O'REILLY®

2nd Edition

Data Visualization with Python and JavaScript

Scrape, Clean, Explore, and Transform Your Data

Early Release

RAW & UNEDITED

Kyran Dale

Data Visualization with Python and JavaScript

Scrape, Clean, Explore, and Transform Your Data

SECOND EDITION

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Kyran Dale

Data Visualization with Python and JavaScript

by Kyran Dale

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://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Michelle Smith

Development Editor: Shira Evans

Production Editor: Greg Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

July 2016: First Edition

December 2022: Second Edition

Revision History for the Early Release

- 2021-09-29: First Release
- 2021-12-02: Second Release
- 2022-01-26: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098111878 for release details.

978-1-098-11180-9

Chapter 1. A Language-Learning Bridge Between Python and JavaScript

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the second chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Probably the most ambitious aspect of this book is that it deals with two programming languages. Moreover, it only requires that you are competent in one of these languages. This is only possible because Python and JavaScript (JS) are fairly simple languages with much in common. The aim of this chapter is to draw out those commonalities and use them to make a learning bridge between the two languages such that core skills acquired in one can easily be applied to the other.

After showing the key similarities and differences between the two languages, I'll show how to set up a learning environment for Python and JS. The bulk of the chapter will then deal with core syntactical and conceptual differences, followed by a selection of patterns and idioms that I use often while doing data visualization work.

Similarities and Differences

Syntax differences aside, Python and JavaScript actually have a lot in common. After a short while, switching between them can be almost seamless.[1] Let's compare the two from a data visualizer's perspective:

These are the chief similarities:

- They both work without needing a compilation step (i.e., they are interpreted).
- You can use both with an interactive interpreter, which means you can type in lines of code and see the results right away.
- Both have garbage collection.
- Neither language has header files, package boilerplate, and so on.
- Both are primarily developed with a text editor—not an IDE.
- In both, functions are first-class citizens, which can be passed as arguments.

These are the key differences:

- Possibly the biggest difference is that JavaScript is single-threaded and non-blocking, using asynchronous I/O. This means that simple things like file access involve the use of a callback function.
- JS is used essentially in web development and until relatively recently was browser-bound,[2] but Python is used almost everywhere.
- JS is the only first-class language in web browsers, Python being excluded.

- Python has a comprehensive standard library, whereas JS has a limited set of utility objects (e.g., JSON, Math).
- Python has fairly classical object-oriented classes, whereas JS uses prototypes.
- JS lacks heavyweight general-purpose data-processing libs.[3]

The differences here emphasize the need for this book to be bi-lingual. JavaScript's monopoly of browser dataviz needs the complement of a conventional data-processing stack. And Python has the best there is.

Interacting with the Code

One of the great advantages of Python and JavaScript is that because they are interpreted on the fly, you can interact with them. Python's interpreter can be run from the command line, whereas JavaScript's is generally accessed from the web browser through a console, usually available from the built-in development tools. In this section, we'll see how to fire up a session with the interpreter and start trying out your code.

Python

By far, the best command-line Python interpreter is IPython, which comes in three shades: the basic terminal version, an enhanced graphical version, and a browser-based notebook. Since version 4.0 the latter two have been spun out into project Jupyter. The Jupyter notebook is a rather brilliant and fairly recent innovation, providing a browser-based interactive computational environment. There are pros and cons to the different versions. The command line is fastest to scratch a problematic itch but lacks some bells and whistles, particularly embedded plotting courtesy of *Matplotlib* and friends. The makes it suboptimal for *Pandas*-based data-processing and data visualization work. Of the other two, both are better for multiline coding (e.g., trying out functions) than the basic interpreter, but I find the graphical Qt console more intuitive, having a familiar command line rather than executable cells.[4] The great boon of the Notebook is session persistence and the possibility of web access.[5] The ease with which one can share programming sessions, complete with embedded data visualizations, makes the notebook a fantastic teaching tool as well as a great way to recover programming context. That's why the Python chapters of this book have accompanying Jupyter notebooks.

You can start an ipython session at the command-line like this:

```
$ ipython
```

To start a Jupyter notebook just run `jupyter` at the command-line.

```
$ jupyter notebook
[I 15:27:44.553 NotebookApp] Serving notebooks from local
directory:
...
[I 15:27:44.553 NotebookApp] http://localhost:8888/?token=5e09...
```

Then open a browser tab at the URL specified (+http://localhost:8888 in the case) and start reading or writing Python notebooks.

