fourier transform returns both positive and "negative" frequencies (more on that in "Frequencies and their ordering"), so we slice out the positive M / 2 frequencies for now.

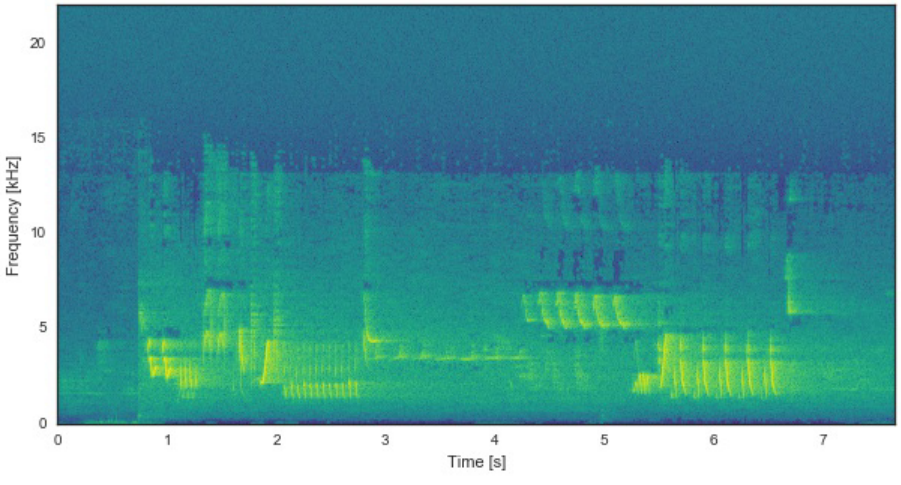
```
In [11]:

spectrum = np.fft.fft(slices, axis=0)[:M // 2 + 1:-1]
spectrum = np.abs(spectrum)
```

Do a log plot of the ratio of the signal / the maximum signal The unit for such a ratio is the decibel.

Another reason to take logs is because the spectrum can contain both very large and very small values. Taking the log compresses the range significantly.

```
In [12]:

f, ax = plt.subplots(figsize=(10, 5))

S = np.abs(spectrum)
S = 20 * np.log10(S / S.max())

ax.imshow(S, cmap='viridis', origin='lower',
          extent=(0, L, 0, rate / 2 / 1000))
ax.axis('tight')
ax.set_ylabel('Frequency [kHz]')
```

```
ax.set_xlabel('Time [s]')
plt.show()
```



Much better! We can now see the frequencies vary over time, and it corresponds to the way the audio sounds. See if you can match my earlier description: chee-chee-woorrrr-hee-hee cheet-wheet-hoorrr-chi rrr-whi-wheo-wheo-wheo-wheo-wheo-wheo (I didn't transcribe the section from 3 to 5 seconds—that's another bird).

SciPy already includes an implementation of this procedure as `scipy.signal.spectrogram`, which can be invoked as follows:

```
In [13]:

from scipy import signal

freqs, times, Sx = signal.spectrogram(audio, fs=rate, window='hanning',
                                      nperseg=1024, noverlap=M - 100,
                                      detrend=False, scaling='spectrum')

## Plot using:
# plt.pcolormesh(times, freqs, 10 * np.log10(Sx), cmap='viridis');
```

The only differences are that SciPy returns the spectrum squared (which turns measured voltage into measured energy), and multiplies it by some normalization factors[1].

History

Tracing the exact origins of the Fourier transform is tricky. Some related procedures go as far back as Babylonian times, but it was the hot topics of calculating asteroid orbits and solving the heat (flow) equation that led to several breakthroughs in the early 1800s. Whom exactly among Clairaut, LaGrange, Euler, Gauss and D'Alembert

we should thank is not exactly clear, but Gauss was the first to describe the Fast Fourier Transform (an algorithm for computing the Discrete Fourier Transform, popularized by Cooley and Tukey in 1965). Joseph Fourier, after whom the transform is named, first claimed that *arbitrary* functions can be expressed as a sum of trigonometric functions.

Implementation

The Fourier Transform functionality in SciPy lives in the `scipy.fftpack`` module. Among other things, it provides the following FFT-related functionality:

- `fft`, `fft2`, `fftn`: Compute the Fast (discrete) Fourier Transform
- `ifft`, `ifft2`, `ifftn`: Compute the inverse of the FFT
- `dct`, `idct`, `dst`, `idst`: Compute the cosine and sine transforms
- `fftshift`, `ifftshift`: Shift the zero-frequency component to the center of the spectrum and back, respectively (more about that soon)
- `fftfreq`: Return the Discrete Fourier Transform sample frequencies

This is complemented by the following functions in NumPy:

- `np.hanning`, `np.hamming`, `np.bartlett`, `np.blackman`, `np.kaiser`: Tapered windowing functions.

It is also used to perform fast convolutions of large inputs by `scipy.signal.fftconvolve`.

SciPy wraps the Fortran FFTPACK library—it is not the fastest out there, but unlike packages such as FFTW, it has a permissive free software license.

Consider that a naive calculation of the FFT takes $\mathcal{O}\left(N^2\right)$ operations. How come? Well, you have N (complex) sinusoids of different frequencies ($2\pi f \times 0, 2\pi f \times 1, 2\pi f \times 3, \ldots, 2\pi f \times (N-1)$), and you want to see how strongly your signal corresponds to each. Starting with the first, you take the dot product with the signal (which, in itself, entails N multiplication operations). Repeating this operation N times, once for each sinusoid, then gives N^2 operations.

Now, contrast that with the Fast Fourier Transform, which is $\mathcal{O}(N \log N)$ in the ideal case—a great improvement! However, the classical Cooley-Tukey algorithm implemented in FFTPACK recursively breaks up the transform into smaller (prime-sized) pieces and only shows this improvement for "smooth" input lengths (an input length is considered smooth when its largest prime factor is small). For large prime sized pieces, the Bluestein or Rader algorithms can be used in conjunction with the

Cooley-Tukey algorithm, but this optimization is not implemented in FFTPACK. [^fast]

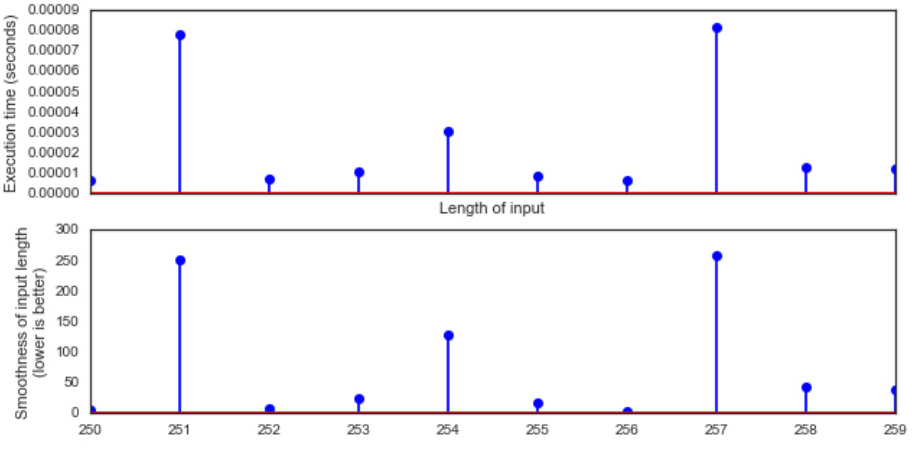
Let us illustrate:

1. SciPy goes to some effort to preserve the energy in the spectrum. Therefore, when taking only half the components, it multiplies the remaining components, apart from the first and last components, by two (those two components are "shared" by the two halves of the spectrum). It also normalizes the window by dividing it by its sum.

```
In [14]:
```

```
import time

from scipy import fftpack
from sympy import factorint

K = 1000
lengths = range(250, 260)

# Calculate the smoothness for all input lengths
smoothness = [max(factorint(i).keys()) for i in lengths]


exec_times = []
for i in lengths:
    z = np.random.random(i)

    # For each input length i, execute the FFT K times
    # and store the execution time

    times = []
    for k in range(K):
        tic = time.monotonic()
        fftpack.fft(z)
        toc = time.monotonic()
        times.append(toc - tic)

    # For each input length, remember the *minimum* execution time
    exec_times.append(min(times))


f, (ax0, ax1) = plt.subplots(2, 1, sharex=True, figsize=(10, 5))
ax0.stem(lengths, exec_times)
ax0.set_xlabel('Length of input')
ax0.set_ylabel('Execution time (seconds)')

ax1.stem(lengths, smoothness)
ax1.set_ylabel('Smoothness of input length\n(lower is better)')
```

```
plt.show()
```



The intuition is that, for smooth numbers, the FFT can be broken up into many small pieces. After performing the FFT on the first piece, those results can be reused in subsequent computations. This explains why we chose a length of 1024 for our audio slices earlier—it has a smoothness of only 2, resulting in the optimal "radix-2 Cooley-Tukey" algorithm, which computes the FFT using only $(N/2)\log_2 N = 5120$ complex multiplications, instead of $N^2 = 1048576$.

Discrete Fourier Transform concepts

Next, we present a couple of common concepts worth knowing before operating heavy Fourier Transform machinery, whereafter we tackle another real-world problem: analyzing target detection in radar data.

Frequencies and their ordering

For historical reasons, most implementations return an array where frequencies vary from low-to-high-to-low. E.g., when we do the real Fourier transform of a signal of all ones, an input that has no variation and therefore only has the slowest, constant Fourier component (also known as the "DC" or Direct Current component—just electronics jargon for "mean of the signal"), appearing as the first entry:

```
In [15]:

from scipy import fftpack
N = 10

fftpack.fft(np.ones(N)) # Note first component is np.mean(x) * N
```

```
Out[15]:

array([ 10.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,
         0.-0.j,   0.-0.j,   0.-0.j,   0.-0.j])
```

Note that the FFT returns a complex spectrum which, in the case of real inputs, is conjugate symmetrical.

When we try the FFT on a rapidly changing signal, we see a high frequency component appear:

```
In [16]:

z = np.ones(10)
z[::2] = -1

print("Applying FFT to {}".format(z))
fftpack.fft(z)
```

```
Applying FFT to [-1. 1. -1. 1. -1. 1. -1. 1. -1. 1.]
```

```
Out[16]:

array([  0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j, -10.+0.j,
         0.-0.j,   0.-0.j,   0.-0.j,   0.-0.j])
```

The `fftfreq` function tells us which frequencies we are looking at:

```
In [17]:

fftpack.fftfreq(10)
```

```
Out[17]:

array([ 0. ,  0.1,  0.2,  0.3,  0.4, -0.5, -0.4, -0.3, -0.2, -0.1])
```

The result tells us that our maximum component occured at a frequency of 0.5 cycles per sample. This agrees with the input, where a minus-one-plus-one cycle repeated every second sample.

Sometimes, it is convenient to view the spectrum organized slightly differently, from high-negative to low to-high-positive (for now, we won't dive too deeply into the concept of negative frequency, other than saying a real-world sine wave is produced by a combination of positive and negative frequencies). We re-shuffle the spectrum using the `fftshift` function. Let's examine the frequency components in a noisy image.

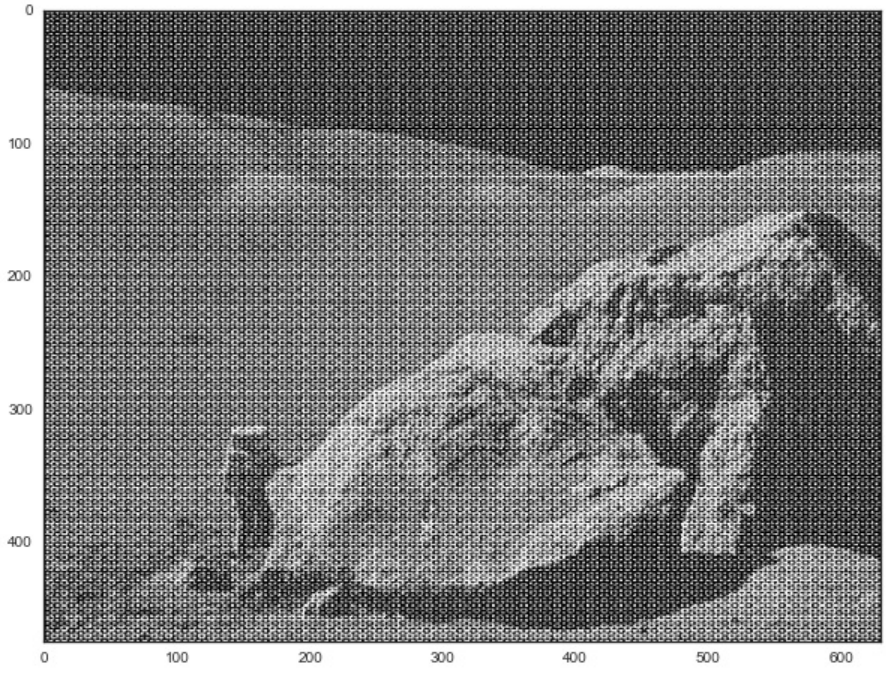
First, load and display the image:

```
In [18]:

from skimage import io
image = io.imread('images/moonlanding.png')
M, N = image.shape

f, ax = plt.subplots(figsize=(10, 10))
```

```
ax.imshow(image, cmap='gray')
```

```
print((M, N), image.dtype)
```

```
(474, 630) uint8
```



Do not adjust your monitor! The image you are seeing is real, although clearly distorted by either the measurement or transmission equipment.

To examine the spectrum of the image, we use `fftn` (instead of `fft`) to compute the FFT, since it has more than one dimension. The two-dimensional FFT is equivalent to taking the 1-D FFT across rows and then across columns (or vice versa).

```
In [19]:
```

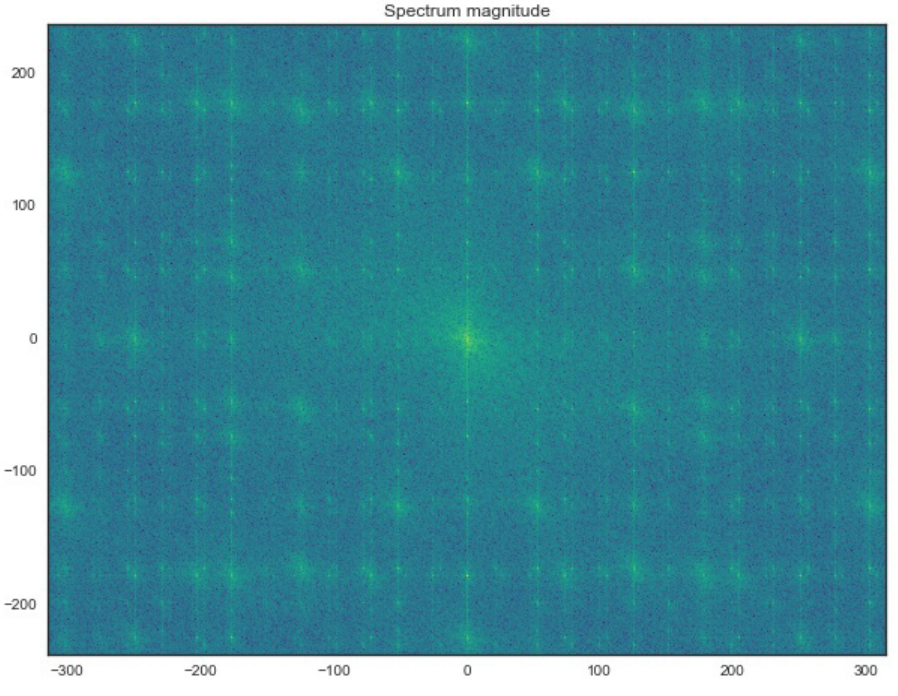
```
F = fftpack.fftn(image)

F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)
```

Again, we take the log of the spectrum to compress the range of values, before displaying:

```
In [20]:
```

```
f, ax = plt.subplots(figsize=(10, 10))
```

```
ax.imshow(np.log(1 + F_magnitude),
          cmap='viridis', interpolation='nearest',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude')
plt.show()
```



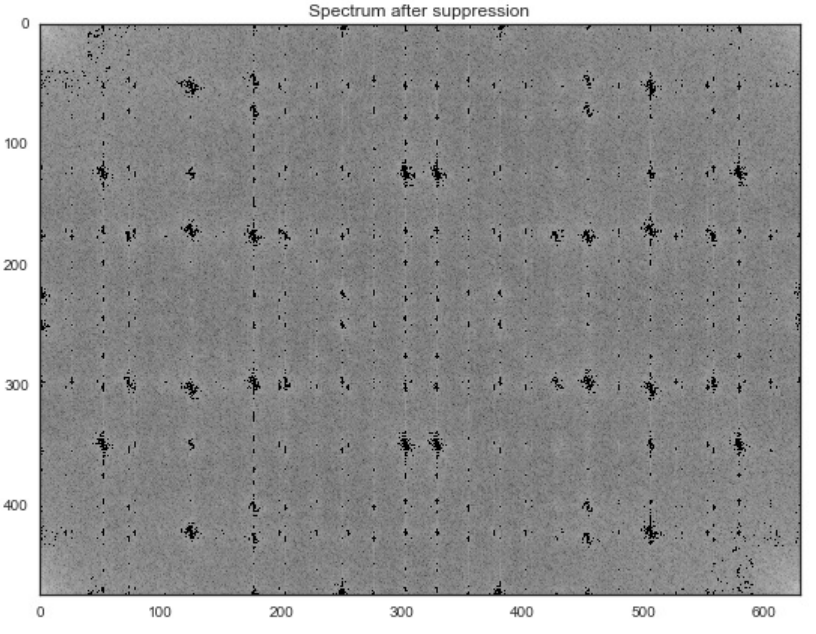
Note the high values around the origin (middle) of the spectrum—these coefficients describe the low frequencies or smooth parts of the image; a vague canvas of the photo. Higher frequency components, spread throughout the spectrum, fill in the edges and detail. Peaks around higher frequencies correspond to the periodic noise.

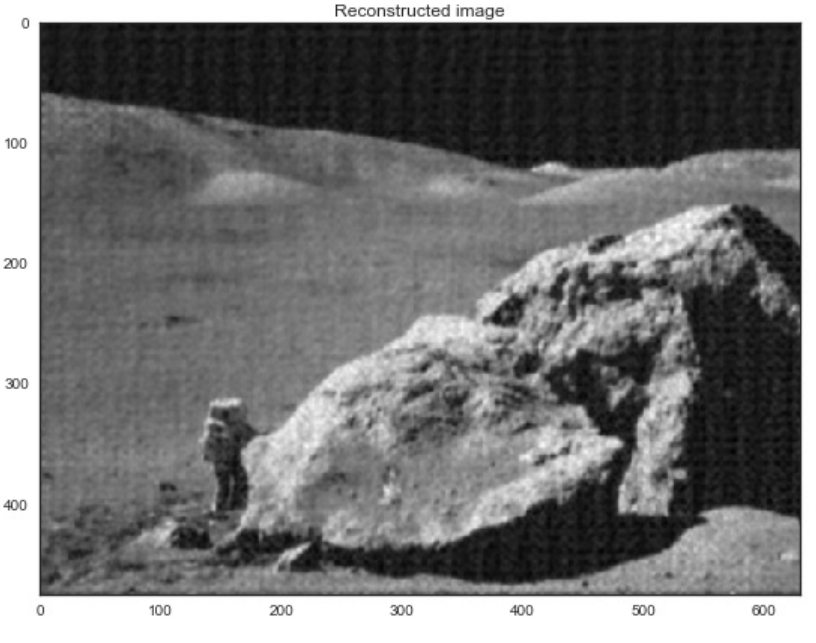
From the photo, we can see that the noise (measurement artifacts) is highly periodic, so we hope to remove it by zeroing out the corresponding parts of the spectrum.

The image with those peaks suppressed indeed looks quite different!

```
In [21]:

# Set block around center of spectrum to zero
K = 40
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

# Find all peaks higher than the 98th percentile
peaks = F_magnitude < np.percentile(F_magnitude, 98)
```

```
# Shift the peaks back to align with the original spectrum
peaks = fftpack.ifftshift(peaks)

# Make a copy of the original (complex) spectrum
F_dim = F.copy()

# Set those peak coefficients to zero
F_dim = F_dim * peaks.astype(int)

# Do the inverse Fourier transform to get back to an image
# Since we started with a real image, we only look at the real part of
# the output.
image_filtered = np.real(fftpack.ifft2(F_dim))

# And add a slight bit of blurring to soften the result
from scipy import ndimage
image_filtered = ndimage.gaussian_filter(image_filtered, sigma=1)

f, (ax0, ax1) = plt.subplots(2, 1, figsize=(20, 15))
ax0.imshow(np.log10(1 + np.abs(F_dim)), cmap='gray', interpolation='nearest')
ax0.set_title('Spectrum after suppression')

ax1.imshow(ndimage.gaussian_filter(image_filtered, sigma=1), cmap='gray',
interpolation='nearest')
ax1.set_title('Reconstructed image')

plt.show()
```

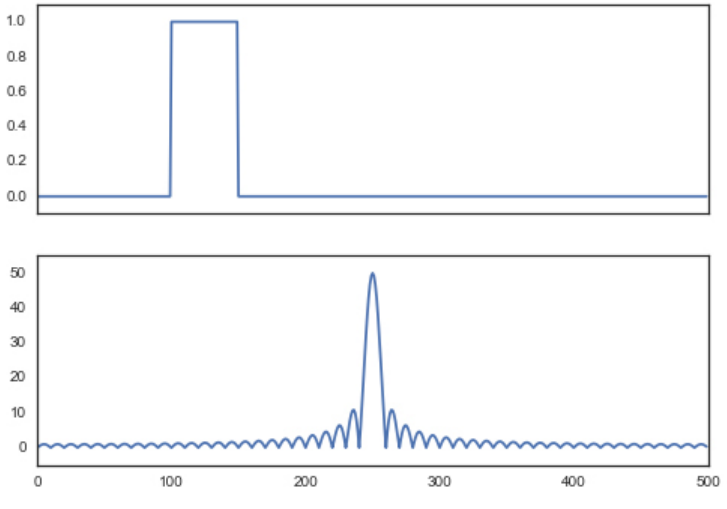

Spectrum after suppression
0
100
200
300
400
0
100
200
300
400
500
600


Reconstructed image
0
100
200
300
400
0
100
200
300
400
500
600

Windowing

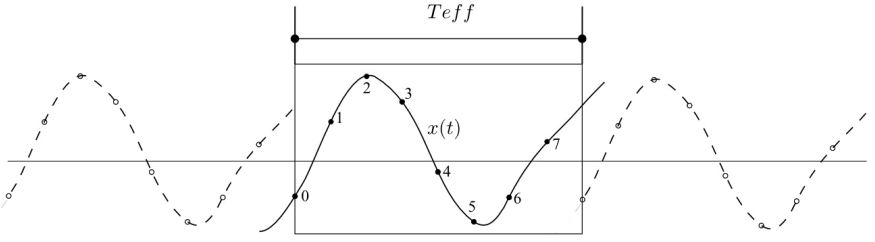
If we examine the Fourier transform of a step function, we see significant ringing in the spectrum:

```
In [22]:

x = np.zeros(500)
x[100:150] = 1

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1, sharex=True)

ax0.plot(x)
ax0.set_ylim(-0.1, 1.1)

ax1.plot(fftpack.fftshift(np.abs(X)))
ax1.set_ylim(-5, 55)
plt.show()
```



In theory, you would need a combination of infinitely many sinusoids (frequencies) to make an ideal step function; the coefficients would have the ringing shape shown.

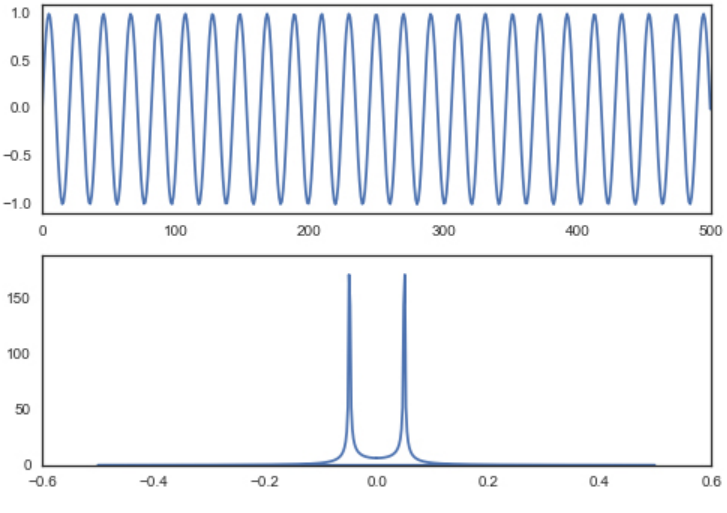
Importantly, the Fourier transform assumes that the input signal is periodic. If the signal is not, the assumption is simply that, right at the end of the signal, it jumps back to its beginning value. Consider the function, $x(t)$, shown here:



We only measure the signal for a short time, labeled T_{eff}. The Fourier transform assumes that $x(8) = x(0)$, and that the signal is continued as the dashed, rather than the solid line. This introduces a big jump at the edge, with the expected ossilation in the spectrum:

```
In [23]:
```

```
t = np.linspace(0, 1, 500)
x = np.sin(49 * np.pi * t)

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1)

ax0.plot(x)
ax0.set_ylim(-1.1, 1.1)

ax1.plot(fftpack.fftfreq(len(t)), np.abs(X))
ax1.set_ylim(0, 190)
plt.show()
```

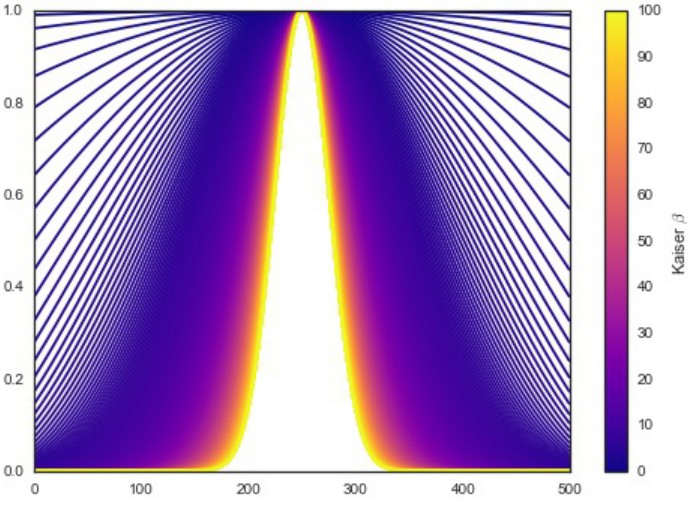


Instead of the expected two sharp peaks, they are spread out in the spectrum.

We can counter this effect by a process called *windowing*. The original function is multiplied with a window function such as the Kaiser window $K(N, \beta)$. Here we visualize it for β ranging from 0 to 100:

```
In [24]:
```

```
f, ax = plt.subplots()

N = 500
beta_max = 100
colormap = plt.cm.plasma

norm = plt.Normalize(vmin=0, vmax=beta_max)

lines = [
    ax.plot(np.kaiser(N, beta), color=colormap(norm(beta)))
    for beta in np.linspace(0, beta_max, N)
    ]

sm = plt.cm.ScalarMappable(cmap=colormap, norm=norm)

# Dirty hack, not sure why matplotlib >= 1.4 introduced this "feature"
sm._A = []

plt.colorbar(sm).set_label(r'Kaiser $\beta$')

plt.show()
```



By changing the parameter β, the shape of the window can be changed from rectangular ($\beta = 0$, no windowing) to a window that produces signals that smoothly

increase from zero and decrease to zero at the endpoints of the sampled interval, producing very low side lobes (β typically between 5 and 10).

Applying the Kaiser window here, we see that the peaks are significantly sharper, at the cost of some reduction in peak width (spectrum resolution):

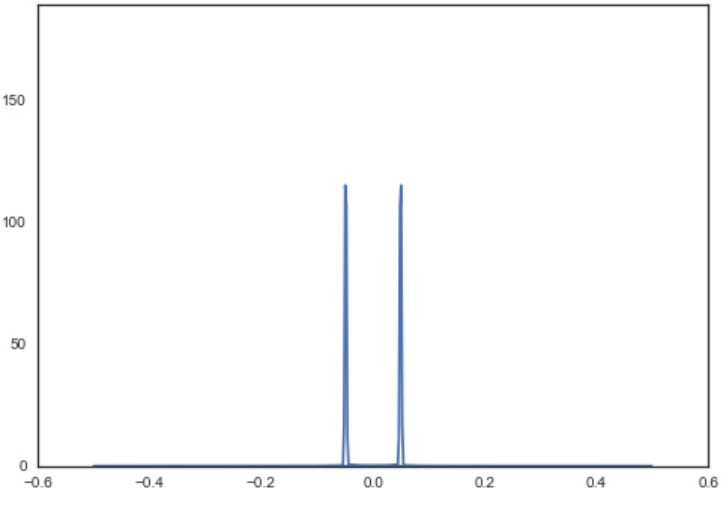
For online notebook, use something like:

```
# @interact(beta=(0, 20.))
# def window(beta):
#     x = np.kaiser(1000, beta)
#     f, axes = plt.subplots(1, 2, figsize=(10, 5))
#     axes[0].plot(x)
#     axes[1].plot(fftpack.fftshift(np.abs(np.fft.fft(x, 10000))))
#     axes[1].set_xlim(2*2480, 2*2520)
#     plt.show()
```

The effect of windowing our previous example is noticeable:

```
In [25]:
```

```
win = np.kaiser(len(t), 5)
X_win = fftpack.fft(x * win)

plt.plot(fftpack.fftfreq(len(t)), np.abs(X_win))
plt.ylim(0, 190)
plt.show()
```



Real-world Application: Analyzing Radar Data

Linearly modulated FMCW (Frequency-Modulated Continuous-Wave) radars make extensive use of the FFT algorithm for signal processing and provide examples of var-

ious application of the FFT. We will use actual data from an FMCW radar to demonstrate one such an application: target detection.

Roughly[1], an FMCW radar works like this:

A signal with changing frequency is generated. This signal is transmitted by antenna, after which it travels outwards, away from the radar. When it hits an object, part of the signal is reflected back to the radar, where it is received, multiplied by a copy of the transmitted signal, and sampled, turning it into numbers that are packed into an array. Our challenge is to interpret those numbers to form meaningful results.

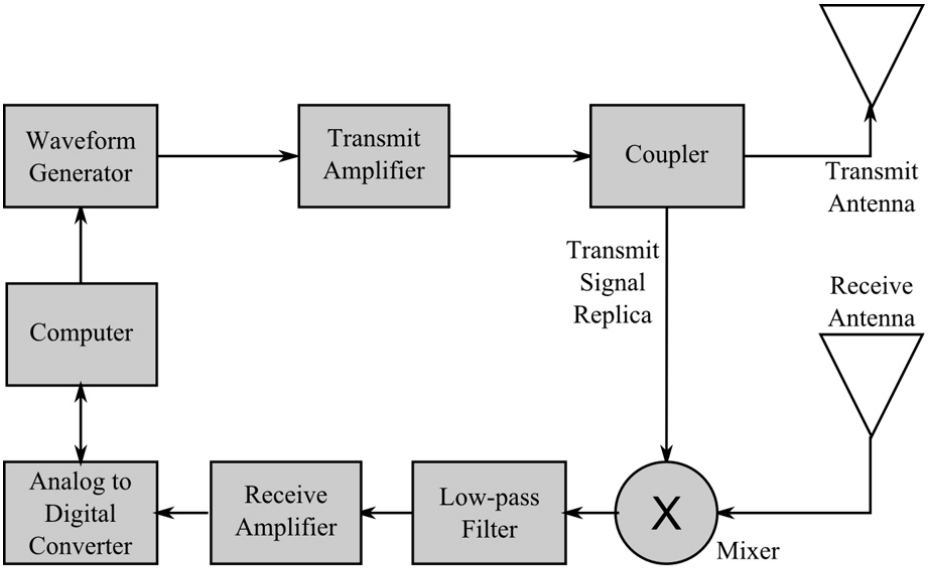
The multiplication step above is important. From school, recall the trigonometric identity: $\sin(xt)\sin(yt) = \frac{1}{2}\left[\sin\left((x-y)t+\frac{\pi}{2}\right) - \sin\left((x+y)t+\frac{\pi}{2}\right)\right]$

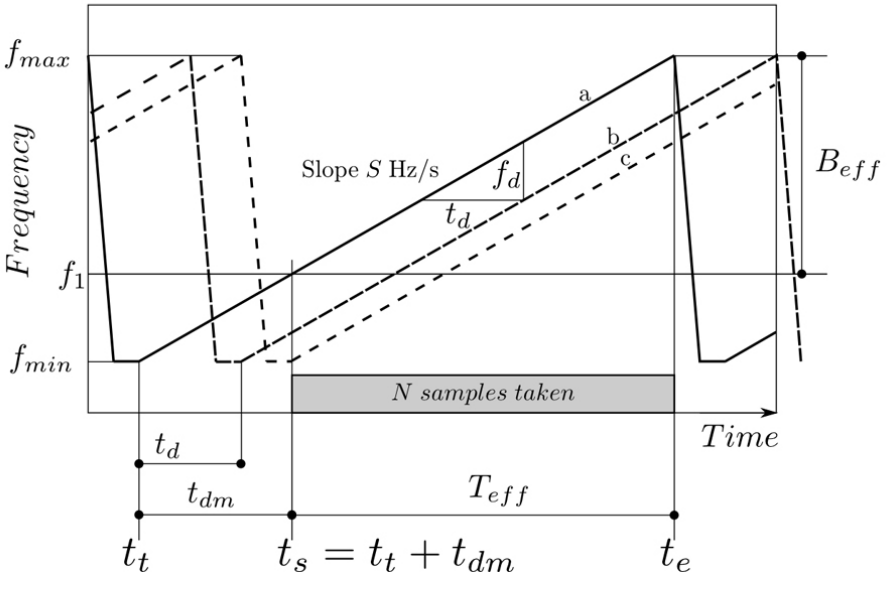
Thus, if we multiply the received signal by the transmitted signal, we expect two frequency components to appear in the spectrum: one that is the difference in frequencies between the received and transmitted signal, and one that is the sum of their frequencies.

We are particularly interested in the first, since that gives us some indication of how long it took the signal to reflect back to the radar (in other words, how far away the object is from us!). We discard the other by applying a low-pass filter to the signal (i.e., a filter that discards any high frequencies).

To summarize, we should note that:

- The data that reaches the computer consists of N samples sampled (from the multiplied, filtered signal) at a sample frequency of f_s.
- The **amplitude** of the returned signal varies depending on the **strength of the reflection** (i.e., a property of the target object).
- The **frequency measured** is an indication of the **distance** of the target object from the radar.





To start off, we'll generate some synthetic signals, after which we'll turn our focus to the output of an actual radar.

Recall that the radar is increasing its frequency as it transmits at a rate of *S* Hz/s. After a certain amount of time, *t*, has passed, the frequency will now be *tS* higher. In that

same time span, the radar signal has traveled $d = t/v$ meters, where v is the speed of the transmitted wave through air (roughly the same as the speed of light, 3×10^8 m/s).

Combining the above observations, we can calculate the amount of time it would change the signal to travel to, bounce off, and return from a target that is distance R away: $ t_R = 2R / v

Therefore, the change in frequency for a target at range R will be:

Equation 4-1. Difference Frequency

$$f_d = t_R S = \frac{2RS}{v}$$

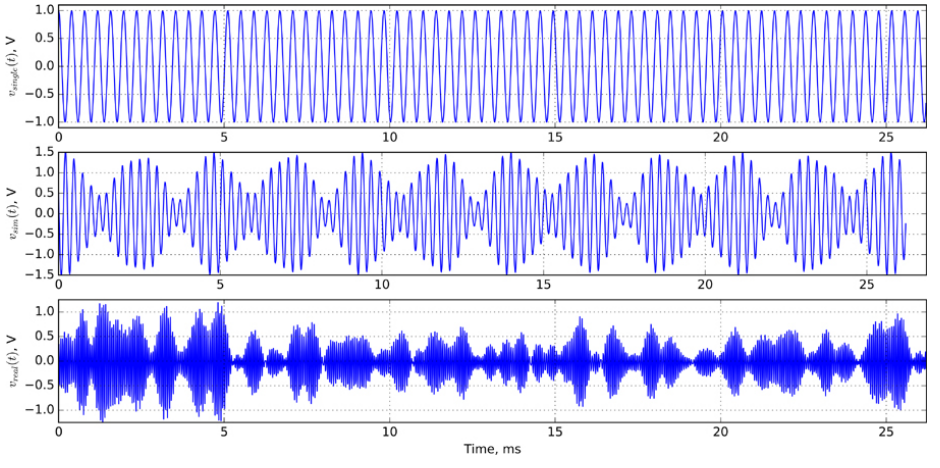
1. A block diagram of a simple FMCW radar that uses separate transmit and receive antennas is shown in Fig. [fig: block-diagram]. The radar consists of a waveform generator that generates a sinusoidal signal of which the frequency varies linearly around the required transmit frequency. The generated signal is amplified to the required power level by the transmit amplifier and routed to the transmit antenna via a coupler circuit where a copy of the transmit signal is tapped off. The transmit antenna radiates the transmit signal as an electromagnetic wave in a narrow beam towards the target to be detected. When the wave encounters an object that reflects electromagnetic waves, a fraction of of the energy irradiating the target is reflected back to the receiver as a second electromagnetic wave that propagates in the direction of the radar system. When this wave encounters the receive antenna, the antenna collects the energy in the wave energy impinging on it and converts it to a fluctuating voltage that is fed to the mixer. The mixer multiplies the received signal with a replica of the transmit signal and produces a sinusoidal signal with a frequency equal to the difference in frequency between the transmitted and received signals. The low-pass filter ensures that the received signal is band limited (i.e., does not contain frequencies that we don't care about) and the receive amplifier strengthens the signal to a suitable amplitude for the analog to digital converter (ADC) that feeds data to the computer.

```
In [26]:

pi = np.pi

# Radar parameters
fs = 78125          # Sampling frequency in Hz, i.e. we sample 78125

                    # times per second

ts = 1 / fs         # Sampling time, i.e. one sample is taken each
                    # ts seconds
```

```
Teff = 2048.0 * ts  # Total sampling time for 2048 samples
                    # (AKA effective sweep duration) in seconds.

Beff = 100e6        # Range of transmit signal frequency during the time the
                    # radar samples, known as the "effective bandwidth"
                    # (given in Hz)

S = Beff / Teff     # Frequency sweep rate in Hz/s

# Specification of targets

R = np.array([100, 137, 154, 159, 180])  # Ranges (in meter)
M = np.array([0.33, 0.2, 0.9, 0.02, 0.1])  # Target size
P = np.array([0, pi / 2, pi / 3, pi / 5, pi / 6])  # Randomly chosen phase offsets

t = np.arange(2048) * ts  # Sample times

fd = 2 * S * R / 3E8      # Frequency differences for these targets

# Generate five targets
signals = np.cos(2 * pi * fd * t[:, np.newaxis] + P)

v_single = signals[:, 0]
v_sim = np.sum(M * signals, axis=1)

## The above code is equivalent to:
#
# v0 = np.cos(2 * pi * fd[0] * t)
# v1 = np.cos(2 * pi * fd[1] * t + pi / 2)
# v2 = np.cos(2 * pi * fd[2] * t + pi / 3)
# v3 = np.cos(2 * pi * fd[3] * t + pi / 5)
# v4 = np.cos(2 * pi * fd[4] * t + pi / 6)
#
## Blend them together
# v_single = v0
# v_sim = (0.33 * v0) + (0.2 * v1) + (0.9 * v2) + (0.02 * v3) + (0.1 * v4)
```

Above, we generate a synthetic signal, v_{single}, received when looking at a single target (see figure below). By counting the number of cycles seen in a given time period, we can compute the frequency of the signal and thus the distance to the target.

A real radar will rarely receive only a single echo, though. The simulated signal v_{sim} shows what a radar signal will look like with five targets at different ranges (including two close to one another at 154 and 159 meter), and $v_{\text{actual}}(t)$ shows the output signal obtained with an actual radar. We cannot interpret these signals in the time domain. They make no sense at all!



The real world radar data is read from a NumPy-format `.npz` file (a light-weight, cross platform and cross-version compatible storage format). These files can be saved with the `np.savez` or `np.savez_compressed` functions. Note that SciPy's `io` submodule can also easily read other formats, such as MATLAB(R) and NetCDF files.

```
In [27]:

data = np.load('data/radar_scan_0.npz')

# Load variable 'scan' from 'radar_scan_0.npz'
scan = data['scan']

# Grab one (azimuth, elevation) measurement
# It has shape (2048,)
v_actual = scan['samples'][5, 14, :]

# The signal amplitude ranges from -2.5V to +2.5V. The 14-bit
# analogue-to-digital converter in the radar gives out integers
# between -8192 to 8192. We convert back to voltage by multiplying by
# $(2.5 / 8192)$.

v_actual = v_actual * (2.5 / 8192)
```

Since `.npz`-files can store multiple variables, we have to select the one we want: `data['scan']`. That returns a *structured NumPy array* with the following fields:

- **time**: unsigned 64-bit (8 byte) integer (`np.uint64`)
- **size**: unsigned 32-bit (4 byte) integer (`np.uint32`)
- **position**:

  — **az**: 32-bit float (`np.float32`)

— **el**: 32-bit float (`np.float32`)

— **region_type**: unsigned 8-bit (1 byte) integer (`np.uint8`)

— **region_ID**: unsigned 16-bit (2 byte) integer (`np.uint16`)

— **gain**: unsigned 8-bit (1 byte) integer (`np.uin8`)

— **samples**: 2048 unsigned 16-bit (2 byte) integers (`np.uint16`)

While it is true that NumPy arrays are *homogeneous* (i.e., all the elements inside are the same), it does not mean that those elements cannot be compound elements, as is the case here.

An individual field is accessed using dictionary syntax:

```
In [28]:

azimuths = scan['position']['az']  # Get all azimuth measurements
```

To construct an array such as the above from scratch, one would first set up the appropriate dtype:

```
In [29]:

dt = np.dtype([('time', np.uint64),
               ('size', np.uint32),
               ('position', [('az', np.float32),
               ('el', np.float32),
               ('region_type', np.uint8),
               ('region_ID', np.uint16)]),
               ('gain', np.uint8),
               ('samples', (np.int16, 2048))])
```

The dtype can then be used to create an array, which we can later fill with values:

```
In [30]:

data = np.zeros(500, dtype=dt)  # Construct array with 500 measurements
```

To summarize what we've seen so far: the shown measurements (v_{sim} and v_{actual}) are the sum of sinusoidal signals reflected by each of several objects. We need to determine each of the constituent components of these composite radar signals. The FFT is the tool that will do this for us.

SIDEBOX: Discrete Fourier transforms

The Discrete Fourier Transform (DFT) converts a sequence of N equally spaced real or complex samples $x_{0,}x_{1,\ldots}x_{N-1}$ of a function $x(t)$ of time (or another variable, depending on the application) into a sequence of N complex numbers X_k by the summation $X_k = \Sigma_{n=0}^{N-1} x_n e^{-j2\pi kn/N},\ k = 0, 1, \ldots N-1$. (Forward DFT)

With the numbers X_k known, the inverse DFT *exactly* recovers the sample values x_n through the summation $x_n = \frac{1}{N}\Sigma_{k=0}^{N-1} X_k e^{j2\pi kn/N}$(Inverse DFT).

Keeping in mind that $e^{j\theta} = \cos\theta + j\sin\theta$, the last equation shows that the DFT has decomposed the sequence x_n into a complex discrete Fourier series with coefficients X_k. Comparing the DFT with a continuous complex Fourier series $x(t) = \Sigma_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t}$ (Complex Fourier series)

the DFT is a *finite* series with N terms defined at the equally spaced discrete instances of the *angle* $(\omega_0 t_n) = 2\pi\frac{k}{N}$ in the interval $[0, 2\pi)$, i.e. *including* 0 and *excluding* 2π. This automatically normalizes the DFT so that time does not appear explicitly in the forward or inverse transform.

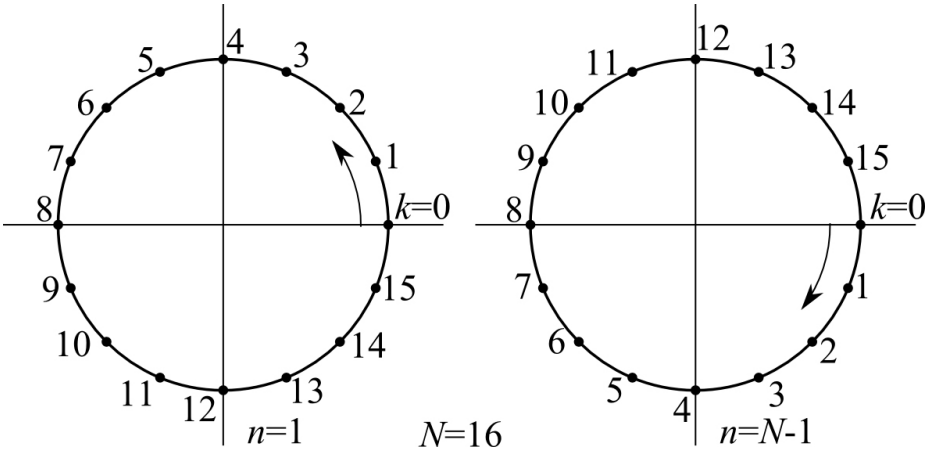
If the original function $x(t)$ is limited in frequency to less than half of the sampling frequency (the so-called *Nyquist frequency*), interpolation between sample values produced by the inverse DFT will usually give a faithful reconstruction of $x(t)$. If $x(t)$ is *not* limited as such, the inverse DFT can, in general, not be used to reconstruct $x(t)$ by interpolation. Note that this limit does not imply that there are *no* methods that can do such a reconstruction—see, e.g., compressed sensing.

The function $e^{j2\pi k/N} = \left(e^{j2\pi/N}\right)^k = w^k$ takes on discrete values between 0 and $2\pi\frac{N-1}{N}$ on the unit circle in the complex plane. The function $e^{j2\pi kn/N} = w^{kn}$ encircles the origin $n\frac{N-1}{N}$ times, thus generating harmonics of the fundamental sinusoid for which $n = 1$.

The way in which we defined the DFT leads to a few subtleties when $n > \frac{N}{2}$. The function $e^{j2\pi kn/N}$ is plotted for increasing values of k in Fig. ([fig:wkn values]) for the cases $n = 1$ and $n = N - 1$ for $N = 16$. When k increases from k to $k + 1$, the angle increases by $\frac{2\pi n}{N}$. When $n = 1$, the step is $\frac{2\pi}{N}$. When $n = N - 1$, the angle increases by $2\pi\frac{N-1}{N} = 2\pi - \frac{2\pi}{N}$. Since $2\pi is$precisely once around the circle, the step equates to $-\frac{2\pi}{N}$, i.e. in the direction of a negative frequency. The components up to $N/2$ represent *positive* frequency components, those above $N/2$ up to $N - 1$ represent *negative* frequencies with frequency. The angle increment for the component $N/2$ for N even advances precisely halfway around the circle for each increment in k and can therefore be interpreted as either a positive or a negative frequency. This component of the DFT represents the Nyquist Frequency, i.e. half of the sampling frequency, and is useful to orientate oneself when looking at DFT graphics.

The FFT in turn is simply a special and highly efficient algorithm for calculating the DFT. Whereas a straightforward calculation of the DFT takes of the order of N^2 calculations to compute, the FFT algorithm requires of the order $N \log N$ calculations.

The FFT was the key to the wide-spread use of the DFT in real-time applications and was included in a list of the top 10 algorithms of the 20th century by the IEEE journal Computing in Science & Engineering in the year 2000.



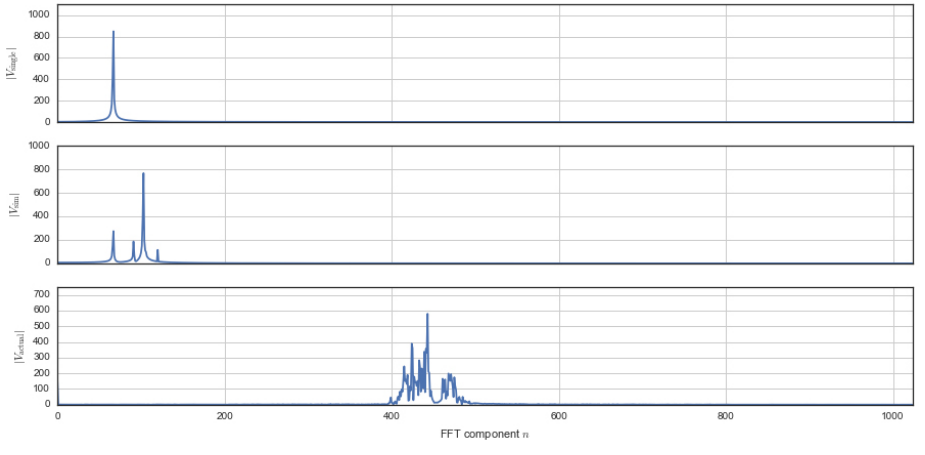
Signal properties in the frequency domain

First, we take the FFTs of our three signals and then display the positive frequency components (i.e., components 0 to $N/2$). These are called the *range traces* in radar terminology.

In [31]:

```
fig, axes = plt.subplots(3, 1, sharex=True, figsize=(15, 7))

# Take FFTs of our signals. Note the convention to name FFTs with a
# capital letter.

V_single = np.fft.fft(v_single)
V_sim = np.fft.fft(v_sim)
V_actual = np.fft.fft(v_actual)

N = len(V_single)

axes[0].plot(np.abs(V_single[:N // 2]))
axes[0].set_ylabel("$|V_\mathrm{single}|$")
axes[0].set_xlim(0, N // 2)
axes[0].set_ylim(0, 1100)

axes[1].plot(np.abs(V_sim[:N // 2]))
axes[1].set_ylabel("$|V_\mathrm{sim} |$")
axes[1].set_ylim(0, 1000)

axes[2].plot(np.abs(V_actual[:N // 2]))
axes[2].set_ylim(0, 750)
```

```
axes[2].set_ylabel("$|V_\mathrm{actual}|$")

axes[2].set_xlabel("FFT component $n$")

for ax in axes:
    ax.grid()

plt.show()
```



Suddenly, the information makes sense!

The plot for $|V_0|$ clearly shows a target at component 67, and for $|V_{\text{sim}}|$ shows the targets that produced the signal that was uninterpretable in the time domain. The real radar signal, $|V_{\text{actual}}|$ shows a large number of targets between component 400 and 500 with a large peak in component 443. This happens to be an echo return from a radar illuminating the high wall of an open-cast mine.

To get useful information from the plot, we must determine the range! Again, we use the formula: $R_n = \frac{nv}{2B_{eff}}$

In radar terminology, each DFT component is known as a *range bin*.

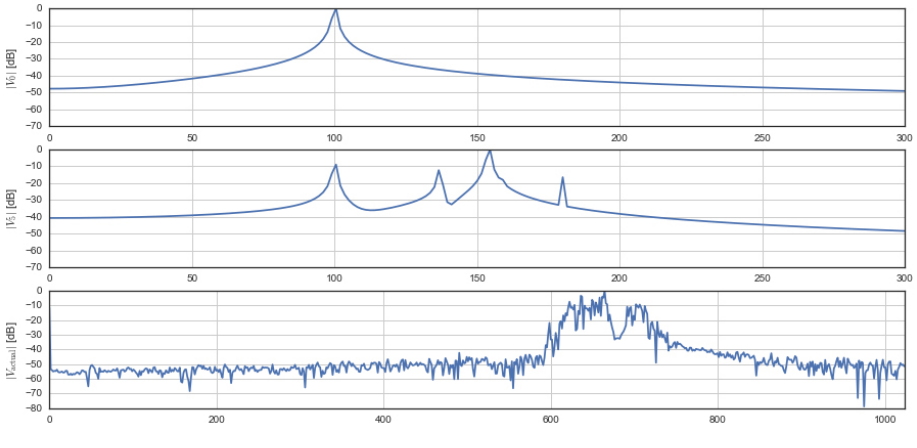
This equation also defines the range resolution of the radar: targets will only be distinguishable if they are separated by more than two range bins, i.e. $\Delta R > \frac{1}{B_{eff}}$.

This is a fundamental property of all types of radar.

This result is quite satisfying—but the dynamic range is so large that we could very easily miss some peaks. Let's take the $\log$ as before with the spectrogram:

```
In [32]:
```

```
c = 3e8  # Approximately the speed of light and of
         # electro-magnetic waves in air

fig, (ax0, ax1,ax2) = plt.subplots(3, 1, figsize=(15, 7))


def dB(y):
    "Calculate the log ratio of y / max(y) in decibel."

    y = np.abs(y)
    y /= y.max()

    return 20 * np.log10(y)


def log_plot_normalized(x, y, ylabel, ax):
    ax.plot(x, dB(y))
    ax.set_ylabel(ylabel)
    ax.grid()


rng = np.arange(N // 2) * c / 2 / Beff

log_plot_normalized(rng, V_single[:N // 2], "$|V_0|$ [dB]", ax0)
log_plot_normalized(rng, V_sim[:N // 2], "$|V_5|$ [dB]", ax1)
log_plot_normalized(rng, V_actual[:N // 2], "$|V_{\mathrm{actual}}|$ [dB]", ax2)

ax0.set_xlim(0, 300)  # Change x limits for these plots so that
ax1.set_xlim(0, 300)  # we are better able to see the shape of the peaks.
ax2.set_xlim(0, len(V_actual) // 2)

plt.show()
```



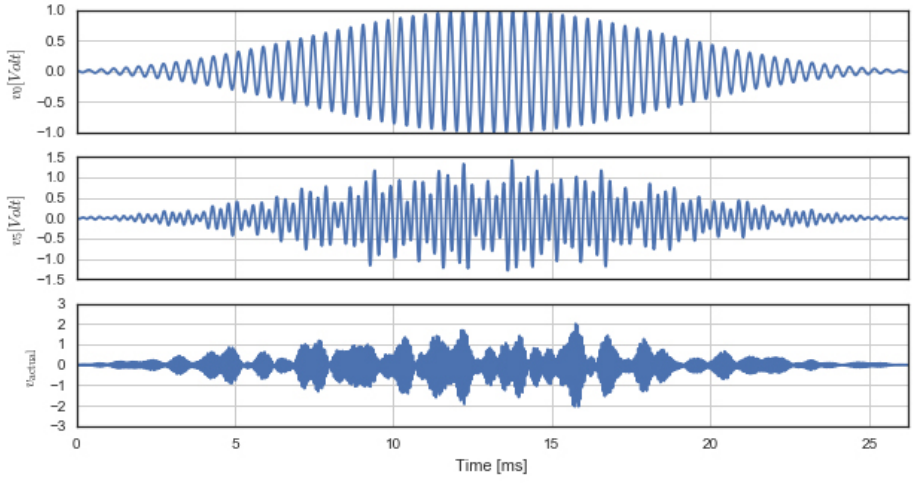
The observable dynamic range is much improved in these plots. For instance, in the real radar signal the *noise floor* of the radar has become visible (i.e., the level where electronic noise in the system starts to limit the radar's ability to detect a target).

Windowing, applied

We're getting there, but in the spectrum of the simulated signal, we still cannot distinguish the peaks at 154 and 159 meters. Who knows what we're missing in the real-world signal! To sharpen the peaks, we'll return to our toolbox and make use of *windowing*.