JavaScript

There are lots of options for trying out JavaScript code without starting a server, though the latter isn't that difficult. Because the JavaScript interpreter comes embedded in all modern web browsers, there are a number of sites that let you try out bits of JavaScript along with HTML and CSS and see the results. *Codepen* is a good option. These sites are great for sharing code and trying out snippets, and usually allow you to add libraries such as *D3.js* with a few mouse-clicks.

If you want to try out code one-liners or quiz the state of live code, browser-based consoles are your best bet. With Chrome, you can access the console with the key combo Ctrl-Shift-J. As well as trying little JS snippets, the console allows you to drill down into any objects in scope, revealing their methods and properties. This is a great way to quiz the state of a live object and search for bugs.

One disadvantage of using online JavaScript editors is losing the power of your favorite editing environment, with linting, familiar keyboard shortcuts, and the like (see [Link to Come]). Online editors tend to be rudimentary, to say the least. If you anticipate an extensive JavaScript session and want to use your favorite editor, the best bet is to run a local server.

First, create a project directory—called *sandpit*, for example—and add a minimal HTML file that includes a JS script:

```
sandpit
├── index.html
└── script.js
```

The *index.html* file need only be a few lines long, with an optional `div` placeholder on which to start building your visualization or just trying out a little DOM manipulation.

```
<!-- index.html -->
<!DOCTYPE html>
<meta charset="utf-8">

<div id='viz'></div>

<script type="text/javascript" src="script.js" async></script>
```

You can then add a little JavaScript to your *script.js* file:

```
// script.js
var data = [3, 7, 2, 9, 1, 11];
var sum = 0;
data.forEach(function(d){
    sum += d;
});

console.log('Sum = ' + sum);
// outputs 'Sum = 33'
```

Start your development server in the project directory using Python's `http` module:

```
$ python -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 ...
```

Then open your browser at *http://localhost:8000*, press Ctrl-Shift-J (Cmd-Opt-J on OS X) to access the console and you should see Figure 1-1, showing the logged output of the script (see [Link to Come] for further details).

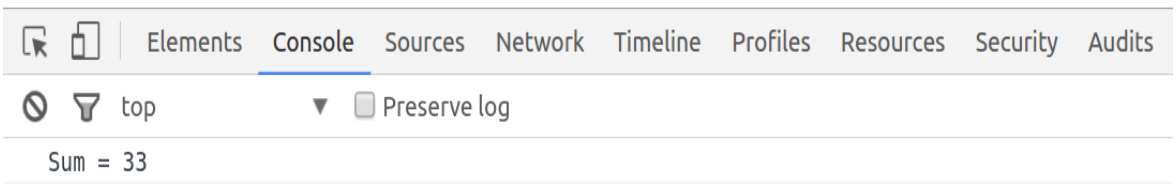


Figure 1-1. Outputting to the Chrome console

Now that we've established how to run the demo code, let's start building a bridge between Python and JavaScript. First, we'll cover the basic differences in syntax. As you'll see, they're fairly minor and easily

absorbed.

Basic Bridge Work

In this section, I'll contrast the basic nuts and bolts of programming in the two languages.

Style Guidelines, PEP 8, and use strict

Where JavaScript style guidelines are a bit of a free-for-all (with people often defaulting to those used by a big library like React), Python has a Python Enhancement Proposal (PEP) dedicated to it. I'd recommend getting acquainted with PEP-8 but not submitting totally to its leadership. It's right about most things, but there's room for some personal choice here. There's a handy online checker here, which will pick up any infractions of PEP-8.

In Python, you should use four spaces to indent a code block. JavaScript is less strict, but two spaces is the most common indent.

One recent addition to JavaScript (Ecmascript 5) is the `'use strict'` directive, which imposes strict mode. This mode enforces some good JavaScript practice, which includes catching accidental global declarations, and I thoroughly recommend its use. To use it, just place the string at the top of your function or module:

```
(function(foo){
  'use strict';
  // ...
}(window.foo = window.foo || {});
```

CamelCase Versus Underscore

JS conventionally uses CamelCase (e.g., `processStudentData`) for its variables, whereas Python, in accordance with PEP-8, uses underscores (e.g., `process_student_data`) in its variable names (see Section B in Examples 1-3 and 1-4). By convention (and convention is more important in the Python ecosystem than JS), Python uses capitalized CamelCase for class declarations (see the following example), uppercase for constants, and underscores for everything else:

```
FOO_CONST = 10
class FooBar(object): # ...
def foo_bar():
    baz_bar = 'some string'
```

Importing Modules, Including Scripts

Using other libraries in your code, either your own or third-party, is fundamental to modern programming, which makes it all the more surprising that until relatively recently JavaScript didn't have a dedicated way of doing it.[6] Python has a simple import system that, on the whole, works pretty well.