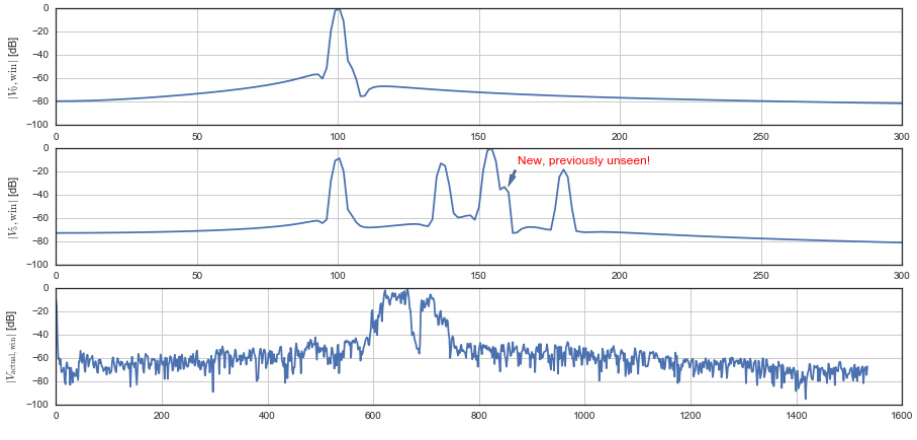
Here are the signals used thus far in this example, windowed with a Kaiser window with $\beta = 6.1$:

```
In [33]:
f, axes = plt.subplots(3, 1, sharex=True, figsize=(10, 5))

t_ms = t * 1000  # Sample times in milli-second

w = np.kaiser(N, 6.1)  # Kaiser window with beta = 6.1

for n, (signal, label) in enumerate([(v_single, r'$v_0 [Volt]$'),
                                     (v_sim, r'$v_5 [Volt]$'),
                                     (v_actual, r'$v_{\mathrm{actual}}$')]):
    axes[n].plot(t_ms, w * signal)
    axes[n].set_ylabel(label)
    axes[n].grid()

axes[2].set_xlim(0, t_ms[-1])
axes[2].set_xlabel('Time [ms]')
plt.show()
```



And the corresponding FFTs (or "range traces", in radar terms):

```
In [34]:

V_single_win = np.fft.fft(w * v_single)
V_sim_win = np.fft.fft(w * v_sim)
V_actual_win = np.fft.fft(w * v_actual)

fig, (ax0, ax1,ax2) = plt.subplots(3, 1, figsize=(15, 7))

log_plot_normalized(rng, V_single_win[:N // 2], r"$|V_0,\mathrm{win}|$ [dB]", ax0)
log_plot_normalized(rng, V_sim_win[:N // 2], r"$|V_5,\mathrm{win}|$ [dB]", ax1)
log_plot_normalized(rng, V_actual_win[:N // 2], r"$|V_\mathrm{actual,win}|$ [dB]", ax2)

ax0.set_xlim(0, 300)  # Change x limits for these plots so that
ax1.set_xlim(0, 300)  # we are better able to see the shape of the peaks.

ax1.annotate("New, previously unseen!", (160, -35),
             xytext=(10, 25), textcoords="offset points", color='red',
             arrowprops=dict(width=2, headwidth=6, headlength=12, shrink=0.1))

plt.show()
```



Compare these with the earlier range traces. There is a dramatic lowering in side lobe level, but at a price: the peaks have changed in shape, widening and becoming less peaky, thus lowering the radar resolution, that is, the ability of the radar to distinguish between two closely space targets. The choice of window is a compromise between side lobe level and resolution. Even so, referring to the trace for V_{sim}, windowing has dramatically increased our ability to distinguish the small target from its large neighbor.

In the real radar data range trace windowing has also reduced the side lobes. This is most visible in the depth of the notch between the two groups of targets.

Radar Images

Knowing how to analyze a single trace, we can expand to looking at radar images.

The data is produced by a radar with a parabolic reflector antenna. It produces a highly directive round pencil beam with a 2° spreading angle between half-power points. When directed with normal incidence at a plane, the radar will illuminate a spot of about 2 m in diameter. Outside this spot the power drops off quite rapidly but strong echoes from outside the spot will nevertheless still be visible.

By varying the pencil beam's azimuth and elevation, we can sweep it across the target area of interest. When reflections are picked up, we can calculate the distance to the reflector (the object hit by the radar signal). Together with the current pencil beam azimuth and elevation, this defines the reflector's position in 3D.

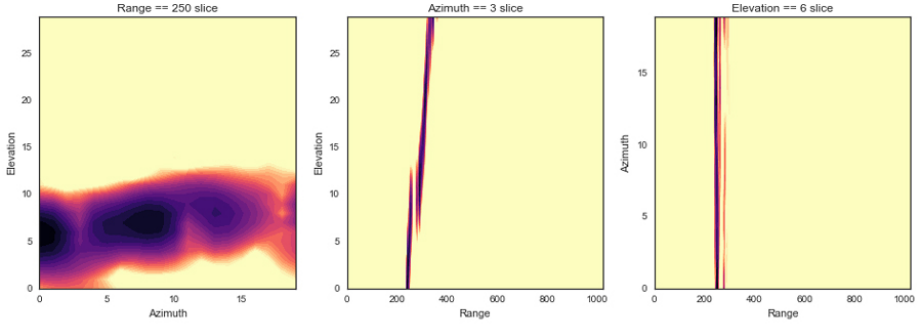
A rock slope consists of thousands of reflectors. A range bin can be thought of as a large sphere with the radar at its center that intersects the slope along a ragged line. The scatterers on this line will produce reflections in this range bin. The reflectors are essentially randomly arranged along the line. The wavelength of the radar (distance the transmitted wave travels in one oscillation second) is about 30 mm. The reflections from scatterers separated by odd multiples of a quarter wavelength in range, about 7.5 mm, will tend to interfere destructively, while those from scatterers separated by multiples of a half wavelength will tend to interfere constructively at the radar. The reflections combine to produce apparent spots of strong reflections. This specific radar moves its antenna in order to scan small regions consisting of 20° azimuth and 30° elevation bins scanned in steps of 0 . 5°.

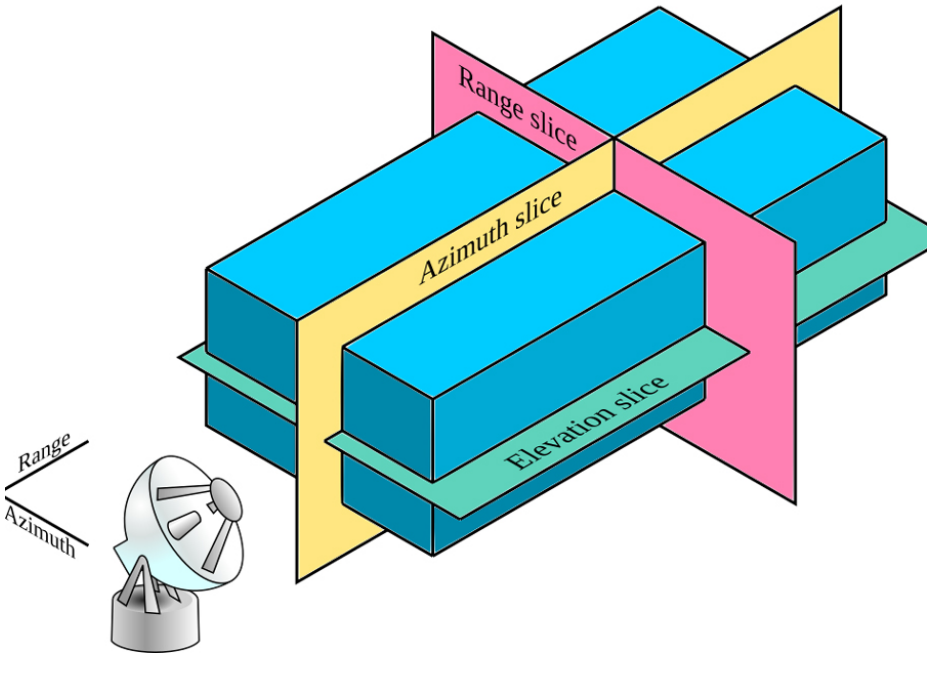
Finally, lets draw some contour plots of the resulting radar data.

```
In [35]:
```

```
data = np.load('data/radar_scan_1.npz')
scan = data['scan']

# The signal amplitude ranges from -2.5V to +2.5V. The 14-bit
# analogue-to-digital converter in the radar gives out integers
# between -8192 to 8192. We convert back to voltage by multiplying by
# $(2.5 / 8192)$.

v = scan['samples'] * 2.5 / 8192
win = np.hanning(N + 1)[:-1]

# Take FFT for each measurement
V = np.fft.fft(v * win, axis=2)[::-1, :, :N // 2]

contours = np.arange(-40, 1, 2)

f, axes = plt.subplots(1, 3, figsize=(16, 5))
```

```
labels = ('Range', 'Azimuth', 'Elevation')

def plot_slice(ax, radar_slice, title, xlabel, ylabel):
    ax.contourf(dB(radar_slice), contours, cmap='magma_r')
    ax.set_title(title)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_axis_bgcolor(plt.cm.magma_r(-40))

plot_slice(axes[0], V[:, :, 250], 'Range == 250 slice', 'Azimuth', 'Elevation')
plot_slice(axes[1], V[:, 3, :], 'Azimuth == 3 slice', 'Range', 'Elevation')
plot_slice(axes[2], V[6, :, :], 'Elevation == 6 slice', 'Range', 'Azimuth')

plt.show()
```





Please refer to the diagram above to see how the different slices are taken. A first slice at fixed range shows the strength of echoes against elevation and azimuth. Another two slices at fixed elevation and azimuth respectively shows the slope. The stepped construction of the high wall in an opencast mine is visible in the azimuth plane.

Further applications of the FFT

The examples above show just one of the uses of the FFT in radar. There are many others, such as movement (Doppler) measurement and target recognition. The Fourier Transform is pervasive, and is seen anywhere from Magnetic Resonance Imaging (MRI) to statistics. With the basic techniques that this chapter outlines in hand, you should be well equipped to use it!

Exercise: The FFT is often used to speed up image convolution (convolution is the application of a moving filter mask). Convolve an image with `np.ones((5, 5))`, using a) numpy's `np.convolve` and b) `np.fft.fft2`. Confirm that the results are identical.

Hints:

- The convolution of `x` and `y` is equivalent to `ifft2(X * Y)`, where `X` and `Y` are the FFTs of x and y respectively.
- In order to multiply `X` and `Y`, they have to be the same size. Use `np.pad` to extend `x` and `y` with zeros (toward the right and bottom) *before* taking their FFT.
- You may see some edge effects. These can be removed by increasing the padding size, so that both `x` and `y` have dimensions `shape(x) + shape(y) - 1`.

CHAPTER 5

Contingency tables using sparse coordinate matrices

```
In [1]:
```

```
%matplotlib inline
import matplotlib.pyplot as plt
# Set up plotting
```

Code by Juan Nunez-Iglesias,

with suggestions by Jaime Frio and Warren Weckesser.

Nominated by Andreas Mueller.

Many real-world matrices are *sparse*, which means that most of their values are zero.

Using numpy arrays to manipulate sparse matrices wastes a lot of time and energy multiplying many, many values by 0. Instead, we can use SciPy's `sparse` module to solve these efficiently, examining only non-zero values. In addition to helping solve these "canonical" sparse matrix problems, `sparse` can be used for problems that are not obviously related to sparse matrices.

One such problem is the comparison of image segmentations. (Review Chapter 3 for a definition of segmentation.)

The code sample motivating this chapter uses sparse matrices twice: once to compute a *contingency matrix* that counts the correspondence of labels between two segmentations, and again to use that contingency matrix to compute the *variation of information*, which measures the differences between segmentations.

```
In [2]:
```

```
def vi(x, y):
    # compute contingency matrix, aka joint probability matrix
```

```
    Pxy = sparse.coo_matrix((np.ones(x.size), (x.ravel(), y.ravel())),
                            dtype=float).tocsr()
    Pxy.data /= np.sum(Pxy.data)

    # compute marginal probabilities, converting to array
    px = Pxy.sum(axis=1).A
    py = Pxy.sum(axis=0).A

    # use sparse matrix linear algebra to compute VI
    Px_inv = sparse.diags(invert_nonzero(px).T, [0])
    Py_inv = sparse.diags(invert_nonzero(py), [0])
    hygx = -px.T @ xlogx(Px_inv @ Pxy).sum(axis=1)
    hxgy = -xlogx(Pxy @ Py_inv).sum(axis=0) @ py.T

    return float(hygx + hxgy)
```

Python 3.5 pro-tip!

The `@` symbols in the above paragraph represent the *matrix multiplication* operator, and were introduced in Python 3.5 in 2015. This is one of the most compelling arguments to use Python 3 for scientific programmers: they enable the programming of linear algebra algorithms using code that remains very close to the original mathematics. Compare the above:

```
hygx = -px.T @ xlogx(Px_inv @ Pxy).sum(axis=1)
```

with the equivalent Python 2 code:

```
hygx = -px.T.dot(xlogx(Px_inv.dot(Pxy)).sum(axis=1))
```

Yuck! By using the `@` operator to stay closer to mathematical notation, we can avoid implementation errors and produce code that is much easier to read.

Actually, SciPy's authors knew this long before the `@` operator was introduced, and actually altered the meaning of the `*` operator when the inputs are SciPy matrices. Available in Python 2.7, it lets us produce nice, readable code like the above:

```
hygx = -px.T * xlog(Px_inv * Pxy).sum(axis=1)
```

But there is a huge catch: this code will behave differently when `px` or `Px_inv` are SciPy matrices than when they are not! If `Px_inv` and `Pxy` are NumPy arrays, `*` produces the element-wise multiplication, while if they are SciPy matrices, it produces the matrix product! As you can imagine, this is the source of a great many errors, and much of the SciPy community has abandoned this use in favor of the uglier but unambiguous `.dot` method.

Python 3.5's `@` operator gives us the best of both worlds!

But let's start simple and work our way up to segmentations.

Suppose you just started working as a data scientist at email startup Spam-o-matic. You are tasked with building a detector for spam email. You encode the detector outcome as a numeric value, 0 for not spam and 1 for spam.

If you have a set of 10 emails to classify, you end up with a vector of *predictions*:

```
In [3]:

import numpy as np
pred = np.array([0, 1, 0, 0, 1, 1, 1, 0, 1, 1])
```

You can check how well you've done by comparing it to a vector of *ground truth*, classifications obtained by inspecting each message by hand.

```
In [4]:

gt = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

Now, classification is hard for computers, so the values in `pred` and `gt` don't match up exactly. At positions where `pred` is 0 and `gt` is 0, the prediction has correctly identified a message as nonspam. This is called a *true negative*. Conversely, at positions where both values are 1, the predictor has correctly identified a spam message, and found a *true positive*.

Then, there are two kinds of errors. If we let a spam message (where `gt` is 1) through to the user's inbox (`pred` is 0), we've made a *false negative* error. If we predict a legitimate message (`gt` is 0) to be spam (`pred` is 1), we've made a *false positive* prediction. (An email from the director of my scientific institute once landed in my spam folder. The reason? His announcement of a postdoc talk competition started with "You could win $500!")

If we want to measure how well we are doing, we have to count the above kinds of errors using a *contingency matrix*. (This is also sometimes called a confusion matrix. The name is apt.) For this, we place the prediction labels along the rows and the ground truth labels along the columns. Then we count the number of times they correspond. So, for example, since there are 4 true positives (where `pred` and `gt` are both 1), the matrix will have a value of 3 at position (1, 1).

Generally:

$$C_{i,j} = \Sigma_k \mathbb{I}(p_k = i)\mathbb{I}(g_k = j)$$

Here's an intuitive, but inefficient way of building the above:

```
In [5]:

def confusion_matrix(pred, gt):
    cont = np.zeros((2, 2))
    for i in [0, 1]:
        for j in [0, 1]:
```

```
            cont[i, j] = np.sum((pred == i) & (gt == j))
    return cont
```

We can check that this gives use the right counts:

```
In [6]:

confusion_matrix(pred, gt)

Out[6]:

array([[ 3.,  1.],
       [ 2.,  4.]])
```

Question: Why did we call this inefficient?

Exercise: Write an alternative way of computing the confusion matrix that only makes a single pass through `pred` and `gt`.

```
In [7]:

def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    # your code goes here
    return cont
```

We can make this example a bit more general: Instead of classifying spam and non-spam, we can classify spam, newsletters, sales and promotions, mailing lists, and personal email. That's 5 categories, which we'll label 0 to 4. The confusion matrix will now be 5-by-5, with matches counted on the diagonal, and errors counted on the off-diagonal entries.

The definition of the `confusion_matrix` function, above, doesn't extend well to this larger matrix, because now we must have *twenty-five* passes though the result and ground truth arrays. This problem only grows as we add more email categories, such as social media notifications.

Exercise: Write a function to compute the confusion matrix in one pass, as above, but instead of assuming two categories, infer the number of categories from the input.

```
In [10]:

def general_confusion_matrix(pred, gt):
    n_classes = None  # replace `None` with something useful
    # your code goes here
    return cont
```

Your one-pass solution will scale well with the number of classes, but, because the for-loop runs in the Python interpreter, it will be slow when you have a large number of documents. Also, because some classes are easier to mistake for one another, the matrix will be *sparse*, with many 0 entries. Indeed, as the number of classes increases, dedicating lots of memory space to the 0 entries of the contingency matrix is increas-

ingly wasteful. Instead, we can use the `sparse` module of SciPy, which contains objects to efficiently represent sparse matrices.

scipy.sparse data formats

We covered the internal data format of NumPy arrays in Chapter 1. I hope you agree that it's a fairly intuitive, and, in some sense, inevitable format to hold n-dimensional array data. For sparse matrices, there are actually a wide array of possible formats, and the "right" format depends on the problem you want to solve.

Perhaps the most intuitive is the coordinate, or COO, format. This uses three 1D arrays to represent a 2D matrix *A*. Each of these arrays has length equal to the number of nonzero values in *A*, and together they list (i, j, value) coordinates of every entry that is not equal to 0.

- the `row` and `col` arrays, which together specify the location of each non-zero entry (row and column indices, respectively).
- the `data` array, which specifies the *value* at each location.

Every part of the matrix that is not represented by the `(row, col)` pairs is considered to be 0. Much more efficient!

So, to represent the matrix:

```
In [12]:

s = np.array([[ 4,  0,  3],
              [ 0, 32, 0]], dtype=float)
```

We can do the following:

```
In [13]:

from scipy import sparse

data = np.array([4, 3, 32], dtype=float)
row = np.array([0, 0, 1])
col = np.array([0, 2, 1])

s_coo = sparse.coo_matrix((data, (row, col)))
```

The `.todense()` method of every sparse format in `scipy.sparse` returns a numpy array representation of the sparse data. We can use this to check that we created `s_coo` correctly:

```
In [14]:

s_coo.todense()

Out[14]:
```

```
matrix([[  4.,   0.,   3.],
        [  0.,  32.,   0.]])
```

Exercise: write out the COO representation of the following matrix:

```
In [15]:
```

```
s2 = np.array([[0, 0, 6, 0, 0],
               [1, 2, 0, 4, 5],
               [0, 1, 0, 0, 0],
               [9, 0, 0, 0, 0],
               [0, 0, 0, 6, 7]])
```

Unfortunately, although the COO format is intuitive, it's not very optimized to use the minimum amount of memory, or to traverse the array as quickly as possible during computations. (Remember from Chapter 1, *data locality* is very important to efficient computation!) However, you can look at your COO representation above to help you identify redundant information: Notice all those repeated `1`s?

If we use COO to enumerate the nonzero entries row-by-row, rather than in arbitrary order (which the format allows), we end up with many consecutive, repeated values in the `row` array. These can be compressed by indicating the *indices* in `col` where the next row starts, rather than repeatedly writing the row index. This is the basis for the *compressed sparse row* or *CSR* format.

Let's work through the example above. In CSR format, the `col` and `data` arrays are unchanged (but `col` is renamed to `indices`). However, the `row` array, instead of indicating the rows, indicates *where* in `col` each row begins, and is renamed to `indptr`, for "index pointer".

So, let's look at `row` and `col` in COO format, ignoring `data`:

```
In [21]:
```

```
row = [0, 1, 1, 1, 1, 2, 3, 4, 4]
col = [2, 0, 1, 3, 4, 1, 0, 3, 4]
```

Each new row begins at the index where `row` changes. The 0th row starts at index 0, and the 1st row starts at index 1, but the 2nd row starts where "2" first appears in `row`, at index 5. Then, the indices increase by 1 for rows 3 and 4, to 6 and 7. The final index, indicating the end of the matrix, is the total number of nonzero values (9). So:

```
In [22]:
```

```
indptr = [0, 1, 5, 6, 7, 9]
```

Let's use these hand-computed arrays to build a CSR matrix in SciPy. We can check our work by comparing the `.todense()` output from our COO and CSR representations to the numpy array `s2` that we defined earlier.

```
In [23]:
```

```
data = np.array([6, 1, 2, 4, 5, 1, 9, 6, 7])

coo = sparse.coo_matrix((data, (row, col)))
csr = sparse.csr_matrix((data, col, indptr))

print('The COO and CSR arrays are equal: ',
      np.all(coo.todense() == csr.todense()))
print('The CSR and NumPy arrays are equal: ',
      np.all(s2 == csr.todense()))
```

```
The COO and CSR arrays are equal:  True
The CSR and NumPy arrays are equal:  True
```

The ability to store large, sparse matrices is incredibly powerful! The combination of sparsity and linear algebra abounds. For example, one can think of the entire web as a large, sparse, $N \times N$ matrix. Each entry X_{ij} indicates whether web page i links to page j. By normalizing this matrix and solving for its dominant eigenvector, one obtains the so-called PageRank—one of the numbers Google uses to order your search results. (You can read more about this in the next chapter!)

Now, consider studying the human brain, and represent it as a large $M \times M$ graph, where there are M nodes (positions) in which you measure activity using an MRI scanner. After a while of measuring, correlations can be calculated and entered into a matrix C_{ij}. The matrix is thresholded (which makes it sparse and fills it with ones and zeros), representing an adjacency matrix. The dominant eigenvector of this matrix is then calculated. The sign of each entry in the (length M) eigenvector groups the nodes into two sub-groups (see Newman (2006), Modularity and community structure in networks, http://www.pnas.org/content/103/23/8577.full). Rinse and repeat to form more and more sub-groups[1]. It turns out that these subgroups, or communities, tell us a lot about functional regions of the brain!

	bsr_matrix	**coo_matrix**	**csc_matrix**	**csr_matrix**	**dia_matrix**	**dok_matrix**	**lil_matrix**
Full name	Block Sparse Row	Coordinate	Compressed Sparse Column	Compressed Sparse Row	Diagonal	Dictionary of Keys	Row-based linked-list
Note	Similar to CSR	Only used to construct sparse matrices, which are then converted to CSC or CSR for further operations.				Used to construct sparse matrices incrementally.	Used to construct sparse matrices incrementally.

	bsr_matrix	coo_matrix	csc_matrix	csr_matrix	dia_matrix	dok_matrix	lil_matrix
Use cases	• Storage of dense sub-matrices • Often used in numerical analyses of discretized problems, • such as finite elements, differential equations	• Fast and straightforward way of constructing sparse matrices • During construction, duplicate coordinates are summed—useful for, e.g., finite element analysis	• Arithmetic operations (supports addition, subtraction, multiplication, division, and matrix power • Efficient column slicing • Fast matrix-vector products (CSR, BSR can be faster, depending on the problem)	• Arithmetic operations • Efficient row slicing • Fast matrix-vector products	• Arithmetic operations	• Changes in sparsity structure are inexpensive • Arithmetic operations • Fast access to individual elements • Efficient conversion to COO (but no duplicates allowed)	• Changes in sparsity structure are inexpensive • Flexible slicing
Cons		• No arithmetic operations • No slicing	• Slow row slicing (see CSR) • Changes to sparsity structure are expensive (see LIL, DOK)	• Slow column slicing (see CSC) • Changes to sparsity structure are expensive (see LIL, DOK)	• Sparsity structure limited to values on diagonals	• Expensive for arithmetic operations • Slow matrix-vector products	• Expensive for arithmetic operations • Slow column slicing • Slow matrix-vector products

Applications of sparse matrices: image transformations

Libraries like scikit-image and SciPy already contain algorithms for transforming (rotating & warping) images effectively, but what if you were head of the NumPy Agency for Space Affairs and had to rotate millions of images streaming in from the newly launched Jupyter Orbiter?

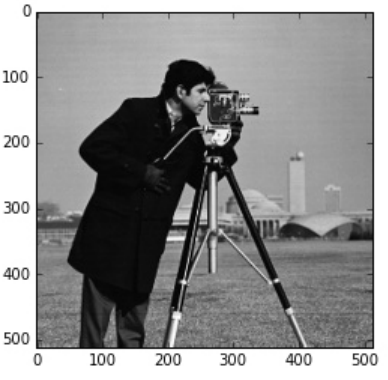
Of course, one option would be to rewrite all your code in C++, but let's presume you have diligently read Scott Meyers' excellent books and are deeply convinced that you will never write bug free C++ under pressure. While options like Cython, Numba, or Julia are available, we'd like to show you a quick workaround using SciPy's sparse matrices.

We'll use the following image as a test:

1. One has to be careful when subdividing the network into more than one group. The graph under analysis remains the original, *not* the previously split result. A correct approach is explained in detail in Newman (2006).

In [24]:

```
from skimage import data
image = data.camera()
plt.imshow(image, cmap='gray');
```



As a test operation, we'll be rotating the image by 30 degrees. We begin by defining the transformation matrix, H which, when multiplied with a coordinate from the input image, $[r, c, 1]$, will give us the corresponding coordinate in the output, $[r', c', 1]$. (Note: we are using homogeneous coordinates, which have a 1 appended to them and which give greater flexibility when defining linear transforms.)

In [25]:

```
angle = 30
c = np.cos(np.deg2rad(angle))
s = np.sin(np.deg2rad(angle))

H = np.array([[c, -s,  0],
              [s,  c,  0],
              [0,  0,  1]])
```

You can verify that this works by multiplying H with the point (1, 0). A 30-degree counterclockwise rotation around the origin (0, 0) should take us to point $\left(\frac{\sqrt{3}}{2}, \frac{1}{2}\right)$:

In [26]:

```
point = np.array([1, 0, 1])
print(np.sqrt(3) / 2)
print(H @ point)
```

```
0.866025403784
[ 0.8660254  0.5        1.         ]
```

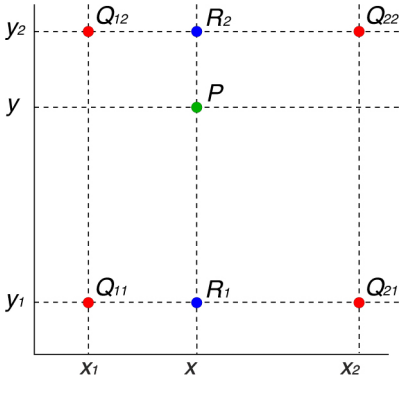
Similarly, applying the 30-degree rotation three times should get us to the column axis, at point (0, 1). We can see that this works, minus some floating point approximation error:

```
In [27]:

print(H @ H @ H @ point)

[  2.77555756e-16   1.00000000e+00   1.00000000e+00]
```

Now, we will build a function that defines a "sparse operator". The goal of the sparse operator is to take all pixels of the output image, figure out where they came from in the input image and, doing the appropriate (bi-linear) interpolation, calculate their values. It does this using just matrix multiplication on the image values, and thus is extremely fast.

It will be used as follows:

```
## Turn input image into a vector
input_flat = input_image.ravel()

## Multiply input image by sparse operator
out_flat = input_flat @ Sparse_operator

## Reshape output vector back into an image of the same shape as
the input
out_image = out_flat.reshape(input_flat.shape)
```



Let's look at the function that builds our sparse operator:

```
In [28]:

from itertools import product

def homography(tf, image_shape):
```

```
    """Represent homographic transformation & interpolation as linear operator.

    Parameters
    ----------
    tf : (3, 3) ndarray
        Transformation matrix.
    image_shape : (M, N)
        Shape of input gray image.

    Returns
    -------
    A : (M * N, M * N) sparse matrix
        Linear-operator representing transformation + bilinear interpolation.

    """
    # Invert matrix. This tells us, for each output pixel, where to
    # find its corresponding input pixel.
    H = np.linalg.inv(tf)

    m, n = image_shape

    # We are going to construct a COO matrix, for which we'll need I
    # (row coordinates), col (column coordinates), and K (values)
    row, col, values = [], [], []

    # For each pixel in the output image...
    for sparse_op_row, (out_row, out_col) in \
            enumerate(product(range(m), range(n))):

        # Compute where it came from in the input image
        in_row, in_col, in_abs = H @ [out_row, out_col, 1]
        in_row /= in_abs
        in_col /= in_abs

        # if the coordinates are outside of the original image, ignore this
        # coordinate; we will have 0 at this position
        if (not 0 <= in_row < m - 1 or
                not 0 <= in_col < n - 1):
            continue

        # We want to find the four surrounding pixels, so that we
        # can interpolate their values to find an accurate
        # estimation of the output pixel value
        # We start with the top, left corner, noting that the remaining
        # points are 1 away in each direction.
        top = int(np.floor(in_row))
        left = int(np.floor(in_col))

        # Calculate the position of the output pixel, mapped into
        # the input image, within the four selected pixels
        # https://commons.wikimedia.org/wiki/File:BilinearInterpolation.svg
        t = in_row - top
```

```
            u = in_col - left

            # The current row of the sparse operator matrix is given by the
            # raveled output pixel coordinates, contained in `sparse_op_row`.
            # We will take the weighted average of the four surrounding input
            # pixels, corresponding to four columns. So we need to repeat the row
            # index four times.
            row.extend([sparse_op_row] * 4)

            # The actual weights are calculated according to the bilinear
            # interpolation algorithm, as shown at
            # https://en.wikipedia.org/wiki/Bilinear_interpolation
            sparse_op_col = np.ravel_multi_index(
                    ([top, top, top + 1, top + 1 ],
                     [left, left + 1, left, left + 1]), dims=(m, n))
            col.extend(sparse_op_col)
            values.extend([(1-t) * (1-u), (1-t) * u, t * (1-u), t * u])

        operator = sparse.coo_matrix((values, (row, col)),
                                     shape=(m*n, m*n)).tocsr()
        return operator
```

Recall that we apply the sparse operator as follows:

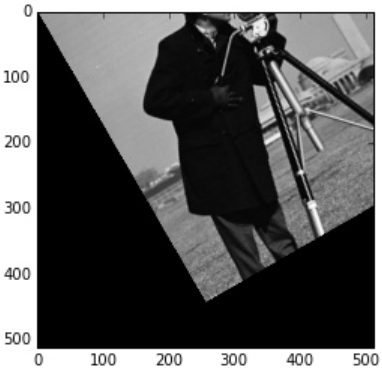
```
In [29]:
```

```
def apply_transform(image, tf):
    return (tf @ image.flat).reshape(image.shape)
```

Let's try it out!

```
In [30]:
```

```
tf = homography(H, image.shape)
out = apply_transform(image, tf)
plt.imshow(out, cmap='gray');
```



There's that rotation!

Exercise: The rotation happens around the origin, coordinate (0, 0). But can you rotate the image around its center?

Hint: The transformation matrix for a *translation*, i.e. sliding the image up/down or left/right, is given by:

$$H_{tr} = \begin{bmatrix} 1 & 0 & t_r \\ 0 & 1 & t_c \\ 0 & 0 & 1 \end{bmatrix}$$

when you want to move the image t_r pixels down and t_c pixels right.

As mentioned above, this sparse linear operator approach to image transformation is extremely fast. Let's measure how it performs in comparison to ndimage. To make the comparison fair, we need to tell ndimage that we want linear interpolation with `order=1`, and that we want to ignore pixels outside of the original shape, with `reshape=False`.

```
In [33]:

%timeit apply_transform(image, tf)

100 loops, best of 3: 3.53 ms per loop

In [34]:

from scipy import ndimage as ndi
%timeit ndi.rotate(image, 30, reshape=False, order=1)

10 loops, best of 3: 19.8 ms per loop
```

On our machines, we see a speed-up of approximately 10 times. While this example does only a rotation, there is no reason why we cannot do more complicated warping operations, such as correcting for a distorted lens during imaging, or making people pull funny faces. Once the transform has been computed, applying it repeatedly is extremely fast, thanks to sparse matrix algebra.

So now that we've seen a "standard" use of SciPy's sparse matrices, let's have a look at the out-of-the-box use that inspired this chapter!

Back to contingency matrices

You might recall that we are trying to quickly build a sparse, joint probability matrix using SciPy's sparse formats. We know that the COO format stores sparse data as three arrays, containing the row and column coordinates of nonzero entries, as well as their values. But we can use a little known feature of COO to obtain our matrix extremely quickly.

Have a look at this data:

```
In [35]:

row = [0, 0, 2]
col = [1, 1, 2]
dat = [5, 7, 1]
S = sparse.coo_matrix((dat, (row, col)))
```

Notice that the entry at (row, column) position (0, 1) appears twice: first as 5, and then at 7. What should the matrix value at (0, 1) be? Cases could be made for both the earliest entry encountered, or the latest, but what was in fact chosen is the *sum*:

```
In [36]:

print(S.todense())

[[ 0 12  0]
 [ 0  0  0]
 [ 0  0  1]]
```

So, COO format will sum together repeated entries... Which is exactly what we need to do to make a contingency matrix! Indeed, our task is pretty much done: we can set `pred` as the rows, `gt` as the columns, and simply 1 as the values. The ones will get summed together and count the number of times that label *i* in `pred` occurs together with label *j* in `gt` at position *i*, *j* in the matrix! Let's try it out:

```
In [37]:

from scipy import sparse

def confusion_matrix(pred, gt):
    cont = sparse.coo_matrix((np.ones(pred.size), (pred, gt)))
    return cont
```

To look at a small one, we simply use the `.todense()` method, which returns the numpy array corresponding to that matrix:

```
In [38]:

cont = confusion_matrix(pred, gt)
print(cont)

  (0, 0) 1.0
  (1, 0) 1.0
  (0, 0) 1.0
  (0, 0) 1.0
  (1, 0) 1.0
  (1, 1) 1.0
  (1, 1) 1.0
  (0, 1) 1.0
  (1, 1) 1.0
  (1, 1) 1.0

In [39]:
```

```
print(cont.todense())
```

```
[[ 3.  1.]
 [ 2.  4.]]
```

It works!

Exercise: Remember from Chapter 1 that NumPy has built-in tools for repeating arrays using *broadcasting*. How can you reduce the memory footprint required for the contingency matrix computation?

Hint: Look at the documentation for the function `np.broadcast_to`.

Contingency matrices in segmentation

You can think of the segmentation of an image in the same way as the classification problem above: The segment label at each *pixel* is a *prediction* about which *class* the pixel belongs to. And numpy arrays allow us to do this transparently, because their `.ravel()` method returns a 1D view of the underlying data.

As an example, here's a segmentation of a tiny 3 by 3 image:

```
In [42]:
```

```
seg = np.array([[1, 1, 2],
                [1, 2, 2],
                [3, 3, 3]], dtype=int)
```

Here's the ground truth, what some person said was the correct way to segment this image:

```
In [43]:
```

```
gt = np.array([[1, 1, 1],
               [1, 1, 1],
               [2, 2, 2]], dtype=int)
```

We can think of these two as classifications, just like before. Every pixel is a different prediction.

```
In [44]:
```

```
print(seg.ravel())
print(gt.ravel())
```

```
[1 1 2 1 2 2 3 3 3]
[1 1 1 1 1 1 2 2 2]
```

Then, like above, the contingency matrix is given by:

```
In [45]:
```

```
cont = sparse.coo_matrix((np.ones(seg.size),
                          (seg.ravel(), gt.ravel())))
print(cont)
```

```
  (1, 1) 1.0
  (1, 1) 1.0
  (2, 1) 1.0
  (1, 1) 1.0
  (2, 1) 1.0
  (2, 1) 1.0
  (3, 2) 1.0
  (3, 2) 1.0
  (3, 2) 1.0
```

Some indices appear more than once, but we can use the summing feature of the COO format to confirm that this represents the matrix we want:

```
In [46]:

print(cont.todense())

  [[ 0.  0.  0.]
   [ 0.  3.  0.]
   [ 0.  3.  0.]
   [ 0.  0.  3.]]
```

Segmentation is a hard problem, so it's important to measure how well a segmentation algorithm is doing, by comparing its output to a "ground truth" segmentation that is manually produced by a human.

But, even this comparison is not an easy task. How do we define how "close" an automated segmentation is to a ground truth? We'll illustrate one method, the *variation of information* or VI (Meila, 2005). This is defined as the answer to the following question: on average, for a random pixel, if we are given its segment ID in one segmentation, how much more *information* do we need to determine its ID in the other segmentation?

In order to answer this question, we'll need a quick primer on information theory. We need to be brief but if you want more information (heh), you should look at Christopher Olah's stellar blog post, Visual Information Theory.

The basic unit of information is the *bit*, commonly shown as a 0 or 1, representing choice between two options. This is straightforward: if I want to tell you whether a coin toss landed as heads or tails, I need one bit, which can take many forms: a long or short pulse over a telegraph wire (as in Morse code), a light flashing one of two colors, or a single number taking values 0 or 1. Importantly, I *always* need one bit, because the outcome of a coin toss is random.

It turns out that we can extend this concept to *fractional* bits for events that are *less* random. Suppose, for example, that you need to transmit whether it rained today in Los Angeles. At first glance, it seems that this requires 1 bit as well: 0 for it didn't rain, 1 for it rained. However, rain in LA is a rare event, so over time we can actually get away with transmitting much less information: Transmit a 0 *occasionally* just to make

sure that our communication is still working, but otherwise simply *assume* that the signal is 0, and send 1 only on those rare occasions that it rains.

Thus, when two events are *not* equally likely, we need *less* than 1 bit to represent them. Generally, we measure this for any random variable X (which could have more than two possible values) by using the *entropy* function H:

$$H(X) = \Sigma_x p_x \log_2 \left(\frac{1}{p_x}\right)$$

where the xs are possible values of X, and p_x is the probability of X taking value x.

So, the entropy of a coin toss T that can take values heads (h) and tails (t) is:

$$H(T) = p_h \log_2 \left(1/p_h\right) + p_t \log_2 \left(1/p_t\right) = 1/2 log_2(2) + 1/2 \log_2 (2) = 1/2 \cdot 1 + 1/2 \cdot 1 = 1$$

The long-term probability of rain on any given day in LA is about 1 in 6, so the entropy of rain in LA, R, taking values rain (r) or shine (s) is:

$$H(R) = p_r \log_2 \left(1/p_r\right) + p_s \log_2 \left(1/p_s\right) = 1/6 \log_2 (6) + 5/6 \log_2 (6/5) \approx 0.65 \text{bits}$$

A special kind of entropy is the *conditional* entropy. This is the entropy of a variable *assuming* that you also know something else about that variable. For example: what is the entropy of rain *given* that you know the month? This is written as:

$$H(R|M) = \Sigma_{m = 1 \ldots 12} p(m) H(R|M = m)$$

and

$$H(R|M = m) = p_{r|m} \log_2 \left(\frac{1}{p_{r|m}}\right) + p_{s|m} \log_2 \left(\frac{1}{p_{s|m}}\right)$$
$$= \frac{p_{rm}}{p_m} \log_2 \left(\frac{p_m}{p_{rm}}\right) + \frac{p_{sm}}{p_m} \log_2 \left(\frac{p_m}{p_{sm}}\right)$$

You now have all the information theory you need to understand the variation of information. In the above example, events are days, and they have two properties:

- rain/shine
- month

By observing many days, we can build a *contingency matrix*, just like the ones in the classification examples, measuring the month of a day and whether it rained. We're not going to travel to LA to do this (fun though it would be), and instead we use the historical table below, roughly eyeballed from WeatherSpark:

Month	P(rain)	P(shine)
1	0.25	0.75
2	0.27	0.73
3	0.24	0.76
4	0.18	0.82
5	0.14	0.86
6	0.11	0.89
7	0.07	0.93
8	0.08	0.92
9	0.10	0.90
10	0.15	0.85
11	0.18	0.82
12	0.23	0.77

The conditional entropy of rain given month is then:

$$H(R|M) = \frac{1}{12}\left(0.25 \log_2 (1/0.25) + 0.75 \log_2 (1/0.75)\right) + \frac{1}{12}\left(0.27 \log_2 (1/0.27) + 0.73 \log_2 (1/0.73)\right) + \ldots + \frac{1}{12}\left(0.23 \log_2 (1/0.23) + 0.77 \log_2 (1/0.77)\right) \approx 0.626 \text{bits}$$

So, by using the month, we've reduced the randomness of the signal, but not by much!

We can also compute the conditional entropy of month given rain, which measures how much information we need to determine the month if we know it rained. Intuitively, we know that this is better than going in blind, since it's more likely to rain in the winter months.

Exercise: Compute the conditional entropy of month given rain. What is the entropy of the month variable? (Ignore the different number of days in a month.) Which one is greater? (*Hint:* the probabilities in the table are the conditional probabilities of rain given month.)

```
In [47]:
```

```
prains = [25, 27, 24, 18, 14, 11, 7, 8, 10, 15, 18, 23]
prains = [p / 100 for p in prains]
```

```
pshine = [1 - p for p in prains]
p_rain_g_month = np.array((prains, pshine)).T
# replace 'None' below with expression for non-conditional contingency
# table. Hint: the values in the table must sum to 1.
p_rain_month = None
# Add your code below to compute H(M|R) and H(M)
```

Together, these two values define the variation of information (VI):

$$VI(A,B) = H(A\,|\,B) + H(B\,|\,A)$$

Back in the image segmentation context, "days" become "pixels", and "rain" and "month" become "label in automated segmentation (AS)" and "label ground truth (GT)". Then, the conditional entropy of the automatic segmentation given the ground truth measures how much additional information we need to determine a pixel's identity in AS if we are told its identity in GT. For example, if every GT segment g is split into two equally-sized segments a_1 and a_2 in AS, then $H(AS\,|\,GT) = 1$, because after knowing a pixel is in g, you still need 1 additional bit to know whether it belongs to a_1 or a_2. However, H(GT|AS) = 0, because regardless of whether a pixel is in a_1 or a_2, it is guaranteed to be in g, so you need no more information than the segment in AS.

So, together, in this case,

$$VI(AS,GT) = H(AS\,|\,GT) + H(GT\,|\,AS) = 1 + 0 = 1\text{bit}\,.$$

Here's a simple example:

```
In [51]:

aseg = np.array([[0, 1],
                 [2, 3]], int)

gt = np.array([[0, 1],
               [0, 1]], int)
```

Here we have two segmentations of a four-pixel image: `as` and `gt`. `as` puts every pixel in its own segment, while `gt` puts the left two pixels in segment 0 and the right two pixels in segment 1. Now, we make a contingency table of the pixel labels, just as we did with the spam prediction labels. The only difference is that the label arrays are 2-dimensional, instead of the 1D arrays of predictions. In fact, this doesn't matter: remember that numpy arrays are actually linear (1D) chunks of data with some shape and other metadata attached. We can ignore the shape by using the arrays' `.ravel()` method:

```
In [52]:

aseg.ravel()
```

```
Out[52]:

array([0, 1, 2, 3])
```

Now we can just make the contingency table in the same way as when we were predicting spam:

```
In [53]:

cont = sparse.coo_matrix((np.broadcast_to(1., aseg.size),
                          (aseg.ravel(), gt.ravel())))
cont = np.asarray(cont.todense())
cont

Out[53]:

array([[ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  0.],
       [ 0.,  1.]])
```

In order to make this a table of probabilities, instead of counts, we simply divide by the total number of pixels:

```
In [54]:

cont /= np.sum(cont)
```

Finally, we can use this table to compute the probabilities of labels in *either* `aseg` or `gt`, using the axis-wise sums:

```
In [55]:

p_as = np.sum(cont, axis=1)
p_gt = np.sum(cont, axis=0)
```

There is a small kink in writing Python code to compute entropy: although 0 log (0) is defined to be equal to 0, in Python, it is undefined, and results in a `nan` (not a number) value:

```
In [56]:

print('The log of 0 is: ', np.log2(0))
print('0 times the log of 0 is: ', 0 * np.log2(0))

The log of 0 is:  -inf
0 times the log of 0 is: nan

/Users/jni/conda/envs/elegant/lib/python3.5/site-packages/ipykernel/__main__.py:1: RuntimeWarning:
by zero encountered in log2
  if __name__ == '__main__':
/Users/jni/conda/envs/elegant/lib/python3.5/site-packages/ipykernel/__main__.py:2: RuntimeWarning:
by zero encountered in log2
  from ipykernel import kernelapp as app
/Users/jni/conda/envs/elegant/lib/python3.5/site-packages/ipykernel/__main__.py:2: RuntimeWarning:
```

```
value encountered in double_scalars
  from ipykernel import kernelapp as app
```

Therefore, we have to use numpy indexing to mask out the 0 values. Additionally, we'll need a slightly different strategy depending on whether the input is a numpy array or a SciPy sparse matrix. We'll write the following convenience function:

```
In [57]:
```

```
def xlog1x(arr_or_mat):
    """Compute the element-wise entropy function of an array or matrix.

    Parameters
    ----------
    arr_or_mat : numpy array or scipy sparse matrix
        The input array of probabilities. Only sparse matrix formats with a
        `data` attribute are supported.

    Returns
    -------
    out : array or sparse matrix, same type as input
        The resulting array. Zero entries in the input remain as zero,
        all other entries are multiplied by the log (base 2) of their
        inverse.
    """
    out = arr_or_mat.copy()
    if isinstance(out, sparse.spmatrix):
        arr = out.data
    else:
        arr = out
    nz = np.nonzero(arr)
    arr[nz] *= np.log2(1/arr[nz])
    return out
```

Let's make sure it works:

```
In [58]:
```

```
a = np.array([0.25, 0.25, 0, 0.25, 0.25])
xlog1x(a)
```

```
Out[58]:
```

```
array([ 0.5, 0.5, 0. , 0.5, 0.5])
```

```
In [59]:
```

```
mat = sparse.csr_matrix([[0.125, 0.125, 0.25,    0],
                         [0.125, 0.125,    0, 0.25]])
xlog1x(mat).todense()
```

```
Out[59]:
```

```
matrix([[ 0.375,  0.375,  0.5  ,  0.   ],
        [ 0.375,  0.375,  0.   ,  0.5  ]])
```

So, the conditional entropy of AS given GT:

```
In [60]:

H_ag = np.sum(np.sum(xlog1x(cont / p_gt), axis=0) * p_gt)
H_ag

Out[60]:
1.0
```

And the converse:

```
In [61]:

H_ga = np.sum(np.sum(xlog1x(cont / p_as[:, np.newaxis]), axis=1) * p_as)
H_ga

Out[61]:

0.0
```

We used numpy arrays and broadcasting in the above examples, which, as we've seen many times, is a powerful way to analyze data in Python. However, for segmentations of complex images, possibly containing thousands of segments, it rapidly becomes inefficient. We can instead use `sparse` throughout the calculation, and recast some of the NumPy magic as linear algebra operations. This was suggested to me by Warren Weckesser on StackOverflow.

The linear algebra version efficiently computes a contingency matrix for very large amounts of data, up to billions of points, and is elegantly concise.

```
In [62]:

import numpy as np
from scipy import sparse


def diag(arr):
    """Return a diagonal square matrix with `arr` as its nonzero elements.

    Parameters
    ----------
    arr : array
        The input array.

    Returns
    -------
    D : the output matrix
    """


def invert_nonzero(arr):
    arr_inv = arr.copy()
    nz = np.nonzero(arr)
```

```
    arr_inv[nz] = 1 / arr[nz]
    return arr_inv


def vi(x, y):

    # Compute the joint probability matrix
    ones = np.broadcast_to(1., x.size)
    pxy = sparse.coo_matrix((ones, (x.ravel(), y.ravel())),
                            dtype=float).tocsr()
    pxy.data /= np.sum(pxy.data)

    # Compute the marginals
    # Use .A.ravel() to make them 1D arrays instead of flat matrices
    px = pxy.sum(axis=1).A.ravel()
    py = pxy.sum(axis=0).A.ravel()

    # Compute the inverse diagonal matrices, which will help
    # compute the conditional probabilities
    px_inv = sparse.diags(invert_nonzero(px))
    py_inv = sparse.diags(invert_nonzero(py))

    # Finally, compute the entropies
    hygx = px @ xlog1x(px_inv @ pxy).sum(axis=1)
    hxgy = xlog1x(pxy @ py_inv).sum(axis=0) @ py

    # return their sum
    return float(hygx + hxgy)
```

We can check that this gives the right value (1) for the VI of our toy `aseg` and `gt`:

```
In [63]:

vi(aseg, gt)

Out[63]:

1.0
```

You can see how we use three types of sparse matrices (COO, CSR, and diagonal) to efficiently solve the entropy calculation in the case of sparse contingency matrices, where NumPy would be inefficient. (Indeed, this whole approach was inspired by a Python `MemoryError`!)

To finish, let's demonstrate the use of VI to estimate the best possible automated segmentation of an image. You may remember our friendly stalking tiger from Chapter 3. (If you don't, you might want to work on your threat-assessment skills!) Using our skills from Chapter 3, we're going to generate a number of possible ways of segmenting the tiger image, and then figure out the best one.

[Ed note: Tiger image and segmentation licensed for "non-commercial research and educational purposes"?. May need to ask permission to use in the book. See: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/]

S: Note to self: the outline image can also be read from the archive at

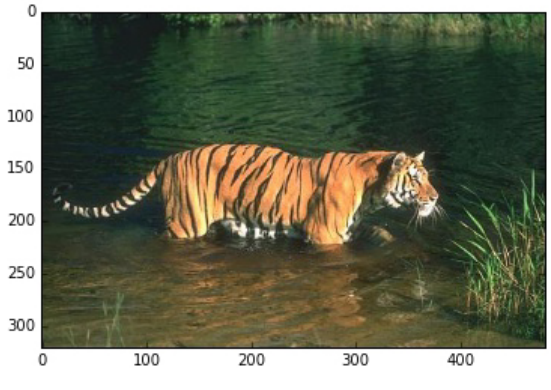
https://github.com/BIDS/BSDS500/blob/master/BSDS500/data/groundTruth/train/108073.mat

and then loaded using

```
from scipy import io
f = io.loadmat('108073.mat')
outline = f['groundTruth'][0, 3]['Boundaries'][0, 0]
# ^ 3 refers to third segmentation, the one previously used in this
example

# I can also convert all the .mat files in the archive to .png files,
# if needed
```

```
In [64]:
```

```
from skimage import io
from matplotlib import pyplot as plt

url =
 'http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300/html/images/plain/normal/c
tiger = io.imread(url)
plt.imshow(tiger);
```

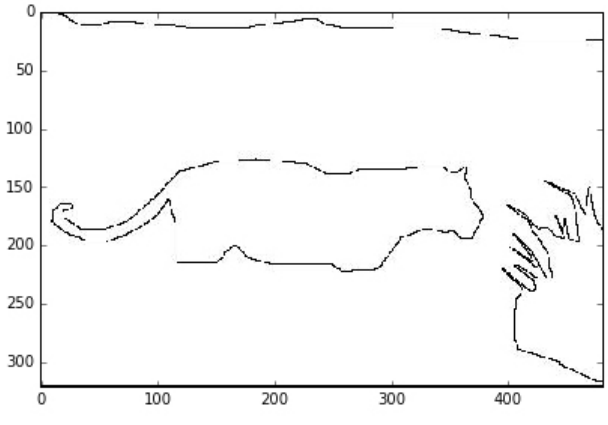


In order to check our image segmentation, we're going to need a ground truth. It turns out that humans are awesome at detecting tigers (natural selection for the win!), so all we need to do is ask a human to find the tiger. Luckily, researchers at Berkeley have already asked dozens of humans to look at this image and manually segment it. Let's grab one of the segmentation images from the Berkeley Segmentation Dataset and Benchmark. It's worth noting that there is quite substantial variation between the

segmentations performed by humans. If you look through the various tiger segmentations, you will see that some humans are more pedantic than others about tracing around the reeds, while others consider the reflections to be objects worth segmenting out from the rest of the water. We have chosen a segmentation that we like (one with pedantic-reed-tracing, because we are perfectionistic scientist-types.) But to be clear, we really have no single ground truth!

In [65]:

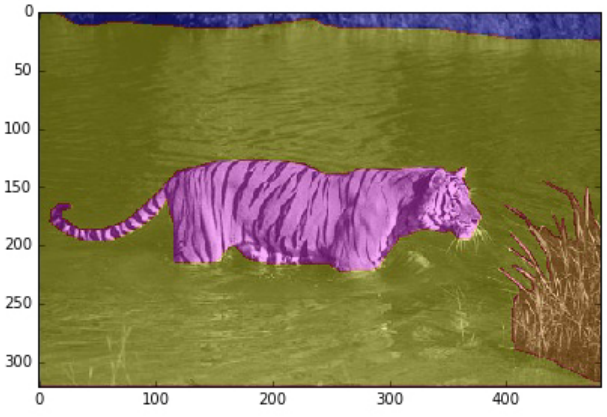
```
from scipy import ndimage as nd
from skimage import color

human_seg_url =
'http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300/html/images/human/normal/ou
boundaries = io.imread(human_seg_url)
io.imshow(boundaries);
```



Overlaying the tiger image with the human segmentation, we can see that (unsurprisingly) this person does a pretty good job of finding the tiger. They have also segmented out the river bank, and a tuft of reeds. Nice job, human #1122!

In [66]:

```
human_seg = nd.label(boundaries > 100)[0]
io.imshow(color.label2rgb(human_seg, tiger));
```



Now, let's grab our image segmentation code from Chapter 3, and see how well a Python does a recognizing a tiger!

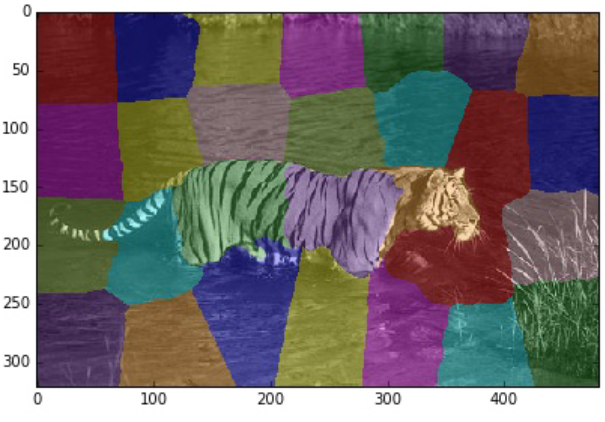
In [67]:

```
# Draw a region adjacency graph (RAG) - all code from Ch3
import networkx as nx
import numpy as np
from scipy import ndimage as nd
from skimage.future import graph

def add_edge_filter(values, graph):
    current = values[0]
    neighbors = values[1:]
    for neighbor in neighbors:
        graph.add_edge(current, neighbor)
    return 0. # generic_filter requires a return value, which we ignore!

def build_rag(labels, image):
    g = nx.Graph()
    footprint = nd.generate_binary_structure(labels.ndim, connectivity=1)
    for j in range(labels.ndim):
        fp = np.swapaxes(footprint, j, 0)
        fp[0, ...] = 0 # zero out top of footprint on each axis
    _ = nd.generic_filter(labels, add_edge_filter, footprint=footprint,
                          mode='nearest', extra_arguments=(g,))
    for n in g:
        g.node[n]['total color'] = np.zeros(3, np.double)
        g.node[n]['pixel count'] = 0
    for index in np.ndindex(labels.shape):
        n = labels[index]
        g.node[n]['total color'] += image[index]
        g.node[n]['pixel count'] += 1
    return g
```

```
def threshold_graph(g, t):
    to_remove = ((u, v) for (u, v, d) in g.edges(data=True)
                 if d['weight'] > t)
    g.remove_edges_from(to_remove)
```

```
In [68]:
```

```
# Baseline segmentation
from skimage import segmentation
seg = segmentation.slic(tiger, n_segments=30, compactness=40.0,
                        enforce_connectivity=True, sigma=3)
io.imshow(color.label2rgb(seg, tiger));
```



In Chapter 3, we set the graph threshold at 80 and sort of hand-waved over the whole thing. Now we're going to have a closer look at how this threshold impacts our segmentation accuracy. Let's pop the segmentation code into a function so we can play with it.

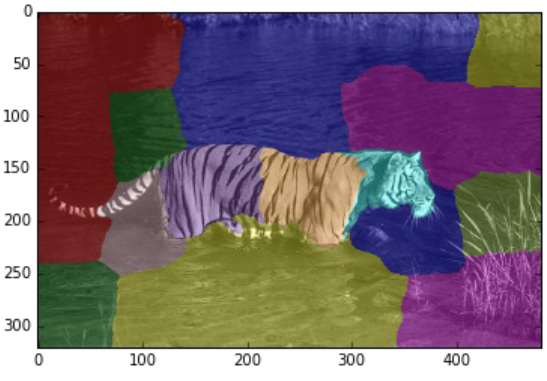
```
In [69]:
```

```
def RAG_segmentation(base_seg, image, threshold=80):
    g = build_rag(base_seg, image)
    for n in g:
        node = g.node[n]
        node['mean'] = node['total color'] / node['pixel count']
    for u, v in g.edges_iter():
        d = g.node[u]['mean'] - g.node[v]['mean']
        g[u][v]['weight'] = np.linalg.norm(d)
    threshold_graph(g, threshold)

    map_array = np.zeros(np.max(seg) + 1, int)
    for i, segment in enumerate(nx.connected_components(g)):
        for initial in segment:
            map_array[int(initial)] = i
    segmented = map_array[seg]
    return(segmented)
```

Let's try a few thresholds and see what happens:

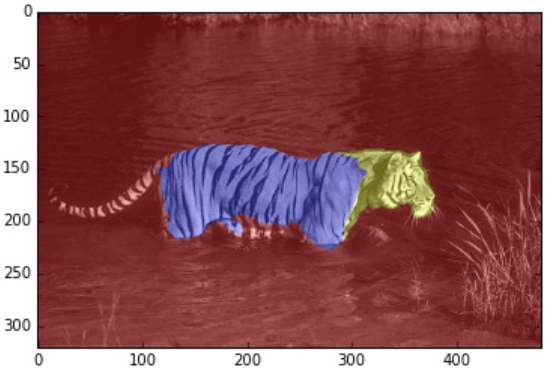
```
In [70]:
```

```
auto_seg_10 = RAG_segmentation(seg, tiger, threshold=10)
plt.imshow(color.label2rgb(auto_seg_10, tiger));
```




```
In [71]:
```

```
auto_seg_40 = RAG_segmentation(seg, tiger, threshold=40)
plt.imshow(color.label2rgb(auto_seg_40, tiger));
```



Actually, in Chapter 3 we did the segmentation a bunch of times with different thresholds and then (because we're human, so we can) picked one that produced a good segmentation. This is a completely unsatisfying way to program image segmentation. Clearly, we need a way to automate this.

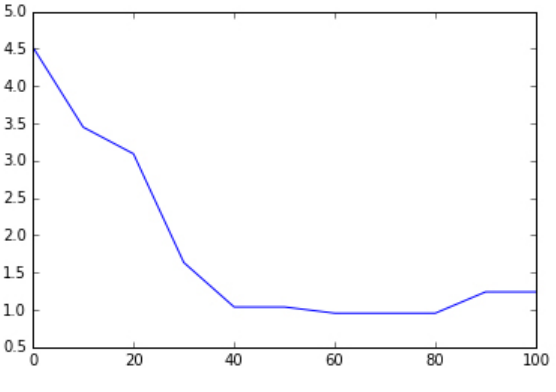
We can see that the higher threshold seems to producing a better segmentation. But we have a ground truth, so we can actually put a number to this! Using all our sparse

matrix skills, we can calculate the *variation of information* or VI for each segmentation.