The good news on the JavaScript front is that since Ecmascript 6, JavaScript has addressed this issue, with the addition of `import` and `export` statements for encapsulated modules. Ecmascript 6 will be getting browser support soon but as of late 2021, for cross-browser support you need a converter to Ecmascript 5, such as Babel.js. Meanwhile, although there have been many attempts to create a reasonable client-side modular system, none have really achieved critical mass and all are a little awkward to use. For now, I recommend using the well-established HTML `<script>` tag to include scripts. So to include the D3 visualization library, you would add this tag to your main HTML file, conventionally *index.html*:

```
<!DOCTYPE html>
<meta charset="utf-8">
```

```
...
  <script src="http://d3js.org/d3.v7.min.js"></script>
```

You can include the script anywhere in your HTML file, but it's best practice to add scripts after the body (div tags, etc.) section.[7] Note that the order of the `<script>` tags is important. If a script is dependent on a module (e.g., it uses the D3 library), its `<script>` tag must be placed after that of the module. In other words, big library scripts, such as jQuery and D3, will be included first.

Python comes with "batteries included," a comprehensive set of libraries covering everything from extended data containers (`collections`) to working with the family of CSV files (`csv`). If you want to use one of these, just import it using the `import` keyword:

```
In [1]: import sys

In [2]: sys.platform
Out[2]: 'linux'
```

If you don't want to import the whole library or want to use an alias, you can use the `as` and `from` keywords instead:

```
import pandas as pd
from csv import DictWriter, DictReader
from numpy import * ❶

df = pd.read_json('data.json')
reader = DictReader('data.csv')
md = median([12, 56, 44, 33])
```

❶ This imports all the variables from the module into the current namespace and is almost always a bad idea. One of the variables could mask an existing one, and it goes against the Python best practice of explicit being better than implicit. One exception to this rule is if you are using the Python interpreter interactively. In this limited context, it may make sense to import all functions from a library to cut down on key presses; for example, importing all the math functions (`from math import *`) if doing some Python math hacking.

If you import a nonstandard library, Python uses `sys.path` to try to find it. `sys.path` consists of:

- The directory containing the importing module (current directory)
- The `PYTHOPATH` variable, containing a list of directories
- The installation-dependent default, where libraries installed using `pip` or `easy_install` will usually be placed

Big libraries are often packaged, divided into submodules. These submodules are accessed by dot notation:

```
import matplotlib.pyplot as plt
```

Packages are constructed from the filesystem via *__init__.py* files, usually empty, as shown in Example 1-1. The presence of an init file makes the directory visible to Python's import system.

Example 1-1. Building a Python package

```
mypackage
├── __init__.py
...
├── core
│   └── __init__.py
.
```

```
|   ...
...
└── io
    ├── __init__.py
    └── api.py
    ...
    └── tests
        └── __init__.py
        └── test_data.py
        └── test_excel.py ❶
        ...
...
```

❶ You would import this module using `from mypackage.io.tests import test_excel`.

You can access packages on `sys.path` from the root directory (that's `mypackage` in Example 1-1) using dot notation. A special case of `import` is intrapackage references. The *test_excel.py* submodule in Example 1-1 can import submodules from the `mypackage` package both absolutely and relatively:

```
from mypackage.io.tests import test_data ❶
from . import test_data ❷
import test_data ❷
from ..io import api ❸
```

❶ Imports the *test_data.py* module absolutely, from the package's head directory.

❷ An explicit (`. import`) and implicit relative import.

❸ A relative import from a sibling package of *tests*.

Keeping Your Namespaces Clean

The variables defined in Python modules are encapsulated, which means that unless you import them explicitly (e.g., `from foo import baa`), you will be accessing them from the imported module's namespace using dot notation (e.g., `foo.baa`). This modularization of the global namespace is quite rightly seen as a very good thing and plays to one of Python's key tenets: the importance of explicit statements over implicit. When analyzing someone's Python code you should be able to see exactly where a class, function, or variable has come from. Just as importantly, preserving the namespace limits the chance of conflicting or masking variables—a big potential problem as code bases get larger.