```
In [72]:

vi(auto_seg_10, human_seg)

Out[72]:

3.44884607874861

In [73]:

vi(auto_seg_40, human_seg)

Out[73]:
1.0381218706889723
```

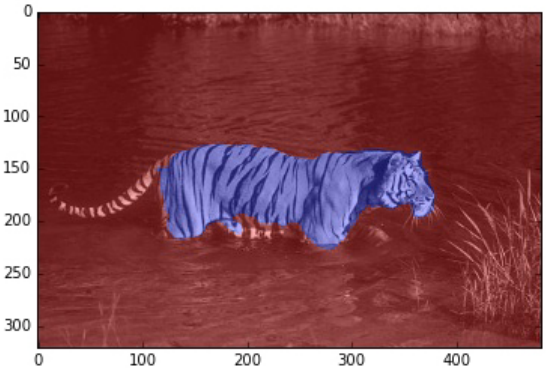
The high threshold has a smaller variation of information, so it's a better segmentation! Now we can calculate the VI for a range of possible thresholds and see which one gives us closes segmentation to the human ground truth.

```
In [74]:

# Try many thresholds
def vi_at_threshold(seg, tiger, human_seg, threshold):
    auto_seg = RAG_segmentation(seg, tiger, threshold)
    return vi(auto_seg, human_seg)

thresholds = range(0,110,10)
vi_per_threshold = [vi_at_threshold(seg, tiger, human_seg, threshold)
                    for threshold in thresholds]

In [75]:

plt.plot(thresholds, vi_per_threshold);
```



Unsurprisingly, it turns out that eyeballing it and picking threshold=80, did give us one of the best segmentations. But now we have a way to automate this process for any image!

```
In [76]:

auto_seg = RAG_segmentation(seg, tiger, threshold=80)
plt.imshow(color.label2rgb(auto_seg, tiger));
```



Exercise: Segmentation in practice

Try finding the best threshold for a selection of other images from the Berkeley Segmentation Dataset and Benchmark. Using the mean or median of those thresholds, then go and segment a new image. Did you get a reasonable segmentation?

CHAPTER 6

Linear algebra in SciPy

```
In [1]:
```

```
%matplotlib inline
```

Just like Chapter 4, which dealt with the Fast Fourier Transform, this chapter will feature an elegant *method*. We want to highlight the linear algebra packages available in SciPy, which form the basis of much scientific computing.

A chapter in a programming book is not really the right place to learn about linear algebra itself, so we assume familiarity with linear algebra concepts. At a minimum, you should know that linear algebra involves vectors (ordered collections of numbers) and their transformations by multiplying them with matrices (collections of vectors). If all of this sounded like gibberish to you, you should probably pick up an introductory linear algebra textbook before reading this. Introductory is all you need though — we hope to convey the power of linear algebra while keeping the operations relatively simple!

As an aside, we will break Python notation convention in order to match linear algebra conventions: in Python, variables names should usually begin with a lower case letter. However, in linear algebra, matrices are usually denoted by a capital letter, while vectors and scalar values are lowercase. Since we're going to be dealing with quite a few matrices and vectors, it helps to keep them straight to follow the linear algebra convention. Therefore, variables that represent matrices will start with a capital letter, while vectors and numbers will start with lowercase.

Laplacian matrix of a graph

We discussed graphs in Chapter 3, where we represented image regions as nodes, connected by edges between them. But we used a rather simple method of analysis: we *thresholded* the graph, removing all edges above some value. Thresholding works

in simple cases, but can easily fail, because all you need is one noisy edge to fall on the wrong side of the threshold for the approach to fail.

In this chapter, we will explore some alternative approaches to graph analysis, based on linear algebra. It turns out that we can think of a graph G as an *adjacency matrix*, in which we number the nodes of the graph from *0* to $n - 1$, and place a 1 in row i, column j of the matrix whenever there is an edge from node i to node j. In other words, if we call the adjacency matrix A, then $A_{i,j} = 1$ if and only if the link (i, j) is in G. We can then use linear algebra techniques to study this matrix, with often striking results.

The degree of a node is the number of edges touching it. For example, if a node is connected to five other nodes in a graph, its degree is 5. (Later, we will differentiate between out-degree and indegree, when edges have a "from" and "to".)

The *Laplacian* matrix of a graph (just "the Laplacian" for short) is defined as the *degree matrix*, D, which contains the degree of each node along the diagonal and zero everywhere else, minus the adjacency matrix A:

$$L = D - A$$

We definitely can't fit all of the linear algebra theory needed to understand the properties of this matrix, but suffice it to say: it has some *great* properties. We will exploit a couple in the following paragraphs.

For example, a common problem in network analysis is visualization. How do you draw nodes and links in such a way that you don't get a complete mess such as this one?

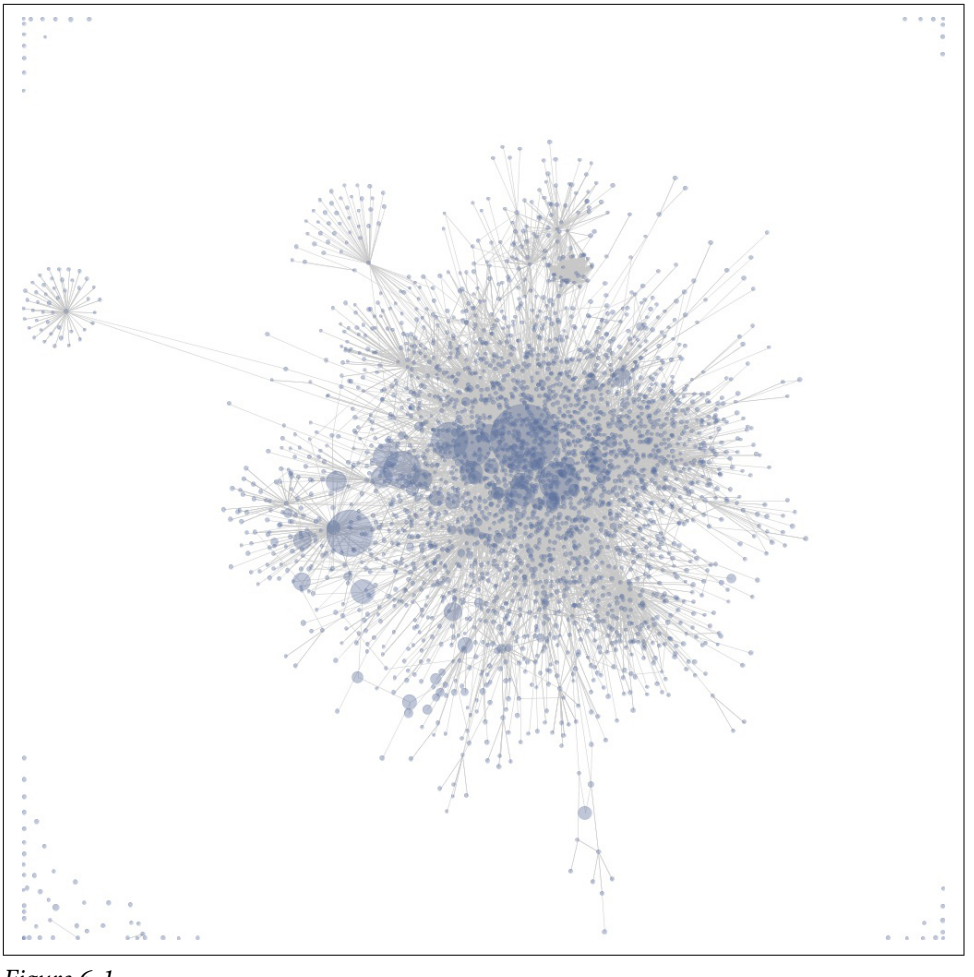


Figure 6-1.

Graph created by Chris Davis. CC-BY-SA-3.0. [Ed note: my reading is that we can use this figure as long as we include the copyright notice and don't put DRM on the books, which O'Reilly doesn't do anyway.]

One way is to put nodes that share many links close together, and it turns out that this can be done by using the second-smallest eigenvalue of the Laplacian matrix, and its corresponding eigenvector, which is so important it has its own name: the Fiedler vector.

As a quick aside, an eigenvector v of a matrix M is a vector that satisfies the property $Mv = \lambda v$ for some number λ, known as the eigenvalue. In other words, v is a special vector in relation to M because Mv simply scales the vector, without changing its direction.[1]

Exercise: Consider the rotation matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When R is multiplied with a 3-dimensional column-vector $p = [x\, y\, z]^T$, the resulting vector Rp is rotated by θ degrees around the z-axis.

1. For $\theta = 45^\circ$, verify (by testing on a few arbitrary vectors) that R rotates these vectors around the z axis.
2. Now, verify that multiplying by R leaves the vector $[0\,0\,1]^T$ unchanged. In other words, $Rp = 1p$, which means p is an eigenvector of R with eigenvalue 1.

The eigenvectors have numerous useful--sometimes seemingly magical!--properties.

Let's use a minimal network to illustrate this. We start by creating the adjacency matrix:

```
In [3]:

import numpy as np
A = np.array([[0, 1, 1, 0, 0, 0],
              [1, 0, 1, 0, 0, 0],
              [1, 1, 0, 1, 0, 0],
              [0, 0, 1, 0, 1, 1],
              [0, 0, 0, 1, 0, 1],
              [0, 0, 0, 1, 1, 0]], dtype=float)
```

We can use NetworkX to draw this network:

```
In [4]:

import networkx as nx
g = nx.from_numpy_matrix(A)
nx.draw_spring(g, with_labels=True, node_color='white')
```

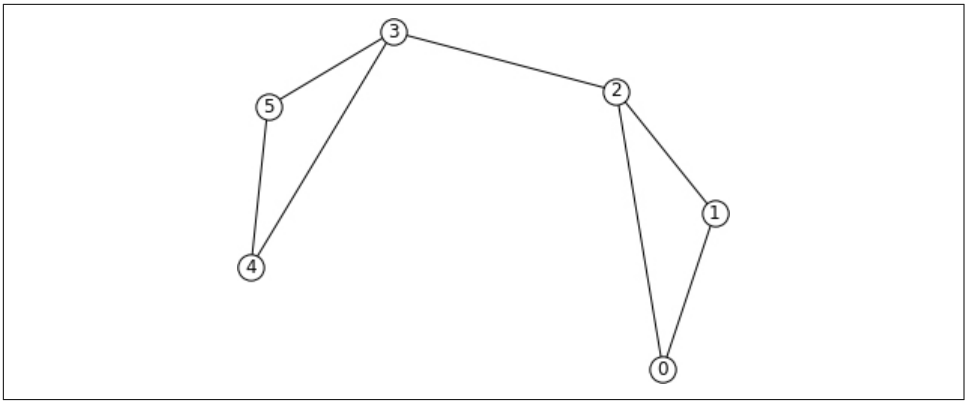


Figure 6-2.

You can see that the nodes fall naturally into two groups, 0, 1, 2 and 3, 4, 5. Can the Fiedler vector tell us this? First, we must compute the degree matrix and the Laplacian. We first get the degrees by summing along either axis of A. (Either axis works because A is symmetric.)

```
In [5]:

d = np.sum(A, axis=0)
print(d)

[ 2.  2.  3.  3.  2.  2.]
```

We then put those degrees into a diagonal matrix of the same shape as A, the *degree matrix*. We can use `scipy.sparse.diags` to do this:

```
In [6]:

from scipy import sparse
D = sparse.diags(d).toarray()
print(D)

[[ 2.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.]
 [ 0.  0.  3.  0.  0.  0.]
 [ 0.  0.  0.  3.  0.  0.]
 [ 0.  0.  0.  0.  2.  0.]
 [ 0.  0.  0.  0.  0.  2.]]
```

Finally, we get the Laplacian from the definition:

```
In [7]:

L = D - A
print(L)

[[ 2. -1. -1.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.]
 [-1. -1.  3. -1.  0.  0.]
```

```
 [ 0.  0. -1.  3. -1. -1.]
 [ 0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0. -1. -1.  2.]]
```

Because L is symmetric, we can use the `np.linalg.eigh` function to compute the eigenvalues and eigenvectors:

```
In [8]:

eigvals, Eigvecs = np.linalg.eigh(L)
```

You can verify that the values returned satisfy the definition of eigenvalues and eigenvectors. For example, the third eigenvalue is 3:

```
In [9]:

eigvals[2]

Out[9]:

2.9999999999999982
```

And we can check that multiplying the matrix L by the second eigenvector does indeed multiply the vector by 3:

```
In [10]:

v2 = Eigvecs[:, 2]

print(v2)
print(L @ v2)

[ 0.12545571 -0.5627829   0.43732719  0.43732719 -0.52639611  0.08906892]
[ 0.37636714 -1.6883487   1.31198156  1.31198156 -1.57918833  0.26720677]
```

As mentioned above, the Fiedler vector is the vector corresponding to the second-smallest eigenvalue of L. Plotting the eigenvalues tells us which one is the second-smallest:

```
In [11]:

from matplotlib import pyplot as plt
print(eigvals)
plt.plot(eigvals, '-o')

[  2.91433544e-16   4.38447187e-01   3.00000000e+00   3.00000000e+00
   3.00000000e+00   4.56155281e+00]

Out[11]:

[<matplotlib.lines.Line2D at 0x10e566710>]
```

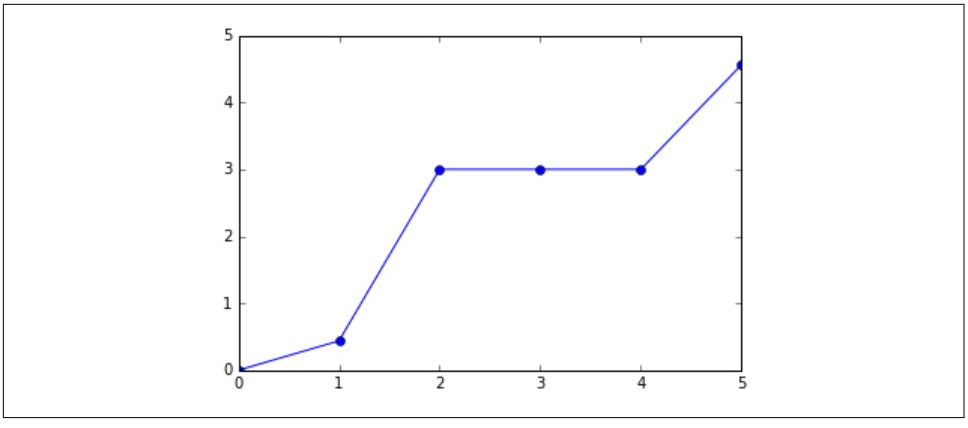


Figure 6-3.

It's the second eigenvalue. The Fiedler vector is thus the second eigenvector.

```
In [12]:
```

```
f = Eigvecs[:, 1]
plt.plot(f, '-o')
```

```
Out[12]:

[<matplotlib.lines.Line2D at 0x10e6680b8>]
```

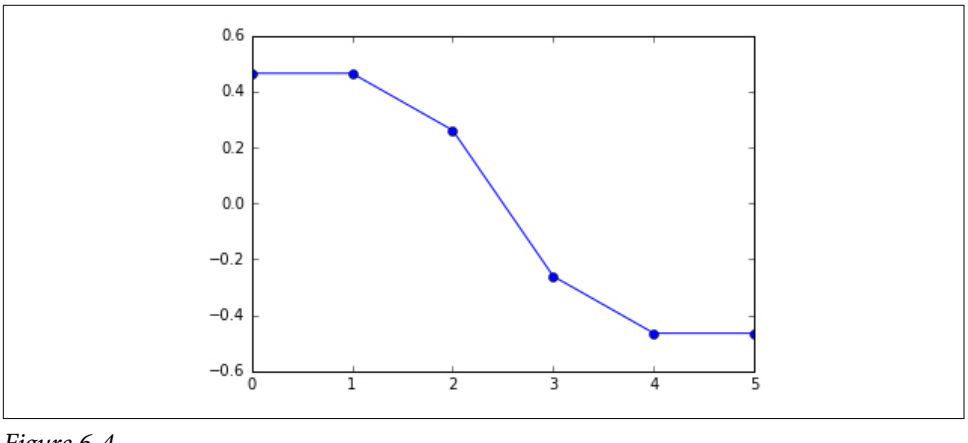


Figure 6-4.

It's pretty remarkable: by looking at the *sign* of the Fiedler vector, we can separate the nodes into the two groups we identified in the drawing!

```
In [13]:
```

```
colors = ['orange' if eigval > 0 else 'gray' for eigval in f]
nx.draw_spring(g, with_labels=True, node_color=colors)
```

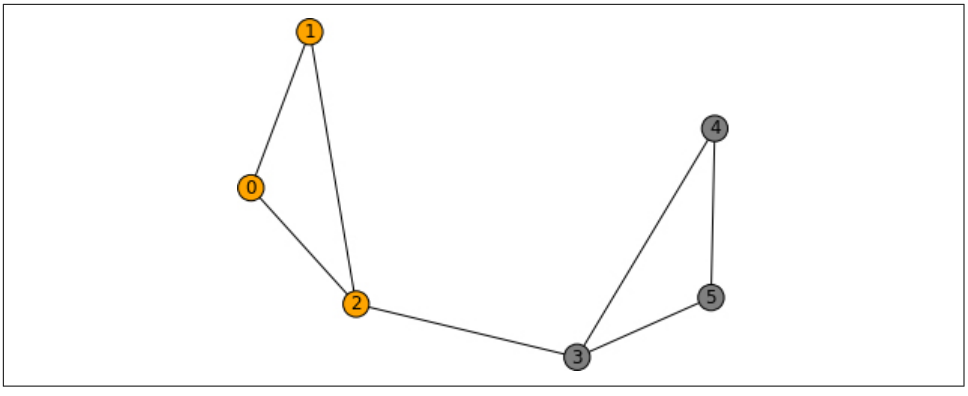


Figure 6-5.

Let's demonstrate this in a real-world example by laying out the brain cells in a worm, as shown in Figure 2 from the Varshney *et al* paper that we introduced in Chapter 3. (Information on how to do this is in the supplementary material for the paper.) To obtain their layout of the worm brain neurons, they used a related matrix, the *degree-normalized Laplacian*.

Because the order of the neurons is important in this analysis, we will use a preprocessed dataset, rather than clutter this chapter with data cleaning. We got the original data from Lars Varshney's website, and the processed data is in our `data/` directory.

First, let's load the data. There are four components:

- the network of chemical synapses, through which a *pre-synaptic neuron* sends a chemical signal to a *post-synaptic* neuron,
- the gap junction network, which contains direct electrical contacts between neurons),
- the neuron IDs (names), and
- the three neuron types:
  - *sensory neurons*, those that detect signals coming from the outside world, encoded as 0;
  - *motor neurons*, those that activate muscles, enabling the worm to move, encoded as 2; and
  - *interneurons*, the neurons in between, which enable complex signal processing to occur between sensory neurons and motor neurons, encoded as 1.

```
In [14]:
```

```
import numpy as np
chem = np.load('data/chem-network.npy')
```

```
gap = np.load('data/gap-network.npy')
neuron_ids = np.load('data/neurons.npy')
neuron_types = np.load('data/neuron-types.npy')
```

We then simplify the network, adding the two kinds of connections together, and removing the directionality of the network by taking the average of in-connections and out-connections of neurons. This seems a bit like cheating but, since we are only looking for the *layout* of the neurons on a graph, we only care about *whether* neurons are connected, not in which direction. We are going to call the resulting matrix the *connectivity* matrix, *C*, which is just a different kind of adjacency matrix.

```
In [15]:
```

```
A = chem + gap
C = (A + A.T) / 2
```

To get the Laplacian matrix *L*, we need the degree matrix *D*, which contains the degree of node i at position [i, i], and zeros everywhere else.

```
In [16]:
```

```
n = C.shape[0]
D = np.zeros((n, n), dtype=np.float)
diag = (np.arange(n), np.arange(n))
D[diag] = np.sum(C, axis=0)
L = D - C
```

The vertical coordinates in Fig 2 are given by arranging nodes such that, on average, neurons are as close as possible to "just above" their downstream neighbors. Varshney *et al* call this measure "processing depth," and it's obtained by solving a linear equation involving the Laplacian. We use `scipy.linalg.pinv`, the pseudoinverse, to solve it:

```
In [17]:
```

```
from scipy import linalg
b = np.sum(C * np.sign(A - A.T), axis=1)
z = linalg.pinv(L) @ b
```

(Note the use of the `@` symbol, which was introduced in Python 3.5 to denote matrix multiplication. As we noted in the preface and in Chapter 5, in previous versions of Python, you would need to use the function `np.dot`.)

In order to obtain the degree-normalized Laplacian, Q, we need the inverse square root of the D matrix:

```
In [18]:
```

```
Dinv2 = np.zeros((n, n))
Dinv2[diag] = D[diag] ** (-.5)
Q = Dinv2 @ L @ Dinv2
```

Finally, we are able to extract the x coordinates of the neurons to ensure that highly-connected neurons remain close: the eigenvector of Q corresponding to its second-smallest eigenvalue, normalized by the degrees:

```
In [19]:

eigvals, eigvecs = linalg.eig(Q)
```

Note from the documentation of `numpy.linalg.eig`:

"The eigenvalues are not necessarily ordered."

Although the documentation in SciPy's `eig` lacks this warning (disappointingly, we must add), it remains true in this case. We must therefore sort the eigenvalues and the corresponding eigenvector columns ourselves:

```
In [20]:

smallest_first = np.argsort(eigvals)
eigvals = eigvals[smallest_first]
eigvecs = eigvecs[:, smallest_first]
```

Now we can find the eigenvector we need to compute the affinity coordinates:

```
In [21]:

x = Dinv2 @ eigvecs[:, 1]
```

(The reasons for using this vector are too long to explain here, but appear in the paper's supplementary material, linked above.)

Now it's just a matter of drawing the nodes and the links. We color them according to the type stored in `neuron_types`:

```
In [22]:

from matplotlib import pyplot as plt
from matplotlib import colors

def plot_connectome(neuron_x, neuron_y, links, labels, types):
    colormap = colors.ListedColormap([[ 0.   ,  0.447,  0.698],
                                      [ 0.   ,  0.62 ,  0.451],
                                      [ 0.835,  0.369,  0.   ]])
    # plot neuron locations:
    points = plt.scatter(neuron_x, neuron_y, c=types, cmap=colormap,
                         edgecolors='face', zorder=1)

    # add text labels:
    for x, y, label in zip(neuron_x, neuron_y, labels):
        plt.text(x, y, ' ' + label,
                 horizontalalignment='left', verticalalignment='center',
                 fontsize=5, zorder=2)

    # plot links
    pre, post = np.nonzero(links)
```

```
    for src, dst in zip(pre, post):
        plt.plot(neuron_x[[src, dst]], neuron_y[[src, dst]],
                 c=(0.85, 0.85, 0.85), lw=0.2, alpha=0.5, zorder=0)
    plt.show()
```

```
In [23]:
```

```
plt.figure(figsize=(16, 9))
plot_connectome(x, z, C, neuron_ids, neuron_types)
```

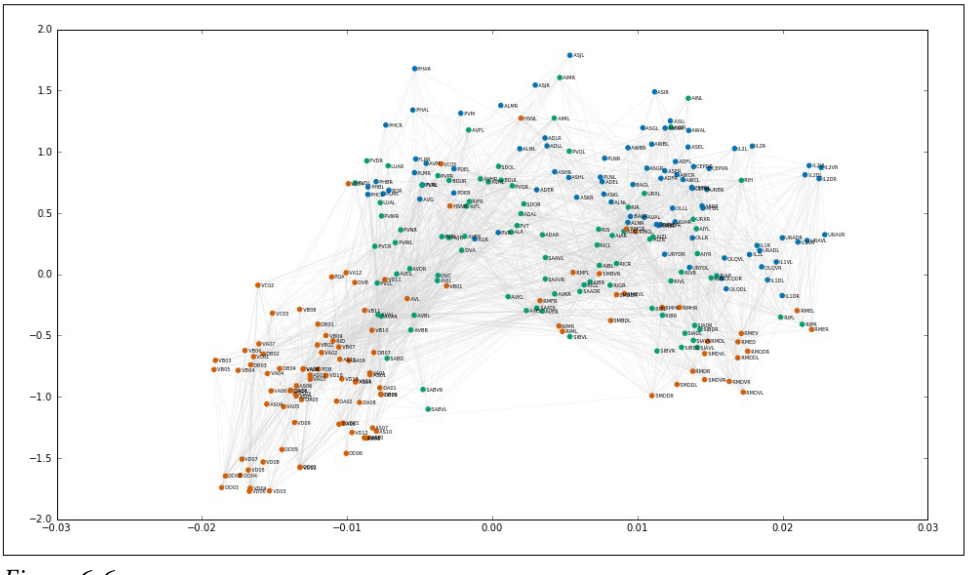


Figure 6-6.

There you are: a worm brain! As discussed in the original paper, you can see the top-down processing from sensory neurons to motor neurons through a network of interneurons. You can also see two distinct groups of motor neurons: these correspond to the neck (left) and body (right) body segments of the worm.

Exercise: How do you modify the above code to show the affinity view in Figure 2B from the paper?

Challenge: linear algebra with sparse matrices

The above code uses numpy arrays to hold the matrix and perform the necessary computations. Because we are using a small graph of fewer than 300 nodes, this is feasible. However, for larger graphs, it would fail.

In what follows, we will analyze the dependency graph for packages in the Python Package Index, or PyPI, which contains over 75 thousand packages. To hold the Laplacian matrix for this graph would take up $8\left(75 \times 10^3\right)^2 = 45 \times 10^9$ bytes, or 45GB, of RAM. If you add to that the adjacency, symmetric adjacency, pseudoinverse, and, say,

two temporary matrices used during calculations, you climb up to 270GB, beyond the reach of most desktop computers.

"Ha!", some of you might be thinking. "Ha! My desktop has 512GB of RAM! It would make short work of this so-called 'large' graph!"

Perhaps. But we will also be analysing the Association for Computing Machinery (ACM) citation graph, a network of over two million scholarly works and references. *That* Laplacian would take up 32 terabytes of RAM.

However, we know that the dependency and reference graphs are *sparse*: packages usually depend on just a few other packages, not on the whole of PyPI. And papers and books usually only reference a few others, too. So we can hold the above matrices using the sparse data structures from `scipy.sparse` (see Chapter 5), and use the linear algebra functions in `scipy.sparse.linalg` to compute the values we need.

Try to explore the documentation in `scipy.sparse.linalg` to come up with a sparse version of the above computation.

Hint: the pseudoinverse of a sparse matrix is, in general, not sparse, so you can't use it here. Similarly, you can't get all the eigenvectors of a sparse matrix, because they would together make up a dense matrix.

You'll find parts of the solution below (and of course in the solutions chapter), but we highly recommend that you try it out on your own.

Pagerank: linear algebra for reputation and importance

Another application of linear algebra and eigenvectors is Google's Pagerank algorithm, which is punnily named both for webpages and for one of its co-founders, Larry Page.

If you're trying to rank webpages by importance, one thing you might look at is how many other webpages link to it. After all, if everyone is linking to a particular page, it must be good, right? But the problem is that this metric is easily gamed: to make your own webpage rise in the rankings, you just have to create as many other webpages as you can and have them all link to your original page.

The key insight that drove Google's early success was that important webpages are not just linked to by many webpages, but also by *other*, *important* webpages. And how do we know that those other pages are important? Because they themselves are linked to by important pages. And so on. As we will see, this recursive definition implies that page importance can be measured by the eigenvector corresponding to the largest eigenvalue of the so-called *transition matrix*. This matrix imagines a web surfer, often named Webster, randomly clicking a link from each webpage he visits, and then asks,

what's the probability that he ends up at any given page? This probability is called the pagerank.

Since Google's rise, researchers have been applying pagerank to all sorts of networks. We'll start with an example by Stefano Allesina and Mercedes Pascual, which they published in PLoS Computational Biology. They thought to apply the method in ecological *food webs*, networks that link species to those that they eat.

Naively, if you wanted to see how critical a species was for an ecosystem, you would look at how many species eat it. If it's many, and that species disappeared, then all its "dependent" species might disappear with it. In network parlance, you could say that its *in-degree* determines its ecological importance.

Could pagerank be a better measure of importance for an ecosystem?

Professor Allesina kindly provided us with a few food webs to play around with. We've saved one of these, from the St Marks National Wildlife Refuge in Florida, in the Graph Markup Language format. The web was described in 1999 by Robert R. Christian and Joseph J. Luczovich. In the dataset, a node *i* has a link to node *j* if species *i* eats species *j*.

We'll start by loading in the data, which NetworkX knows how to read trivially:

```
In [34]:

import networkx as nx

stmarks = nx.read_gml('data/stmarks.gml')
```

Next, we get the sparse matrix corresponding to the graph. Because a matrix only holds numerical information, we need to maintain a separate list of package names corresponding to the matrix rows/columns:

```
In [35]:

species = np.array(stmarks.nodes())  # array for multi-indexing
Adj = nx.to_scipy_sparse_matrix(stmarks, dtype=np.float64)
```

From the adjacency matrix, we can derive a *transition probability* matrix, where every link is replaced by a *probability* of 1 over the number of outgoing links from that species. In the food web, it might make more sense to call this a lunch probability matrix.

The total number of species in our matrix is going to be used a lot, so let's call it *n*:

```
In [36]:

n = len(species)
```

Next, we need the degrees, and, in particular, the *diagonal matrix* containing the inverse of the out-degrees of each node on the diagonal:

```
In [37]:
```

```
np.seterr(divide='ignore')  # ignore division-by-zero errors
from scipy import sparse

degrees = np.ravel(Adj.sum(axis=1))
Deginv = sparse.diags(1 / degrees).tocsr()
```

```
In [38]:
```

```
Trans = (Deginv @ Adj).T
```

Normally, the pagerank score would simply be the first eigenvector of the transition matrix. If we call the transition matrix M and the vector of pagerank values r, we have:

$$\boldsymbol{r} = M\boldsymbol{r}$$

But the `np.seterr` call above is a clue that it's not quite so simple. The pagerank approach only works when the transition matrix is a *column-stochastic* matrix, in which every column sums to 1. Additionally, every page must be reachable from every other page, even if the path to reach it is very long.

In our food web, this causes problems, because the bottom of the food chain, what the authors call *detritus* (basically sea sludge), doesn't actually *eat* anything (the Circle of Life notwithstanding), so you can't reach other species from it.

To deal with this, the pagerank algorithm uses a so-called "damping factor", usually taken to be 0.85. This means that 85% of the time, the algorithm follows a link at random, but for the other 15%, it randomly jumps to any arbitrary page. It's as if every page had a low probability link to every other page. Or, in our case, it's as if shrimp, on rare occasions, ate sharks. It might seem nonsensical but bear with us! It is, in fact, the mathematical representation of the Circle of Life. We'll set it to 0.99, but actually it doesn't really matter for this analysis: the results are similar for a large range of possible damping factors.

If we call the damping factor d, then the modified pagerank equation is:

$$\boldsymbol{r} = dM\boldsymbol{r} + \frac{1-d}{n}\boldsymbol{1}$$

and

$$(\boldsymbol{I} - dM)\boldsymbol{r} = \frac{1-d}{n}\boldsymbol{1}$$

We can solve this equation using `scipy.sparse`'s iterative *biconjugate gradient (bicg)* solver^[bicgstab]. (Note that, if you had a symmetric matrix, you would use Conjugate Gradients, `scipy.sparse.linalg.solve.cg`, instead.)

```
In [39]:
```

```
from scipy.sparse.linalg.isolve import bicgstab as bicg

damping = 0.99

I = sparse.eye(n, format='csc')  # Same sparse format as Trans

pagerank, error = bicg(I - damping * Trans,
                       (1-damping) / n * np.ones(n),
                       maxiter=int(1e4))
print('error code: ', error)
```

```
error code: 0
```

As can be seen in the documentation for the `bicg` solver, an error code of 0 indicates that a solution was found! We now have the "foodrank" of the St. Marks food web!

So how does a species' foodrank compare to the number of other species eating it?

```
In [40]:
```

```
def pagerank_plot(names, in_degrees, pageranks,
                  annotations=[]):
    fig, ax = plt.subplots(figsize=(9, 5))
    ax.scatter(in_degrees, pageranks, c=[0.835, 0.369, 0], lw=0)
    labels = []
    for name, indeg, pr in zip(names, in_degrees, pageranks):
        if name in annotations:
            text = ax.text(indeg + 0.1, pr, name)
            labels.append(text)
    ax.set_ylim(0, np.max(pageranks) * 1.1)
    ax.set_xlim(-1, np.max(in_degrees) * 1.1)
    ax.set_ylabel('PageRank')
    ax.set_xlabel('In-degree')

interesting = ['detritus', 'phytoplankton', 'benthic algae', 'micro-epiphytes',
               'microfauna', 'zooplankton', 'predatory shrimps', 'meiofauna', 'gulls']
in_degrees = np.ravel(Adj.sum(axis=0))
pagerank_plot(species, in_degrees, pagerank, annotations=interesting)
```

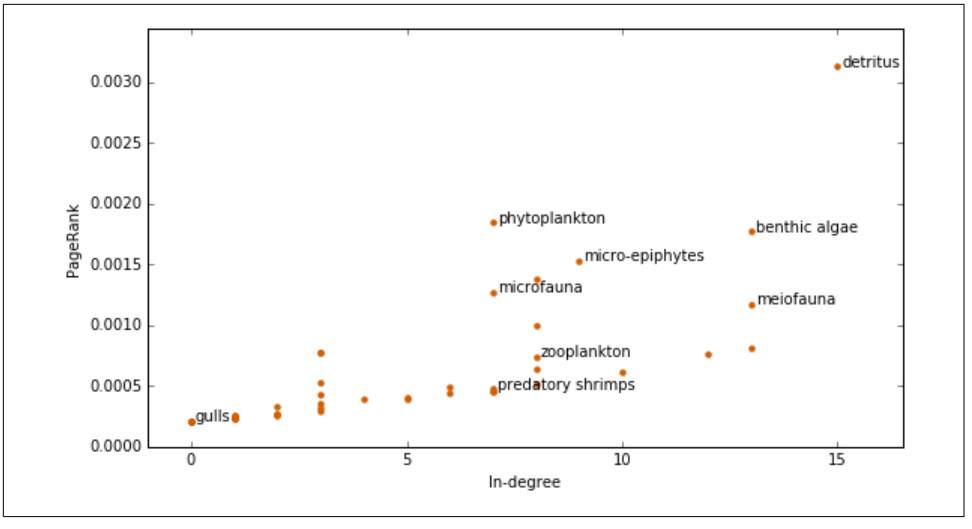


Figure 6-7.

Having explored the dataset ahead of time, we have pre-labeled some interesting nodes in the plot. Sea sludge is the most important element both by number of species feeding on it (15) and by pagerank (>0.003). But the second most important element is *not* benthic algae, which feeds 13 other species, but rather phytoplankton, which feeds just 7! That's because other *important* species feed on it! On the bottom left, we've got sea gulls, who, we can now confirm, do bugger-all for the ecosystem. Those vicious *predatory shrimps* (we're not making this up) support the same number of species as phytoplankton, but they are less essential species, so they end up with a lower foodrank.

Although we won't do it here, Allesina and Pascual go on to model the ecological impact of species extinction, and indeed find that pagerank predicts ecological importance better than in-degree.

Before we move on though, we'll note that pagerank can be computed several different ways. One way, complementary to what we did above, is called the *power method*, and it's pretty, well, powerful! It stems from the Perron-Frobenius theorem, which states, among other things, that a stochastic matrix has 1 as an eigenvalue, and that this is its *largest* eigenvalue. (The corresponding eigenvector is the pagerank vector.) What this means is that, whenever we multiply *any* vector by M, its component pointing towards this major eigenvector stays the same, while *all other components shrink* by a multiplicative factor! The consequence is that if we multiply some random starting vector by M repeatedly, we should eventually get the pagerank vector.

SciPy makes this very efficient with its sparse matrix module:

```
In [41]:
```

```
def power(Trans, damping=0.85, max_iter=int(1e5)):
    n = Trans.shape[0]
    r0 = np.full(n, 1/n)
    r = r0
    for _iter_num in range(max_iter):
        rnext = damping * Trans @ r + (1 - damping) / n
        if np.allclose(rnext, r):
            print('converged')
            break
        r = rnext
    return r
```

Exercise: In the above iteration, note that `Trans` is *not* column-stochastic, so the vector gets shrunk at each iteration. In order to make the matrix stochastic, we have to replace every zero-column by a column of all $1/n$. This is too expensive, but computing the iteration is cheaper. How can you modify the code above to ensure that r remains a probability vector throughout?

Exercise: Verify that these three methods all give the same ranking for the nodes. `numpy.corrcoef` might be a useful function for this.

Exercise: While we were writing this chapter, we started out by computing pagerank on the graph of Python dependencies. We eventually found that we could not get nice results with this graph. The correlation between in-degree and pagerank was much higher than in other datasets, and the few outliers didn't make much sense to us.

Can you think of three reasons why pagerank might not be the best measure of importance for the Python dependency graph?

Community detection

We saw a hint in the introduction to this chapter that the Fiedler vector could be used to detect "communities" in networks, groups of nodes that are tightly connected to each other but not so much to nodes in other groups. Mark Newman published a seminal paper on the topic in 2006, and refined it further in 2013. We'll apply it to the Python library dependency graph, which will tell us whether Python users fall into two or more somewhat independent groups.

We've downloaded and preprocessed dependency data from PyPI ahead of time, available as the file `pypi-dependencies.txt` in the `data/` folder. The data file consists of a list of library-dependency pairs, one per line, e.g. `scipy numpy`. The networkx library that we started using in Chapter 3 makes it easy to build a graph from these data. We will use a directed graph since the relationship is asymmetrical: one library depends on the other, not vice-versa.

```
In [44]:
```

```
import networkx as nx

dependencies = nx.DiGraph()

with open('data/pypi-deps.txt') as lines:
    lib_dep_pairs = (str.split(line) for line in lines)
    dependencies.add_edges_from(lib_dep_pairs)
```

We can then get some statistics about this (incomplete) dataset by using networkx's built-in functions:

```
In [45]:

print('Number of packages: ', dependencies.number_of_nodes())
print('Total number of dependencies: ', dependencies.number_of_edges())

Number of packages:  95945
Total number of dependencies:  271017
```

What is the single most used Python package?

```
In [46]:

print(max(dependencies.in_degree_iter(),
          key=lambda x: x[1]))

('setuptools', 46332)
```

We're not going to cover it in this book, but `setuptools` is not a surprising winner here. It probably belongs in the Python standard library, in the same category as `os`, `sys`, and others!

Since using setuptools is almost a requirement for being listed in PyPI, we will remove it from the dataset, given its undue influence on the shape of the graph.

```
In [47]:

dependencies.remove_node('setuptools')
```

What's the next most depended-upon package?

```
In [48]:

print(max(dependencies.in_degree_iter(),
          key=lambda x: x[1]))

('requests', 5880)
```

The requests library is the foundation of a very large fraction of the web frameworks and web processing libraries.

We can similarly find the top-40 most depended-upon packages:

```
In [49]:

packages_by_in = sorted(dependencies.in_degree_iter(),
                        key=lambda x: x[1], reverse=True)
for i, p in enumerate(packages_by_in, start=1):
```

```
    print(i, '. ', p[0], p[1])
    if i > 40:
        break
```

```
1 .  requests 5880
2 .  pkg_resources 5718
3 .  django 4376
4 .  numpy 3856
5 .  six 2837
6 .  mock 2788
7 .  pytest 2498
8 .  yaml 2260
9 .  simplejson 2236
10 .  ez_setup 1592
11 .  flask 1556
12 .  jinja2 1509
13 .  sqlalchemy 1322
14 .  utils 1251
15 .  lxml 1250
16 .  pil 1183
17 .  unittest2 1139
18 .  matplotlib 1020
19 .  zope 961
20 .  pytz 952
21 .  models 925
22 .  bs4 867
23 .  pandas 867
24 .  testing 849
25 .  click 830
26 .  redis 809
27 .  scipy 776
28 .  cython 771
29 .  settings 766
30 .  nose 729
31 .  config 712
32 .  acquisition 700
33 .  transaction 675
34 .  posixpath 651
35 .  docutils 646
36 .  docopt 641
37 .  base 595
38 .  accesscontrol 582
39 .  pypandoc 569
40 .  pymongo 561
41 .  util 559
```

By this ranking, NumPy ranks 4 and SciPy 27 out of all of PyPI. Not bad! Overall, though, one gets the impression that the web community dominates PyPI. As also mentioned in the preface, this is not altogether surprising: the scientific Python community is still young and growing, and web tools are arguably of more generic application.

Let's see whether we can isolate the scientific community.

Because it's unwieldy to draw 90,000 nodes, we are only going to draw a fraction of PyPI, following the same ideas we used for the worm brain. Let's look at the top 10,000 packages in PyPI, according to number of dependencies:

```
In [50]:

n = 10000
top_names = [p[0] for p in packages_by_in[:n]]
top_subgraph = nx.subgraph(dependencies, top_names)
Dep = nx.to_scipy_sparse_matrix(top_subgraph, nodelist=top_names)
```

As above, we need the connectivity matrix, the symmetric version of the adjacency matrix:

```
In [51]:

Conn = (Dep + Dep.T) / 2
```

And the diagonal matrix of its degrees, as well as its inverse-square-root:

```
In [52]:

degrees = np.ravel(Conn.sum(axis=0))
Deg = sparse.diags(degrees).tocsr()
Dinv2 = sparse.diags(degrees ** (-.5)).tocsr()
```

From this we can generate the Laplacian of the dependency graph:

```
In [53]:

Lap = Deg - Conn
```

We can then generate an affinity view of these nodes, as shown above for the worm brain graph. We just need the second and third smallest eigenvectors of the *affinity matrix*, the degree-normalized version of the Laplacian. We can use `sparse.linalg.eigsh` to obtain these.

Note that eigenvectors here are calculated through an iterative method, for which the convergence may depend on the starting vector `v0`. By default, this vector is chosen at random, but we set it explicitly to all ones to ensure *reliable* convergence.

```
In [54]:

I = sparse.eye(Lap.shape[0], format='csr')
sigma = 0.5

Affn = Dinv2 @ Lap @ Dinv2
v0 = np.ones(Lap.shape[0])

eigvals, vec = sparse.linalg.eigsh(Affn + sigma*I, k=3, which='SM', v0=v0)

sorted_indices = np.argsort(eigvals)
eigvals = eigvals[sorted_indices]
```

```
vec = vec[:, sorted_indices]

_ignored, x, y = (Dinv2 @ vec).T
```

That should give us a nice layout for our Python packages!

```
In [55]:

plt.figure()
plt.scatter(x, y)

Out[55]:

<matplotlib.collections.PathCollection at 0x11546bef0>
```

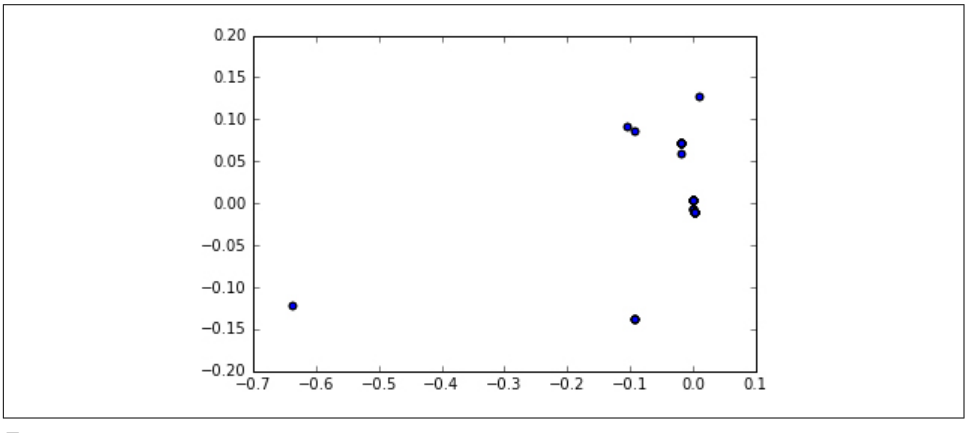


Figure 6-8.

That's looking promising! But, obviously, this plot is missing some critical details! Which packages depend on which? Where are the most important packages? Perhaps most critically, do particular packages naturally belong together? In the worm brain, we had prior information about neuron types that helped us to make sense of the graph. Let's try to infer some similar information about PyPI packages.

[Further content to be added in final edition]

CHAPTER 7

CHAPTER 8
Big Data in Little Laptop with Toolz

```
In [1]:
```

```
%matplotlib inline
```

Whenever I think too hard about streaming data analysis, my head hurts.

You have probably already done some streaming, perhaps without thinking about it in these terms. The simplest form is probably iterating through lines in a files, processing each line without ever reading the entire file into memory. For example a loop like this to calculate the mean of each row and sum them:

```
In [2]:
```

```
import numpy as np
with open('data/expr.tsv') as f:
    sum_of_means = 0
    for line in f:
        sum_of_means += np.mean(np.fromstring(line, dtype=int, sep='\t'))
print(sum_of_means)
```

```
1463.0
```

This strategy works really well for cases where your problem can be neatly solved with by-row processing. But things can quickly get out of hand when your code becomes more sophisticated.

In traditional programming models, you pass your data to a function, the function processes the data, and then returns the result. Done.

But in streaming programs, a function processes *some* of the data, returns the processed chunk, then, while downstream functions are dealing with that chunk, the function receives a bit more, and so on... All these things are going on at the same time! How can one keep them straight?

For many years, I didn't. But Matt Rocklin's blog posts on this topic really opened my eyes to the utility and elegance of streaming data analysis, to the point that it was impossible to contemplate writing this book without including a chapter on it. Although streaming is not really a SciPy feature, it is a way of writing code that is critical to efficiently processing the large datasets that we see in science. The Python language contains some very nice primitives for streaming data processing, and these can be combined with Matt's Toolz library to generate gorgeous, concise code that is extremely memory-efficient. We will show you how to apply streaming concepts to make your SciPy code fast and and elegant.

Let me clarify what I mean by "streaming" and why you might want to do it. Suppose you have some data in a CSV text file, and you want to compute the column-wise average of log $(x + 1)$ of the values. The most common way to do this would be to use NumPy to load the values, compute the log function for all values in the full matrix, and then take the mean over the 1st axis:

```
In [3]:

import numpy as np
expr = np.loadtxt('data/expr.tsv')
logexpr = np.log(expr + 1)
np.mean(logexpr, axis=1)

Out[3]:

array([ 6.22172881,  0.03465736,  0.45502628,  5.38409181,  1.98899162])
```

This works, and it follows a reassuringly familiar input-output model of computation. But it's a pretty inefficient way to go about it! We load the full matrix into memory (1), then make a copy with 1 added to each value (2), then make another copy to compute the log (3), before finally passing it on to `np.mean`. That's three instances of the data array, to perform an operation that doesn't require keeping even *one* instance in memory. It's clear that for any kind of "big data" operation, this approach won't work.

Python's creators knew this, and created the "yield" keyword, which enables a function to process just one "sip" of the data, pass the result on to the next process, and *let the chain of processing complete* for that one piece of data before moving on to the next one. "Yield" is a rather nice name for it: the function *yields* control to the next function, waiting to resume processing the data until all the downstream steps have processed that data point.

As I mentioned above, trying to think too hard about the flow of control in this paradigm is a surefire way to experience headaches, nausea, and other side effects. An awesome feature of Python is that it abstracts this complexity away, allowing you to focus on the analysis functionality. Here's how I think about it: for every processing function that would normally take a list (a collection of data) and transform that list,

simply rewrite that function as taking a *stream* and *yielding* the result of every element of that stream.

Here's an example where we take the log of each element in a list, using either a standard data-copying method or a streaming method:

```
In [4]:

def log_all_standard(input):
    output = []
    for elem in input:
        output.append(np.log(elem))
    return output

def log_all_streaming(input_stream):
    for elem in input_stream:
        yield np.log(elem)

In [5]:

# We set the random seed so we will get consistent results
np.random.seed(seed=7)
# Set print options to show only 3 significant digits
np.set_printoptions(precision=3, suppress=True)

In [6]:

arr = np.random.rand(1000) + 0.5
result_batch = sum(log_all_standard(arr))
print(result_batch)

-48.2409194561

In [7]:

result_stream = sum(log_all_streaming(arr))
print(result_stream)

-48.2409194561
```

The advantage of the streaming approach is that elements of a stream aren't processed until they're needed, whether it's for computing a running sum, or for writing out to disk, or something else. This can conserve a lot of memory when you have many input items, or when each item is very big. (Or both!) This quote from one of Matt's posts very succinctly summarizes the utility of streaming data analysis:

> *In my brief experience people rarely take this [streaming] route. They use single-threaded in-memory Python until it breaks, and then seek out Big Data Infrastructure like Hadoop/Spark at relatively high productivity overhead.*

Indeed, this describes my computational career perfectly, up until recent months. But the intermediate approach can get you a *lot* farther than you think. In some cases, it can get you there even faster than the supercomputing approach, by eliminating the overhead of multi-core communication and random-access to databases. (For exam-

ple, see this post by Frank McSherry, where he processes a 128 billion edge graph on his laptop *faster* than using a graph database on a supercomputer.)

To clarify the flow of control when using streaming-style functions, it's useful to make *verbose* versions of the functions, which print out a message with each operation.

```
In [8]:
```

```
import numpy as np

def tsv_line_to_array(line):
    lst = [float(elem) for elem in line.rstrip().split('\t')]
    return np.array(lst)

def readtsv_verbose(filename):
    print('starting readtsv')
    with open(filename) as fin:
        for i, line in enumerate(fin):
            print('reading line {}'.format(i))
            yield tsv_line_to_array(line)
    print('finished readtsv')

def add1_verbose(arrays_iter):
    print('starting adding 1')
    for i, arr in enumerate(arrays_iter):
        print('adding 1 to line {}'.format(i))
        yield arr + 1
    print('finished adding 1')

def log_verbose(arrays_iter):
    print('starting log')
    for i, arr in enumerate(arrays_iter):
        print('taking log of array {}'.format(i))
        yield np.log(arr)
    print('finished log')

def running_mean_verbose(arrays_iter):
    print('starting running mean')
    for i, arr in enumerate(arrays_iter):
        if i == 0:
            mean = arr
        mean += (arr - mean) / (i + 1)
        print('adding line {} to the running mean'.format(i))
    print('returning mean')
    return mean
```

Let's see it in action for a small sample file:

```
In [9]:
```

```
fin = 'data/expr.tsv'
print('Creating lines iterator')
lines = readtsv_verbose(fin)
print('Creating loglines iterator')
```

```
loglines = log_verbose(add1_verbose(lines))
print('Computing mean')
mean = running_mean_verbose(loglines)
print('the mean log-row is: {}'.format(mean))
```

```
Creating lines iterator
Creating loglines iterator
Computing mean
starting running mean
starting log
starting adding 1
starting readtsv
reading line 0
adding 1 to line 0
taking log of array 0
adding line 0 to the running mean
reading line 1
adding 1 to line 1
taking log of array 1
adding line 1 to the running mean
reading line 2
adding 1 to line 2
taking log of array 2
adding line 2 to the running mean
reading line 3
adding 1 to line 3
taking log of array 3
adding line 3 to the running mean
reading line 4
adding 1 to line 4
taking log of array 4
adding line 4 to the running mean
finished readtsv
finished adding 1
finished log
returning mean
the mean log-row is: [ 3.118  2.487  2.196  2.36  2.701  2.647  2.437  3.285  2.054  2.372
  3.855  3.949  2.467  2.363  3.184  2.644  2.63  2.848  2.617  4.125]
```

Note a few things:

- None of the computation is run when creating the lines and loglines iterators. This is because iterators are *lazy*, meaning they are not evaluated (or *consumed*) until a result is needed.
- When the computation is finally triggered, by the call to `running_mean_verbose`, it jumps back and forth between all the functions, as various computations are performed on each line, before moving on to the next line.

Introducing the Toolz streaming library

This chapter's code example is from Matt Rocklin (who else?), in which he creates a Markov model from an entire fly genome in under 5 minutes on a laptop, using just a few lines of code. (It has been slightly edited for easier downstream processing.) Matt's example uses a human genome, but apparently our laptops weren't quite so fast, so we're going to use a fly genome instead (it's about 1/20 the size). Over the course of the chapter we'll actually augment it a little bit to start from compressed data (who wants to keep an uncompressed dataset on their hard drive?). This modification is almost *trivial*, which speaks to the elegance of his example.

```
In [10]:
```

```
import toolz as tz
from toolz import curried as c
from glob import glob
import itertools as it

LDICT = dict(zip('ACGTacgt', range(8)))
PDICT = {(a, b): (LDICT[a], LDICT[b])
         for a, b in it.product(LDICT, LDICT)}

def is_sequence(line):
    return not line.startswith('>')

def is_nucleotide(letter):
    return letter in LDICT  # ignore 'N'

@tz.curry
def increment_model(model, index):
    model[index] += 1

def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted,  # Filenames
                   c.map(open),  # lines
                   # concatenate lines from all files:
                   tz.concat,
                   # drop header from each sequence
                   c.filter(is_sequence),
                   # concatenate characters from all lines
                   tz.concat,
                   # discard newlines and 'N'
                   c.filter(is_nucleotide))


def markov(seq):
    """Get a 1st-order Markov model from a sequence of nucleotides."""
    model = np.zeros((8, 8))
    tz.last(tz.pipe(seq,
                    c.sliding_window(2),        # each successive tuple
```

```
                c.map(PDICT.__getitem__),   # location in matrix of tuple
                c.map(increment_model(model))))  # increment matrix
    # convert counts to transition probability matrix
    model /= np.sum(model, axis=1)[:, np.newaxis]
    return model
```

We can then do the following to obtain a Markov model of repetitive sequences in the fruit-fly genome:

```
In [11]:
```

```
%%timeit -r 1 -n 1
dm = 'data/dm6.fa'
model = tz.pipe(dm, genome, c.take(1000000), markov)
# we use `take` to just run on the first 1000000 bases, to speed things up.
# the take step can just be removed if you have ~5-10 mins to wait.
```

```
1 loop, best of 1: 3.85 s per loop
```

There's a *lot* going on in that example, so we are going to unpack it little by little. We'll actually run the example at the end of the chapter.

The first thing to note is how many functions come from the Toolz library. For example from Toolz we've used, `pipe`, `sliding_window`, `frequencies`, and a curried version of `map` (more on this later). That's because Toolz is written specifically to take advantage of Python's iterators, and easily manipulate streams.

Let's start with `pipe`. This function is simply syntactic sugar to make nested function calls easier to read. This is important because that pattern becomes increasingly common when dealing with iterators.

As a simple example, let's rewrite our running mean using `pipe`:

```
In [12]:
```

```
import toolz as tz
filename = 'data/expr.tsv'
mean = tz.pipe(filename, readtsv_verbose,
               add1_verbose, log_verbose, running_mean_verbose)

# This is equivalent to nesting the functions like this:
# running_mean_verbose(log_verbose(add1_verbose(readtsv_verbose(filename))))
```

```
starting running mean
starting log
starting adding 1
starting readtsv
reading line 0
adding 1 to line 0
taking log of array 0
adding line 0 to the running mean
reading line 1
adding 1 to line 1
taking log of array 1
```

```
adding line 1 to the running mean
reading line 2
adding 1 to line 2
taking log of array 2
adding line 2 to the running mean
reading line 3
adding 1 to line 3
taking log of array 3
adding line 3 to the running mean
reading line 4
adding 1 to line 4
taking log of array 4
adding line 4 to the running mean
finished readtsv
finished adding 1
finished log
returning mean
```

What was originally multiple lines, or an unwieldy mess of parentheses, is now a clean description of the sequential transformations of the input data. Much easier to understand!

This strategy also has an advantage over the original NumPy implementation: if we scale our data to millions or billions of rows, our computer might struggle to hold all the data in memory. In contrast, here we are only loading lines from disk one at a time, and maintaining a single line's worth of data.

k-mer counting and error correction

You might want to review Chapter 1 and Chapter 2 for information about DNA and genomics. Briefly, your genetic information, the blueprint for making *you*, is encoded as a sequence of chemical *bases* in your *genome*. These are really, really tiny, so you can't just look in a microscope and read them. You also can't read a long string of them: errors accumulate and the readout becomes unusable. (New technology is changing this, but here we will focus on short-read sequencing data, the most common today.) Luckily, every one of your cells has an identical copy of your genome, so what we can do is shred those copies into tiny segments (about 100 bases long), and then assemble those like an enormous puzzle of 30 million pieces.

Before performing assembly, it is vital to perform read correction. During DNA sequencing some bases are incorrectly read out, and must be fixed, or they will mess up the assembly. (Imagine having puzzle pieces with the wrong shape.)

One correction strategy is to find similar reads in your dataset and fix the error by grabbing the correct information from those reads. Or alternatively, you may choose to completely discard those reads containing errors.

However, this is a very inefficient way to do it, because finding similar reads means you would compare each read to every other read. This takes N^2 operations, or 9×10^14 for a 30 million read dataset! (And these are not cheap operations.)

There is another way. Pavel Pevzner and others realized that reads could be broken down into smaller, overlapping *k-mers*, substrings of length k, which can then be stored in a hash table (a dictionary, in Python). This has tons of advantages, but the main one is that instead of computing on the total number of reads, which can be arbitrarily large, we can compute on the total number of k-mers, which can only be as large as the genome itself — usually 1-2 orders of magnitude smaller than the reads.

If we choose a value for k that is large enough to ensure any k-mer appears only once in the genome, the number of times a k-mer appears is exactly the number of reads that originate from that part of the genome. This is called the *coverage* of that region.

If a read has an error in it, there is a high probability that the k-mers overlapping the error will be unique or close to unique in the genome. Think of the equivalent in English: if you were to take reads from Shakespeare, and one read was "to be or nob to be", the 6-mer "nob to" will appear rarely or not at all, whereas "not to" will be very frequent.

This is the basis for k-mer error correction: split the reads into k-mers, count the occurrence of each k-mer, and use some logic to replace rare k-mers in reads with similar common ones. (Or, alternatively, discard reads with erroneous k-mers. This is possible because reads are so abundant that we can afford to toss out erroneous data.)

This is also an example in which streaming is *essential*. As mentioned before, the number of reads can be enormous, so we don't want to store them in memory.

DNA sequence data is commonly represented in FASTA format. This is a plaintext format, consisting of one or many DNA sequences per file, each with a name and the actual sequence.

A sample FASTA file:

```
> sequence_name1
ACGT

> sequence_name2
GACT
```

Now we have the required information to convert a stream of lines from a FASTA file to a count of k-mers:

- filter lines so that only sequence lines are used
- for each sequence line, produce a stream of k-mers

- add each k-mer to a dictionary counter

Here's how you would do this in pure Python, using nothing but built-ins:

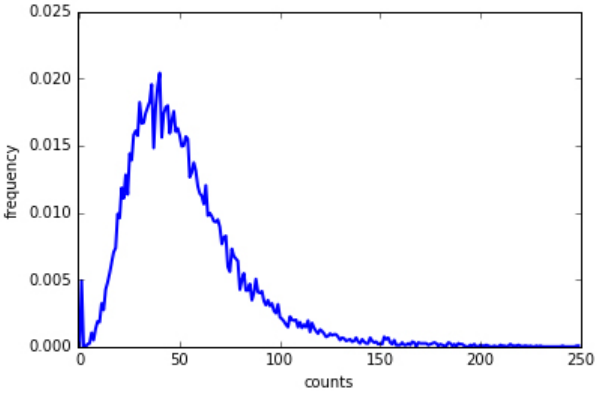
```
In [13]:
```

```
def is_sequence(line):
    line = line.rstrip()  # remove '\n' at end of line
    return len(line) > 0 and not line.startswith('>')

def reads_to_kmers(reads_iter, k=7):
    for read in reads_iter:
    for start in range(0, len(read) - k):
        yield read[start : start + k] # note yeild, so this is a generator

def kmer_counter(kmer_iter):
    counts = {}
    for kmer in kmer_iter:
        if kmer not in counts:
            counts[kmer] = 0
        counts[kmer] += 1
    return counts

with open('data/sample.fasta') as fin:
    reads = filter(is_sequence, fin)
    kmers = reads_to_kmers(reads)
    counts = kmer_counter(kmers)
```

This totally works and is streaming, so reads are loaded from disk one at a time and piped through the k-mer converter and to the k-mer counter. We can then plot a histogram of the counts, and confirm that there are indeed two well-separated populations of correct and erroneous k-mers:

```
In [14]:
```

```
from matplotlib import pyplot as plt

def integer_histogram(counts, normed=True, xlim=[], ylim=[],
                      *args, **kwargs):
    hist = np.bincount(counts)
    if normed:
        hist = hist / np.sum(hist)
    plt.plot(np.arange(hist.size), hist, *args, **kwargs)
    plt.xlabel('counts')
    plt.ylabel('frequency')
    plt.xlim(*xlim)
    plt.ylim(*ylim)

counts_arr = np.fromiter(counts.values(), dtype=int, count=len(counts))
integer_histogram(counts_arr, xlim=(-1, 250), lw=2)
```



Notice the nice distribution of k-mer frequencies, along with a big bump of k-mers (at the left of the plot) that appear only once. Such low frequency k-mers are likely to be errors.

But, with the code above, we are actually doing a bit too much work. A lot of the functionality we wrote in for loops and yields is actually *stream manipulation*: transforming a stream of data into a different kind of data, and accumulating it at the end. Toolz has a lot of stream manipulation primitives that make it easy to write the above in just one function call; and, once you know the names of the transforming functions, it also becomes easier to visualize what is happening to your data stream at each point.

For example, the *sliding window* function is exactly what we need to make k-mers:

```
In [15]:

print(tz.sliding_window.__doc__)

A sequence of overlapping subsequences

   >>> list(sliding_window(2, [1, 2, 3, 4]))
   [(1, 2), (2, 3), (3, 4)]

   This function creates a sliding window suitable for transformations like
   sliding means / smoothing

   >>> mean = lambda seq: float(sum(seq)) / len(seq)
   >>> list(map(mean, sliding_window(2, [1, 2, 3, 4])))
   [1.5, 2.5, 3.5]
```

Additionally, the *frequencies* function counts the appearance of individual items in a data stream! Together with pipe, we can now count k-mers in a single function call (though we will still use our original FASTA parsing function):

```
In [16]:
```

```
from toolz import curried as c

k = 7
counts = tz.pipe('data/sample.fasta', open, c.filter(is_sequence),
                 c.map(str.rstrip),
                 c.map(c.sliding_window(k)),
                 tz.concat, c.map(''.join),
                 tz.frequencies)
```

We neglected to discuss the *curried* part of this approach.

"Currying" means *partially* evaluating a function and returning another, "smaller" function. Normally in Python if you don't give a function all of its required arguments then it will throw a fit. In contrast, a curried function can just take *some* of those arguments. If the curried function doesn't get enough arguments, it returns a new function that takes the leftover arguments. Once that second function is called with the remaining arguments it can perform the original task. Another word for currying is partial evaluation. In functional programming, currying is a way to produce a function that can wait for the rest of the arguments to show up later.

Currying is not named after the spice blend (though it does spice up your code). It is named for Haskell Curry, the mathematician who invented the concept. Haskell Curry is also the namesake of the Haskell programming language, which has functions curried by default!

It turns out that having a function that already knows about some of the arguments is perfect for streaming! We've seen a hint of how powerful currying and pipes can be together in the above code snippet, but we will elaborate further in what follows.

Currying can be a bit of a mind-bend when you first start, so let's make our own curried function to see how it works. Let's start by writing a simple, non-curried function:

```
In [17]:

def add(a, b):
    return a + b

add(2, 5)

Out[17]:
7
```

Now we write a similar function which we curry manually:

```
In [18]:

def add_curried(a, b=None):
    if b is None:
        # second argument not given, so make a function and return it
        def add_partial(b):
            return add(a, b)
```

```
        return add_partial
    else:
        # Both values were given, so we can just return a value
        return add(a, b)
```

Now let's try out a curried function to make sure it does what we expect.

```
In [19]:

add_curried(2, 5)

Out[19]:
7
```

Okay, it acts like a normal function when given both variables. Now let's leave out the second variable.

```
In [20]:

add_curried(2)

Out[20]:

<function __main__.add_curried.<locals>.add_partial>
```

It returned a function. Yay! Now let's see if we can use that returned function as expected.

```
In [21]:

partial_sum = add_curried(2)
partial_sum(5)

Out[21]:
7
```

Now that worked, but add_curried was a reasonably hard function to read. Future me will probably have trouble remembering how I wrote that code. Luckily, Toolz has some syntactic sugar to help us out.

```
In [22]:

import toolz as tz

@tz.curry # Use curry as a decorator
def add(x, y):
    return x + y

add_partial = add(2)
add_partial(5)

Out[22]:

7
```

To summarize what we did, `add` is now a curried function, so it can take one of the arguments and returns another function, `add_partial`, which "remembers" that argument.

In fact, all of the Toolz functions are also available as curried functions in the toolz.curried namespace. Toolz also includes curried version of some handy higher order Python functions like `map`, `filter` and `reduce`. We will import the `curried` namespace as `c` so our code doesn't get too cluttered. So for example the curried version of `map` will be `c.map`. Note, that the curried functions (e.g. `c.map`) are different from the `@curry` decorator, which is used to create a curried function.

```
In [23]:
```

```
from toolz import curried as c
c.map
```

```
Out[23]:

<class 'map'>
```

As a reminder, `map` is a built-in function. From the docs:

> *map(function, iterable, ...) Return an iterator that applies function to every item of iterable, yielding the results.*

A curried version of `map` is particularly handy when working in a Toolz pipe. You can just pass a function to `c.map` and then stream in the iterator later using `tz.pipe`. Take another look at our function for reading in the genome to see how this works in practice.

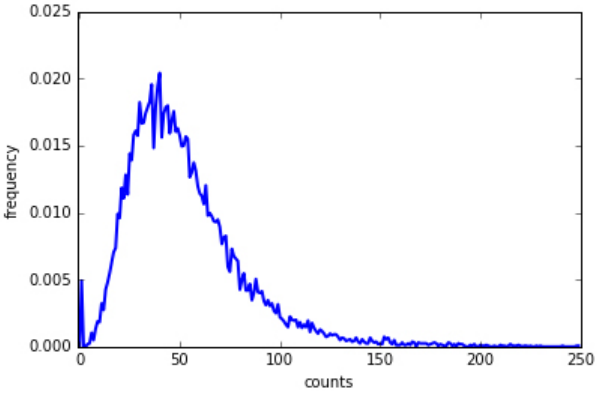
```
In [24]:
```

```
def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted,  # Filenames
                   c.map(open),  # lines
                   # concatenate lines from all files:
                   tz.concat,
                   # drop header from each sequence
                   c.filter(is_sequence),
                   # concatenate characters from all lines
                   tz.concat,
                   # discard newlines and 'N'
                   c.filter(is_nucleotide))
```

Okay, so now we've got our heads around curried, let's get back to our k-mer counting code. We can now observe the frequency of different k-mers:

```
In [25]:
```

```
counts = np.fromiter(counts.values(), dtype=int, count=len(counts))
integer_histogram(counts, xlim=(-1, 250), lw=2)
```



tips {.callout}

- (list of list -> list) with tz.concat
- don't get caught out:
  - iterators get consumed. So if you make a generator object and do some processing on it, and then a later step fails, you need to recreate the generator. The original is already gone.
  - iterators are lazy. You need to force evaluation sometimes.
- when you have lots of functions in a pipe, it's sometimes hard to figure out where things go wrong. Take a small stream and add functions to your pipe one by one from the first/leftmost until you find the broken one.

Exercise: The scikit-learn library has an IncrementalPCA class, which allows you to run principal components analysis on a dataset without loading the full dataset into memory. But you need to chunk your data yourself, which makes the code a bit awkward to use. Make a function that can take a stream of data samples and perform PCA. Then, use the function to compute the PCA of the `iris` machine learning dataset, which is in `data/iris.csv`. (You can also access it from the `datasets` module of scikit-learn.)

Hint: The `IncrementalPCA` class is in `sklearn.decomposition`, and requires a *batch size* greater than 1 to train the model. Look at the `toolz.curried.partition` function for how to create a stream of batches from a stream of data points.

Markov model from a full genome

Back to our original code example. What is a Markov model, and why is it useful?

In general, a Markov model assumes that the probability of the system moving to a given state, is only dependent on the state that it was in just previously. For example if it is sunny right now, there is a high probability that it will be sunny tomorrow. The fact that it was raining yesterday is irrelevant. In this theory, all the information required to predict the future is encoded in the current state of things. The past is irrelevant. This assumption is useful for simplifying otherwise intractable problems, and often gives good results. Markov models are behind much of the signal processing in mobile phone and satellite communications, for example.

In the context of genomics, as we will see, different functional regions of a genome have different *transition probabilities* between similar states. Observing these in a new genome, we can predict something about the function of those regions. Going back to the weather analogy, the probability of going from a sunny day to a rainy day is very different depending on whether you are in Los Angeles or London. Therefore, if I give you a string of (sunny, sunny, sunny, rainy, sunny, ...) days, you can predict whether it came from Los Angeles or London, assuming you have a previously trained model.

In this chapter, we'll cover just the model building, for now.

- You can download the *Drosophila melanogaster* (fruit fly) genome file dm6.fa.gz from http://hgdownload.cse.ucsc.edu/goldenPath/dm6/bigZips/. You will need to unzip it using: `gzip -d dm6.fa.gz`

In the genome data, genetic sequence, which consists of the letters A, C, G, and T, is encoded as belonging to *repetitive elements*, a specific class of DNA, by whether it is in lower case (repetitive) or upper case (non-repetitive). We can use this information when we build the Markov model.

We want to encode the Markov model as a NumPy array, so we will make dictionaries to index from letters to indices in [0, 7] (`LDICT` for "letters dictionary"), and from pairs of letters to 2D indices in ([0, 7], [0, 7]) (`PDICT` or "pairs dictionary"):

```
In [31]:

import itertools as it

LDICT = dict(zip('ACGTacgt', range(8)))
PDICT = {(a, b): (LDICT[a], LDICT[b])
         for a, b in it.product(LDICT, LDICT)}
```

We also want to filter out non-sequence data: the sequence names, which are in lines starting with `>`, and unknown sequence, which is labeled as `N`, so we will make functions to filter on:

```
In [32]:

def is_sequence(line):

return not line.startswith('>')
def is_nucleotide(letter):
    return letter in LDICT  # ignore 'N'
```

Finally, whenever we get a new nucleotide pair, say, ('A', 'T'), we want to increment our Markov model (our NumPy matrix) at the corresponding position. We make a curried function to do so:

```
In [33]:

import toolz as tz

@tz.curry
def increment_model(model, index):
    model[index] += 1
```

We can now combine these elements to stream a genome into our NumPy matrix. Note that, if `seq` below is a stream, we never need to store the whole genome, or even a big chunk of the genome, in memory!

```
In [34]:

from toolz import curried as c

def markov(seq):
    """Get a 1st-order Markov model from a sequence of nucleotides."""
    model = np.zeros((8, 8))
    tz.last(tz.pipe(seq,
                    c.sliding_window(2),        # each successive tuple
                    c.map(PDICT.__getitem__),   # location in matrix of tuple
                    c.map(increment_model(model))))  # increment matrix
    # convert counts to transition probability matrix
    model /= np.sum(model, axis=1)[:, np.newaxis]
    return model
```

Now we simply need to produce that genome stream, and make our Markov model:

```
In [35]:

from glob import glob

def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted, # Filenames
                   c.map(open),  # lines
                   # concatenate lines from all files:
```

```
                    tz.concat,
                    # drop header from each sequence
                    c.filter(is_sequence),
                    # concatenate characters from all lines
                    tz.concat,
                    # discard newlines and 'N'
                    c.filter(is_nucleotide))
```

Let's try it out on the Drosophila (fruit fly) genome:

```
In [36]:

# dm6.fa.gz can be downloaded from ftp://hgdownload.cse.ucsc.edu/goldenPath/dm6/bigZips/
# Unzip before using: gzip -d dm6.fa.gz
dm = 'data/dm6.fa'
model = tz.pipe(dm, genome, c.take(1000000), markov)
# we use `take` to just run on the first 1000000 bases, to speed things up.
# the take step can just be removed if you have ~5-10 mins to wait.
```

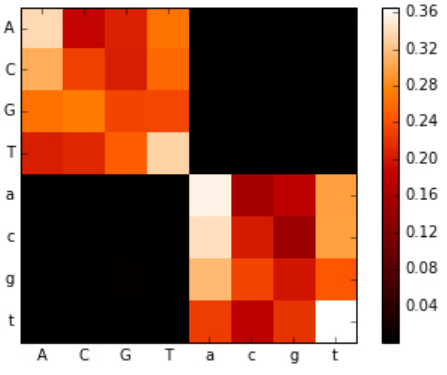
Let's look at the resulting matrix:

```
In [37]:

print(' ', '      '.join('ACGTacgt'), '\n')
print(model)
```

```
     A C G T a c g t
[[ 0.338  0.187  0.208  0.266  0.     0.     0.     0.   ]
 [ 0.307  0.229  0.205  0.258  0.     0.     0.     0.   ]
 [ 0.263  0.271  0.232  0.233  0.     0.     0.     0.   ]
 [ 0.205  0.212  0.249  0.334  0.     0.     0.     0.   ]
 [ 0.002  0.002  0.002  0.002  0.356  0.158  0.18   0.297]
 [ 0.002  0.002  0.002  0.002  0.342  0.202  0.149  0.298]
 [ 0.002  0.002  0.003  0.002  0.315  0.232  0.2    0.244]
 [ 0.001  0.002  0.002  0.002  0.227  0.18   0.22   0.366]]
```

It's probably clearer to look at the result as an image:

```
In [38]:

plt.imshow(model, cmap='gist_heat', interpolation='nearest');
plt.colorbar();
ax = plt.gca()
ax.set_xticklabels(' ACGTacgt');
ax.set_yticklabels(' ACGTacgt');
```



Notice how the G-A and G-C transitions are different between the repeat and non-repeat parts of the genome. This information can be used to classify previously unseen DNA sequence.

Challenge: add a step to the start of the pipe to unzip the data so you don't have to keep a decompressed version on your hard drive. Yes, unzip can be streamed, too!

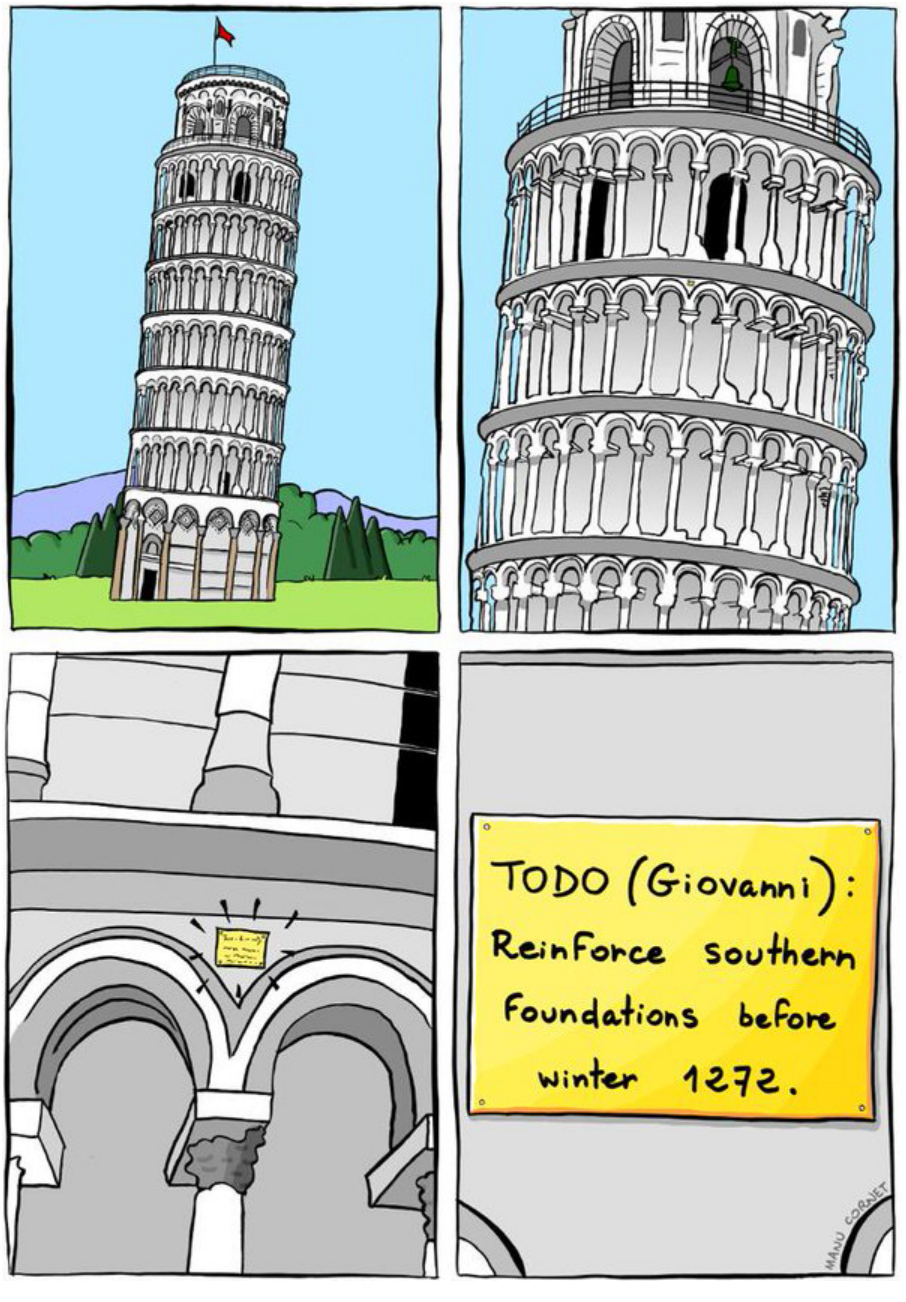
Hint: The `tarfile` package, part of Python's standard library, allows you to open tar files (even compressed ones) as if they were normal files.

Conclusions

We hope to have shown you at least a hint that streaming in Python can be easy when you use a few abstractions, like the ones Toolz provides.

Streaming can make you more productive, because big data takes linearly longer than small data. In batch analysis, big data can take forever to run, because the operating system has to keep transferring data from RAM to the hard disk and back. Or, Python might refuse altogether and simply show a `MemoryError`! This means that, for many analyses, you don't need a bigger machine to analyse bigger datasets. And, if your tests pass on small data, they'll pass on big data, too!

Our take home message from this chapter is this: when writing an algorithm, or analysis, think about whether you can do it streaming. If you can, just do it from the beginning. Your future self will thank you. Doing it later is harder, and results in things like this:



(Comic by Manu Cornet, used with permission.)

File: html/ch2.html

Exercise: Do our clusters do a better job of predicting survival than the original clusters in the paper? What about UV signature? Plot survival curves using the original clusters and UV signature columns of the patient data. How do they compare to our clusters?

Exercise: We leave you the exercise of implementing the approach described in the paper:

1. Take bootstrap samples (random choice with replacement) of the genes used to cluster the samples;
2. For each sample, produce a hierarchical clustering;
3. In a `(n_samples, n_samples)` -shaped matrix, store the number of times a sample pair appears together in a bootstrapped clustering.
4. Perform a hierarchical clustering on the resulting matrix.

This identifies groups of samples that frequently occur together in clusterings, regardless of the genes chosen. Thus, these samples can be considered to robustly cluster together.

Hint: use `np.random.choice` with `replacement=True` to create bootstrap samples of row indices.

File: html/ch3.html

Exercise: We just saw how to select a square and paint it red. Can you extend that to other shapes and colors? Create a function to draw a blue grid onto a color image,

and apply it to the `astronaut` image of Eileen Collins (above). Your function should take two parameters: the input image, and the grid spacing. Use the following template to help you get started.

```
In [9]:
```

```
def overlay_grid(image, spacing=128):
    """Return an image with a grid overlay, using the provided spacing.

    Parameters
    ----------
    image : array, shape (M, N, 3)
        The input image.
    spacing : int
        The spacing between the grid lines.

    Returns
    -------
    image_gridded : array, shape (M, N, 3)
        The original image with a blue grid superimposed.
    """
    image_gridded = image.copy()
    pass  # replace this line with your code...
    return image_gridded

# plt.imshow(overlay_grid(astro, 128));  # ... and uncomment this line to test your function
```

Exercise: Conway's Game of Life.

Suggested by Nicolas Rougier.

Conway's Game of Life is a seemingly simple construct in which "cells" on a regular square grid live or die according to the cells in their immediate surroundings. At every timestep, we determine the state of position (i, j) according to its previous state and that of its 8 neighbors (above, below, left, right, and diagonals):

- a live cell with only one live neighbor or none dies.
- a live cell with two or three live neighbors lives on for another generation.
- a live cell with four or more live neighbors dies, as if from overpopulation.
- a dead cell with exactly three live neighbors becomes alive, as if by reproduction.

Although the rules sound like a contrived math problem, they in fact give rise to incredible patterns, starting with gliders (small patterns of live cells that slowly move in each generation) and glider guns (stationary patterns that sprout off gliders), all the way up to prime number generator machines (see, for example, this page), and even simulating Game of Life itself!

Can you implement the Game of Life using `ndi.generic_filter`?

Solution:

Code by: Nicolas Rougier (@rougier)

This is the game of life (cellular automata) in 10 lines of python with numpy. The code is available from http://www.labri.fr/perso/nrougier/teaching/numpy.100/ (last question). (code explanations is available from http://www.labri.fr/perso/nrougier/teaching/numpy/numpy.html)

Exercise: Use `scipy.optimize.curve_fit` to fit the tail of the in-degree survival function to a power-law, $f(d) \sim d^{-\gamma}, d > d_0$, for $d_0 = 10$ (the red line in Figure 6B of the paper), and modify the plot to include that line.

Exercise: Sobel gradient magnitude.

Above, we saw how we can combine the output of two different filters, the horizontal Sobel filter, and the vertical one. Can you write a function that does this in a single pass using `ndi.generic_filter`?

Exercise: Use `scipy.optimize.curve_fit` to fit the tail of the in-degree survival function to a power-law, $f(d) \sim d^{-\gamma}, d > d_0$, for $d_0 = 10$ (the red line in Figure 6B of the paper), and modify the plot to include that line.

File: html/ch4.html

Exercise: The FFT is often used to speed up image convolution (convolution is the application of a moving filter mask). Convolve an image with `np.ones((5, 5))`, using a) numpy's `np.convolve` and b) `np.fft.fft2`. Confirm that the results are identical.

Hints:

- The convolution of `x` and `y` is equivalent to `ifft2(X * Y)`, where `X` and `Y` are the FFTs of x and y respectively.
- In order to multiply `X` and `Y`, they have to be the same size. Use `np.pad` to extend `x` and `y` with zeros (toward the right and bottom) *before* taking their FFT.
- You may see some edge effects. These can be removed by increasing the padding size, so that both `x` and `y` have dimensions `shape(x) + shape(y) - 1`.

```
In [36]:

from scipy import signal

x = np.random.random((50, 50))
y = np.ones((5, 5))

L = x.shape[0] + y.shape[0] - 1
```

```
Px = L - x.shape[0]
Py = L - y.shape[0]

xx = np.pad(x, ((0, Px), (0, Px)), mode='constant')
yy = np.pad(y, ((0, Py), (0, Py)), mode='constant')

zz = np.fft.ifft2(np.fft.fft2(xx) * np.fft.fft2(yy)).real
print('Resulting shape:', zz.shape, ' <-- Why?')

z = signal.convolve2d(x, y)

print('Results are equal?', np.allclose(zz, z))
```

```
Resulting shape: (54, 54)  <-- Why?
Results are equal? True
```

File: html/ch5.html

Question: Why did we call this inefficient?

Answer: From Chapter 1, you recall that `arr == k` creates an array of Boolean (`True` or `False`) values of the same size as `arr`. This, as you might expect, requires a full pass over `arr`. Therefore, in the above solution, we make a full pass over each of `pred` and `gt` for every combination of values in `pred` and `gt`. In principle, we can compute `cont` using just a single pass over both arrays, so these multiple passes are inefficient.

Exercise: Write an alternative way of computing the confusion matrix that only makes a single pass through `pred` and `gt`.