One of the main criticisms of JavaScript, and a fair one, is that it plays fast and loose with namespace conventions. The most egregious example of this is that variables declared outside of functions or missing the `var` keyword[8] are global rather than confined to the script in which they are declared. If you can deal with the overhead of setting up a modern, modular JavaScript you get Python like encapsulation with imported and exported variables. While keeping things simple and using JavaScript scripts, there are various ways to rectify this situation, but the one I use and recommend is to make each of your scripts a self-calling function. This makes all variables declared via `var` local to the script/function, preventing them from polluting the global namespace. A new JavaScript `let` keyword, which is block-scoped, is pretty much always preferable to `var`. Any objects, functions, and variables you want to make available to other scripts can be attached to an object that is part of the global namespace.

Example 1-2 demonstrates a module pattern. The boilerplate head and tail (labeled ❶ and ❸) effectively create an encapsulated module. This pattern is far from a perfect solution to modular JavaScript but is the best compromise I know until Ecmascript 6 and a dedicated import system for the browser becomes standard. One obvious disadvantage is that the module is part of the global namespace, which means, unlike in Python, there is no need to explicitly import it.

Example 1-2. A module pattern for JavaScript

```
(function(nbviz) { ❶
    'use strict';
    // ...
    nbviz.updateTimeChart = function(data) { ❷
    // ...
}(window.nbviz = window.nbviz || {})); ❸
```

❶ Receives the global `nbviz` object.

❷ Attaches the `updateTimeChart` method to the global `nbviz` object, effectively *exporting* it.

❸ If an `nbviz` object exists in the global (window) namespace, pass it into the module function; otherwise, add it to the global namespace.

Outputting "Hello World!"

By far the most popular initial demonstration of any programming language is getting it to print or communicate "Hello World!" in some form, so let's start with getting output from Python and JavaScript.

Python's output couldn't be much simpler, but version 3 sees a change to the `print` statement, making it a proper function:[9]

```
# In Python 2*
print 'Hello World!'

# In Python 3+
print('Hello World!')
```

You can use Python 3's print function in Python 2 by importing it from the `__future__` module:

```
from __future__ import print_function
```

If you're not using Python 3, then this is a sensible approach but I would strongly recommend moving to the newer Python.

JavaScript has no print function, but you can log output to the browser console:

```
console.log('Hello World!');
```

Simple Data Processing

A good way to get an overview of the language differences is to see the same function written in both. Examples 1-3 and 1-4 show a small, contrived example of data munging in Python and JavaScript, respectively. We'll use these to compare Python and JS syntax.

Example 1-3. Simple data munging with Python

```
# A
student_data = [
    {'name': 'Bob', 'id':0, 'scores':[68, 75, 56, 81]},
    {'name': 'Alice', 'id':1,  'scores':[75, 90, 64, 88]},
    {'name': 'Carol', 'id':2, 'scores':[59, 74, 71, 68]},
    {'name': 'Dan', 'id':3, 'scores':[64, 58, 53, 62]},
]

# B
def process_student_data(data, pass_threshold=60,
                         merit_threshold=75):
    """ Perform some basic stats on some student data. """
```

```
    # C
    for sdata in data:
        av = sum(sdata['scores'])/float(len(sdata['scores']))
        sdata['average'] = av

        if av > merit_threshold:
            sdata['assessment'] = 'passed with merit'
        elif av > pass_threshold:
            sdata['assessment'] = 'passed'
        else:
            sdata['assessment'] = 'failed'
        # D
        print("%s's (id: %d) final assessment is: %s"%(
          sdata['name'], sdata['id'], sdata['assessment'].upper()))

# E
if __name__ == '__main__':
    process_student_data(student_data)
```