```
In [7]:
```

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    # your code goes here
    return cont
```

We offer two solutions here, although many are possible.

Our first solution uses Python's built-in `zip` function to pair together labels from `pred` and `gt`.

```
In [8]:
```

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    for i, j in zip(pred, gt):
        cont[i, j] += 1
    return cont
```

Our second solution is to iterate over all possible indices of `pred` and `gt` and manually grab the corresponding value from each array:

```
In [9]:
```

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    for idx in range(len(pred)):
        i = pred[idx]
        j = gt[idx]
        cont[i, j] += 1
    return cont
```

The first option would be considered the more "Pythonic" of the two, but the second one is easier to speed up by translating and compiling in languages or tools such as C, Cython, and Numba (which are a topic for another book).

Exercise: Write a function to compute the confusion matrix in one pass, as above, but instead of assuming two categories, infer the number of categories from the input.

```
In [10]:
```

```
def general_confusion_matrix(pred, gt):
    n_classes = None  # replace `None` with something useful
    # your code goes here
    return cont
```

We merely need to make an initial pass through both input arrays to determine the maximum label. We then add 1 to it to account for the zero label and Python's 0-based indexing. We then create the matrix and fill it in the same way as above:

```
In [11]:
```

```
def general_confusion_matrix(pred, gt):
    n_classes = max(np.max(pred), np.max(gt)) + 1
    cont = np.zeros((n_classes, n_classes))
    for i, j in zip(pred, gt):
        cont[i, j] += 1
    return cont
```

Exercise: write out the COO representation of the following matrix:

```
In [15]:
```

```
s2 = np.array([[0, 0, 6, 0, 0],
               [1, 2, 0, 4, 5],
               [0, 1, 0, 0, 0],
               [9, 0, 0, 0, 0],
               [0, 0, 0, 6, 7]])
```

We first list the non-zero elements of the array, left-to-right and top-to-bottom, as if reading a book:

```
In [16]:
```

```
s2_data = np.array([6, 1, 2, 4, 5, 1, 9, 6, 7])
```

We then list the row indices of those values in the same order:

```
In [17]:
```

```
s2_row = np.array([0, 1, 1, 1, 1, 2, 3, 4, 4])
```

And finally the column indices:

```
In [18]:

s2_col = np.array([2, 0, 1, 3, 4, 1, 0, 3, 4])
```

We can easily check that these produce the right matrix, by checking equality in both directions:

```
In [19]:

s2_coo0 = sparse.coo_matrix(s2)
print(s2_coo0.data)
print(s2_coo0.row)
print(s2_coo0.col)

[6 1 2 4 5 1 9 6 7]
[0 1 1 1 1 2 3 4 4]
[2 0 1 3 4 1 0 3 4]
```

and:

```
In [20]:

s2_coo1 = sparse.coo_matrix((s2_data, (s2_row, s2_col)))
print(s2_coo1.todense())

[[0 0 6 0 0]
 [1 2 0 4 5]
 [0 1 0 0 0]
 [9 0 0 0 0]
 [0 0 0 6 7]]
```

Exercise: The rotation happens around the origin, coordinate (0, 0). But can you rotate the image around its center?

Hint: The transformation matrix for a *translation*, i.e. sliding the image up/down or left/right, is given by: $H_{tr} = \begin{bmatrix} 1 & 0 & t_r \\ 0 & 1 & t_c \\ 0 & 0 & 1 \end{bmatrix}$ when you want to move the image t_r pixels down and t_c pixels right.

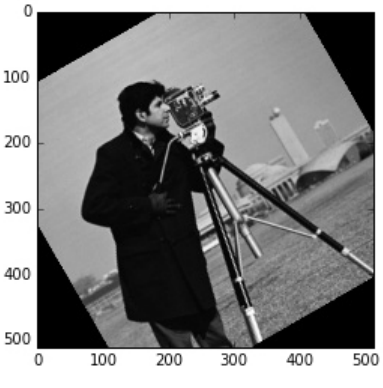
We can *compose* transformations by multiplying them. We know how to rotate an image about the origin, as well as how to slide it around. So what we will do is slide the image so that the center is at the origin, rotate it, and then slide it back.

```
In [31]:

def transform_rotate_about_center(shape, degrees):
    """Return the homography matrix for a rotation about an image center."""
    c = np.cos(np.deg2rad(angle))
    s = np.sin(np.deg2rad(angle))
```

```
    H_rot = np.array([[c, -s,  0],
                      [s,  c,  0],
                      [0,  0,  1]])
    # compute image center coordinates
    center = np.array(image.shape) / 2
    # matrix to center image on origin
    H_tr0 = np.array([[1, 0, -center[0]],
                      [0, 1, -center[1]],
                      [0, 0, 1]])
    # matrix to move center back
    H_tr1 = np.array([[1, 0, center[0]],
                      [0, 1, center[1]],
                      [0, 0,          1]])
    # complete transformation matrix
    H_rot_cent = H_tr1 @ H_rot @ H_tr0

    sparse_op = homography(H_rot_cent, image.shape)

    return sparse_op
```

We can test that this works:

```
In [32]:

tf = transform_rotate_about_center(image.shape, 30)
plt.imshow(apply_transform(image, tf), cmap='gray')

Out[32]:

<matplotlib.image.AxesImage at 0x11223b8d0>
```



Exercise: Remember from Chapter 1 that NumPy has built-in tools for repeating arrays using *broadcasting*. How can you reduce the memory footprint required for the contingency matrix computation?

Hint: Look at the documentation for the function `np.broadcast_to`.

The `np.ones` array that we create is read-only: it will only be used as the values to sum by `coo_matrix`. We can use `broadcast_to` to create a similar array with only one element, "virtually" repeated n times:

```
In [40]:

def confusion_matrix(pred, gt):
    n = pred.size
    ones = np.broadcast_to(1., n)  # virtual array of 1s of size n
    cont = sparse.coo_matrix((ones, (pred, gt)))
    return cont
```

Let's make sure it still works as expected:

```
In [41]:

cont = confusion_matrix(pred, gt)
print(cont.todense())
```

```
[[ 3.  1.]
 [ 2.  4.]]
```

Boom. Instead of making an array as big as the original data, we just make one of size 1.

Exercise: Compute the conditional entropy of month given rain. What is the entropy of the month variable? (Ignore the different number of days in a month.) Which one is greater? (*Hint:* the probabilities in the table are the conditional probabilities of rain given month.)

```
In [47]:

prains = [25, 27, 24, 18, 14, 11, 7, 8, 10, 15, 18, 23]
prains = [p / 100 for p in prains]
pshine = [1 - p for p in prains]
p_rain_g_month = np.array((prains, pshine)).T
# replace 'None' below with expression for non-conditional contingency
# table. Hint: the values in the table must sum to 1.
p_rain_month = None
# Add your code below to compute H(M|R) and H(M)
```

To obtain the joint probability table, we simply divide the table by its total, in this case, 12:

```
In [48]:

print('table total:', np.sum(p_rain_g_month))
p_rain_month = p_rain_g_month / np.sum(p_rain_g_month)
```

```
table total: 12.0
```

Now we can compute the conditional entropy of the month given rain. (This is like asking: if we know it's raining, how much more information do we need to know to figure out what month it is, on average?)

```
In [49]:

p_rain = np.sum(p_rain_month, axis=0)
p_month_g_rain = p_rain_month / p_rain
Hmr = np.sum(p_rain * p_month_g_rain * np.log2(1 / p_month_g_rain))
print(Hmr)

3.5613602411
```

Let's compare that to the entropy of the months:

```
In [50]:

p_month = np.sum(p_rain_month, axis=1) # 1/12, but this method is more general
Hm = np.sum(p_month * np.log2(1 / p_month))
print(Hm)

3.58496250072
```

So we can see that knowing whether it rained today got us 2 hundredths of a bit closer to guessing what month it is! Don't bet the farm on that guess.

Exercise: Segmentation in practice Try finding the best threshold for a selection of other images from the Berkeley Segmentation Dataset and Benchmark. Using the mean or median of those thresholds, then go and segment a new image. Did you get a reasonable segmentation?

File: html/ch6.html

Exercise: Consider the rotation matrix $R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$ When R is multiplied with a 3-dimensional column-vector $p = [x\,y\,z]^T$, the resulting vector Rp is rotated by θ degrees around the z-axis.

1. For $\theta = 45^\circ$, verify (by testing on a few arbitrary vectors) that R rotates these vectors around the z axis.
2. Now, verify that multiplying by R leaves the vector $[0\,0\,1]^T$ unchanged. In other words, $Rp = 1p$, which means p is an eigenvector of R with eigenvalue 1.

Solution:

1. As an example, consider a 3x3 rotation matrix R that, when multiplied by any 3-dimensional vector p, rotates it 30° degrees around the z-axis. R will rotate all vectors except for those that lie *on* the z-axis. For those, we'll see no effect, or $Rp = p$, i.e. $Rp = \lambda p$ with eigenvalue $\lambda = 1$.

   ```
   In [2]:
   ```

```
import numpy as np

theta = np.deg2rad(45)
R = np.array([[np.cos(theta), -np.sin(theta), 0],
              [np.sin(theta),  np.cos(theta), 0],
              [0,              0,             1]])
print("R @ x-axis:", R @ [1, 0, 0])
print("R @ y-axis:", R @ [0, 1, 0])
print("R @ z-axis:", R @ [0, 0, 1])
```

```
R @ x-axis: [ 0.70710678 0.70710678 0.         ]
R @ y-axis: [-0.70710678 0.70710678 0.         ]
R @ z-axis: [ 0.  0.  1.]
```

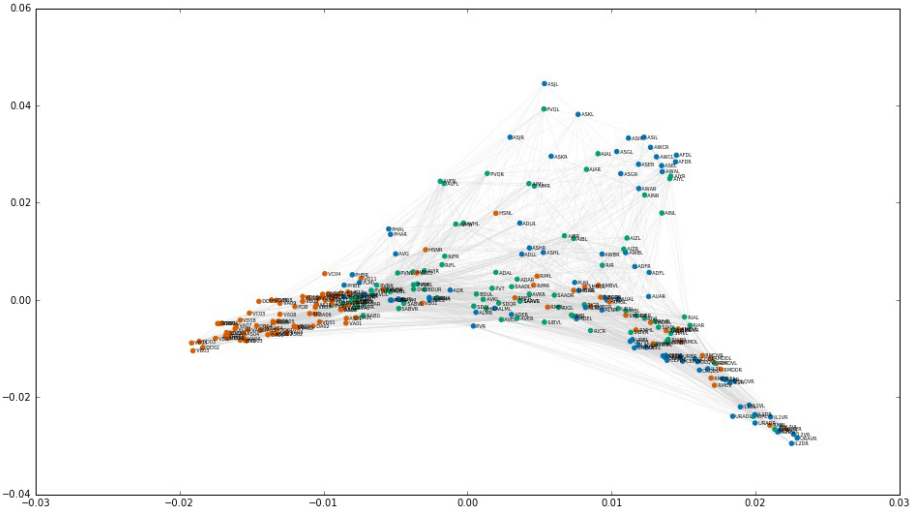
R rotates both the x and y axes, but not the z-axis.

Exercise: How do you modify the above code to show the affinity view in Figure 2B from the paper?

Solution: In the affinity view, instead of using the processing depth on the y-axis, we use the normalized third eigenvector of Q, just like we did with x:

```
In [24]:
```

```
y = Dinv2 @ eigvecs[:, 2]
plt.figure(figsize=(16, 9))
plot_connectome(x, y, C, neuron_ids, neuron_types)
```



Challenge: linear algebra with sparse matrices

The above code uses numpy arrays to hold the matrix and perform the necessary computations. Because we are using a small graph of fewer than 300 nodes, this is feasible. However, for larger graphs, it would fail.

In what follows, we will analyze the dependency graph for packages in the Python Package Index, or PyPI, which contains over 75 thousand packages. To hold the Laplacian matrix for this graph would take up $8\left(75 \times 10^3\right)^2 = 45 \times 10^9$ bytes, or 45GB, of RAM. If you add to that the adjacency, symmetric adjacency, pseudoinverse, and, say, two temporary matrices used during calculations, you climb up to 270GB, beyond the reach of most desktop computers.

"Ha!", some of you might be thinking. "Ha! My desktop has 512GB of RAM! It would make short work of this so-called 'large' graph!"

Perhaps. But we will also be analysing the Association for Computing Machinery (ACM) citation graph, a network of over two million scholarly works and references. *That* Laplacian would take up 32 terabytes of RAM.

However, we know that the dependency and reference graphs are *sparse*: packages usually depend on just a few other packages, not on the whole of PyPI. And papers and books usually only reference a few others, too. So we can hold the above matrices using the sparse data structures from `scipy.sparse` (see Chapter 5), and use the linear algebra functions in `scipy.sparse.linalg` to compute the values we need.

Try to explore the documentation in `scipy.sparse.linalg` to come up with a sparse version of the above computation.

Hint: the pseudoinverse of a sparse matrix is, in general, not sparse, so you can't use it here. Similarly, you can't get all the eigenvectors of a sparse matrix, because they would together make up a dense matrix.

You'll find parts of the solution below (and of course in the solutions chapter), but we highly recommend that you try it out on your own.

Challenge accepted

For the purposes of this challenge, we are going to use the small connectome above, because it's easier to visualise what is going on. In later parts of the challenge we'll use these techniques to analyze larger networks.

First, we start with the adjacency matrix, A, in a sparse matrix format, in this case, CSR, which is the most common format for linear algebra. We'll append `s` to the names of all the matrices to indicate that they are sparse.

```
In [25]:
```

```
from scipy import sparse
import scipy.sparse.linalg

As = sparse.csr_matrix(A)
```

We can create our connectivity matrix in much the same way:

```
In [26]:

Cs = (As + As.T) / 2
```

In order to get the degrees matrix, we can use the "diags" sparse format, which stores diagonal and off-diagonal matrices.

```
In [27]:

degrees = np.ravel(Cs.sum(axis=0))
Ds = sparse.diags(degrees, 0)
```

Getting the Laplacian is straightforward:

```
In [28]:

Ls = Ds - Cs
```

Now we want to get the processing depth. Remember that getting the pseudo-inverse of the Laplacian matrix is out of the question, because it will be a dense matrix (the inverse of a sparse matrix is not generally sparse itself). However, we were actually using the pseudo-inverse to compute a vector z that would satisfy $Lz = b$, where $b = C \odot \text{sign}\left(A - A^T\right)\mathbb{1}$. (You can see this in the supplementary material for Varshney *et al.*) With dense matrices, we can simply use $z = L^{+}b$. With sparse ones, though, we can use one of the *solvers* in `sparse.linalg.isolve` to get the `z` vector after providing `L` and `b`, no inversion required!

```
In [29]:

b = Cs.multiply((As - As.T).sign()).sum(axis=1)
z, error = sparse.linalg.isolve.cg(Ls, b, maxiter=10000)
```

Finally, we must find the eigenvectors of Q, the degree-normalized Laplacian, corresponding to its second and third smallest eigenvalues.

You might recall from Chapter 5 that the numerical data in sparse matrices is in the `.data` attribute. We use that to invert the degrees matrix:

```
In [30]:

Dsinv2 = Ds.copy()
Dsinv2.data = Ds.data ** (-0.5)
```

Finally, we use SciPy's sparse linear algebra functions to find the desired eigenvectors. The Q matrix is symmetric, so we can use the `eigsh` function, specialized for symmetric matrices, to compute them. We use the `which` keyword argument to specify

that we want the eigenvectors corresponding to the smallest eigenvalues, and `k` to specify that we need the 3 smallest:

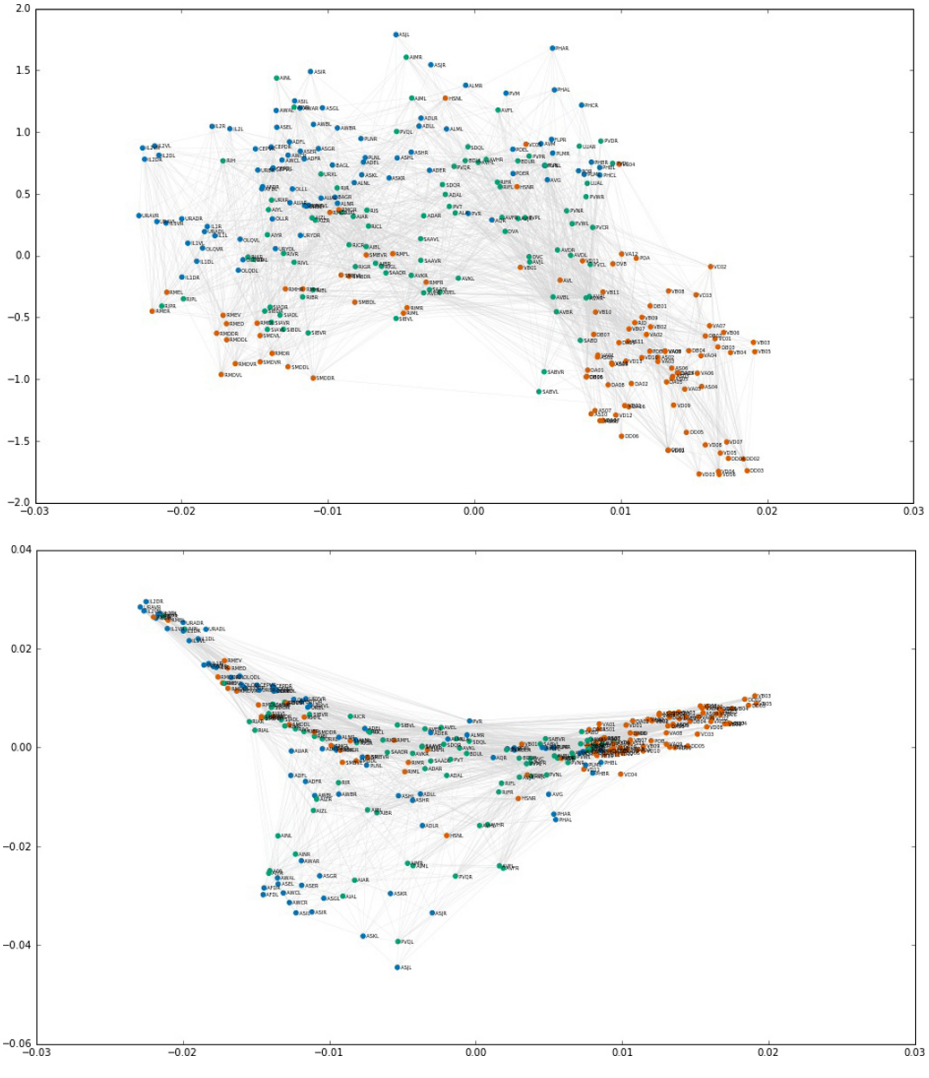
```
In [31]:

Qs = Dsinv2 @ Ls @ Dsinv2
eigvals, eigvecs = sparse.linalg.eigsh(Qs, k=3, which='SM')
sorted_indices = np.argsort(eigvals)
eigvecs = eigvecs[:, sorted_indices]
```

Finally, we normalize the eigenvectors to get the x and y coordinates:

```
In [32]:

_dsinv, x, y = (Dsinv2 @ eigvecs).T
```

(Note that the eigenvector corresponding to the smallest eigenvalue is always a vector of all ones, which we're not interested in.) We can now reproduce the above plots!

```
In [33]:

plt.figure(figsize=(16, 9))
plot_connectome(x, z, C, neuron_ids, neuron_types)

plt.figure(figsize=(16, 9))
plot_connectome(x, y, C, neuron_ids, neuron_types)
```



Note that eigenvectors are defined only up to a (possibly negative) multiplicative constant, so the plots may have ended up reversed! (That is, left is right, or up is down, or both!)

Exercise: In the above iteration, note that `Trans` is *not* column-stochastic, so the vector gets shrunk at each iteration. In order to make the matrix stochastic, we have to replace every zero-column by a column of all $1/n$. This is too expensive, but computing the iteration is cheaper. How can you modify the code above to ensure that r remains a probability vector throughout?

Solution: In order to have a stochastic matrix, all columns of the transition matrix must sum to 1. This is not satisfied when a package doesn't have any dependencies: that column will consist of all zeroes. Replacing all those columns by $1/n\mathbf{1}$, however, would be expensive.

The key is to realise that *every row* will contribute the *same amount* to the multiplication of the transition matrix by the current probability vector. That is to say, adding these columns will add a single value to the result of the iteration multiplication. What value? $1/n$ times the elements of r that correspond to a dangling node. This can be expressed as a dot-product of a vector containing $1/n$ for positions corresponding to dangling nodes, and zero elswhere, with the vector r for the current iteration.

```
In [42]:
```

```
def power2(Trans, damping=0.85, max_iter=int(1e5)):
    n = Trans.shape[0]
    is_dangling = np.ravel(Trans.sum(axis=0) == 0)
    dangling = np.zeros(n)
    dangling[is_dangling] = 1 / n
    r0 = np.ones(n) / n
    r = r0
    for _ in range(max_iter):
        rnext = (damping * (Trans @ r + dangling @ r) +
                 (1 - damping) / n)
        if np.allclose(rnext, r):
            return rnext
        else:
            r = rnext
    return r
```

Try this out manually for a few iterations. Notice that if you start with a stochastic vector (a vector whose elements all sum to 1), the next vector will still be a stochastic vector. Thus, the output pagerank from this function will be a true probability vector, and the values will represent the probability that we end up at a particular species when following links in the food chain.

Exercise: Verify that these three methods all give the same ranking for the nodes. `numpy.corrcoef` might be a useful function for this.

Solution: `np.corrcoef` gives the Pearson correlation coefficient between all pairs of a list of vectors. This coefficient will be equal to 1 if and

only if two vectors are scalar multiples of each other. Therefore, a correlation coefficient of 1 is sufficient to show that the above methods produce the same ranking.

```
In [43]:
```

```
pagerank_power = power(Trans)
pagerank_power2 = power2(Trans)
np.corrcoef([pagerank, pagerank_power, pagerank_power2])
```

```
converged

Out[43]:

array([[ 1.        ,  0.99807154, 0.9980713 ],
       [ 0.99807154,  1.        , 1.        ],
       [ 0.9980713 ,  1.        , 1.        ]])
```

Exercise: While we were writing this chapter, we started out by computing pagerank on the graph of Python dependencies. We eventually found that we could not get nice results with this graph. The correlation between in-degree and pagerank was much higher than in other datasets, and the few outliers didn't make much sense to us.

Can you think of three reasons why pagerank might not be the best measure of importance for the Python dependency graph?

Solution:

Here's our theories. Yours might be different!

First, the graph of dependencies is fundamentally different to the web. In the web, important pages reinforce each other: the Wikipedia pages for Newton's theory of gravitation and Einstein's theory of general relativity link to each other. Thus, our hypothetical Webster has some probability of bouncing between them. In contrast, Python dependencies form a directed acyclic graph (DAG). Whenever Debbie (our hypothetical dependency browser) arrives at a foundational package such as NumPy, she is instantly transported to a random package, instead of staying in the field of similar packages.

Second, the DAG of dependencies is shallow: libraries will depend on, for example, scikit-learn, which depends on scipy, which depends on numpy, and that's it. Therefore, there is not much room for important packages to link to other important packages. The hierarchy is flat, and the importance is captured by the direct dependencies.

Third, scientific programming itself is particularly prone to that flat hierarchy problem, more so than e.g. web frameworks, because scientists are doing the programming, rather than professional coders. In particular, a scientific analysis might end up in a script attached to a paper, or a Jupyter notebook, rather than another library on PyPI. We hope that this book will encourage more scientists to write great code that others can reuse, and put it on PyPI!

File: html/ch8.html

Exercise: The scikit-learn library has an IncrementalPCA class, which allows you to run principal components analysis on a dataset without loading the full dataset into memory. But you need to chunk your data yourself, which makes the code a bit awkward to use. Make a function that can take a stream of data samples and perform

PCA. Then, use the function to compute the PCA of the `iris` machine learning dataset, which is in `data/iris.csv`. (You can also access it from the `datasets` module of scikit-learn.)

Hint: The `IncrementalPCA` class is in `sklearn.decomposition`, and requires a *batch size* greater than 1 to train the model. Look at the `toolz.curried.partition` function for how to create a stream of batches from a stream of data points.

First, we write the function to train the model. The function should take in a stream of samples and output a PCA model, which can *transform* new samples by projecting them from the original n-dimensional space to the principal component space.

```
In [26]:
```

```
import toolz as tz
from toolz import curried as c
from sklearn import decomposition
from sklearn import datasets
import numpy as np

def streaming_pca(samples, n_components=2, batch_size=100):
    ipca = decomposition.IncrementalPCA(n_components=n_components,
                                        batch_size=batch_size)
    # we use `tz.last` to force evaluation of the full iterator
    _ = tz.last(tz.pipe(samples,  # iterator of 1D arrays
                        c.partition(batch_size),  # iterator of tuples
                        c.map(np.array),  # iterator of 2D arrays
                        c.map(ipca.partial_fit)))  # partial_fit on each
    return ipca
```

Now, we can use this function to *train* (or *fit*) a PCA model:

```
In [27]:
```

```
reshape = tz.curry(np.reshape)

def array_from_txt(line, sep=',', dtype=np.float):
    return np.array(line.rstrip().split(sep), dtype=dtype)

with open('data/iris.csv') as fin:
    pca_obj = tz.pipe(fin, c.map(array_from_txt), streaming_pca)
```

Finally, we can stream our original samples through the `transform` function of our model. We stack them together to obtain a `n_samples` by `n_components` matrix of data:

```
In [28]:
```

```
with open('data/iris.csv') as fin:
    components = tz.pipe(fin,
                         c.map(array_from_txt),
                         c.map(reshape(newshape=(1, -1))),
                         c.map(pca_obj.transform),
```

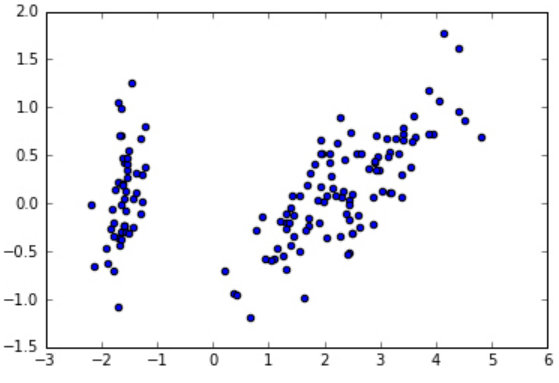
```
                          np.vstack)

print(components.shape)
```

```
(150, 2)
```

We can now plot the components:

```
In [29]:

from matplotlib import pyplot as plt
plt.scatter(*components.T)
```

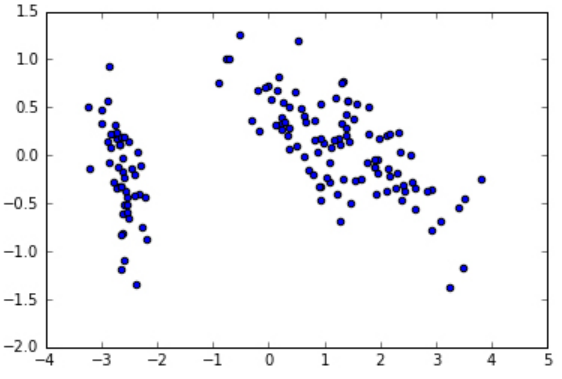
```
Out[29]:

<matplotlib.collections.PathCollection at 0x10f727978>
```



You can verify that this gives (approximately) the same result as a standard PCA:

```
In [30]:

iris = np.loadtxt('data/iris.csv', delimiter=',')
components2 = decomposition.PCA(n_components=2).fit_transform(iris)
plt.scatter(*components2.T)
```

```
Out[30]:

<matplotlib.collections.PathCollection at 0x10f7fa080>
```



The difference, of course, is that streaming PCA can scale to extremely large datasets!

Challenge: add a step to the start of the pipe to unzip the data so you don't have to keep a decompressed version on your hard drive. Yes, unzip can be streamed, too!

Hint: The `tarfile` package, part of Python's standard library, allows you to open tar files (even compressed ones) as if they were normal files.