Example 1-4. Simple data munging with JavaScript

```
// A (note deliberate and valid inconsistency in keys: some quoted
// and some unquoted)
var studentData = [
    {name: 'Bob', id:0, 'scores':[68, 75, 76, 81]},
    {name: 'Alice', id:1, 'scores':[75, 90, 64, 88]},
    {'name': 'Carol', id:2, 'scores':[59, 74, 71, 68]},
    {'name': 'Dan', id:3, 'scores':[64, 58, 53, 62]},
];

// B
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'?\
    passThreshold: 60;
    meritThreshold = typeof meritThreshold !== 'undefined'?\
    meritThreshold: 75;

    // C
    data.forEach(function(sdata){
        var av = sdata.scores.reduce(function(prev, current){
            return prev+current;
        },0) / sdata.scores.length;
        sdata.average = av;

        if(av > meritThreshold){
            sdata.assessment = 'passed with merit';
        }
        else if(av > passThreshold){
            sdata.assessment = 'passed';
        }
        else{
            sdata.assessment = 'failed';
        }
        // D
        console.log(sdata.name + "'s (id: " + sdata.id +
          ") final assessment is: " +
            sdata.assessment.toUpperCase());
    });

}

// E
processStudentData(studentData);
```

String Construction

Section D in Examples 1-3 and 1-4 show the standard way to print output to the console or terminal. JavaScript has no `print` statement but will log to the browser's console through the `console` object.

```
console.log(sdata.name + "'s (id: " + sdata.id +
  ") final assessment is: " + sdata.assessment.toUpperCase());
```

Note that the integer variable `id` is coerced to a string, allowing concatenation. Python doesn't perform this implicit coercion, so attempting to add a string to an integer in this way gives an error. Instead, explicit conversion to string form is achieved through one of the `str` or `repr` functions.

In section A of Example 1-3, the output string is constructed with C type formatting. String (`%s`) and integer (`%d`) placeholders are provided by a final tuple (`% (...)`):

```
print("%s's (id: %d) final assessment is: %s"
  %(sdata['name'], sdata['id'], sdata['assessment'].upper()))
```

These days, I rarely use Python's `print` statement, opting for the much more powerful and flexible `logging` module, which is demonstrated in the following code block. It takes a little more effort to use, but it is worth it. Logging gives you the flexibility to direct output to a file and/or the screen, adjusting the logging level to prioritize certain information, and a whole load of other useful things. Check out the details here.

```
import logging
logger = logging.getLogger(__name__) ❶
//...
logger.debug('Some useful debugging output')
logger.info('Some general information')

// IN INITIAL MODULE
logging.basicConfig(level=logging.DEBUG) ❷
```

❶ Creates a logger with the name of this module.

❷ You can set the logging level, an output file as opposed to the default to screen.

Significant Whitespace Versus Curly Brackets

The syntactic feature most associated with Python is significant whitespace. Whereas languages like C and JavaScript use whitespace for readability and could easily be condensed into one line,[10] in Python leading spaces are used to indicate code blocks and removing them changes the meaning of the code. The extra effort required to maintain correct code alignment is more than compensated for by increased readability—you spend far longer reading than writing code and the easy reading of Python is probably the main reason why the Python library ecosystem is so healthy. Four spaces is pretty much mandatory (see PEP 8) and my personal preference is for what is known as *soft tabs*, where your editor inserts (and deletes) multiple spaces instead of a tab character.[11]

In the following code, the indentation of the `return` statement must be four spaces by convention:[12]

```
def doubler(x):
    return x * 2
#  |<-this spacing is important
```

JavaScript doesn't care about the number of spaces between statements and variables, using curly brackets to demark code blocks, the two doubler functions in this code being equivalent:

```
let doubler = function(x){
  return x * 2;
}

let doubler=function(x){return x*2;}
```

Much is made of Python's whitespace, but most good coders I know set up their editors to enforce indented code blocks and a consistent look and feel. Python merely enforces this good practice. And, to reiterate, I believe the extreme readability of Python code contributes as much to Python's supremely healthy ecosystem as its simple syntax.

Comments and doc-strings

To add comments to code, Python uses hashes, #:

```
# ex.py, a single informative comment

data = {} # Our main data-ball
```

By contrast, JavaScript uses the C language convention of double backslashes (//) or /* ... */ for multiline comments:

```
// script.js, a single informative comment
/* A multiline comment block for
function descriptions, library script
headers, and the like */
let data = {}; // Our main data-ball
```

In addition to comments, and in keeping with its philosophy of readability and transparency, Python has documentation strings (docstrings) by convention. The `process_student_data` function in Example 1-3 has a triple-quoted line of text at its top that will automatically be assigned to the function's `__doc__` attribute. You can also use multiline doc-strings.

```
def doubler(x):
    """This function returns double its input."""
    return 2 * x

def sanitize_string(s):
    """This function replaces any string spaces
    with '-' after stripping any whitespace
    """
    return s.strip().replace(' ', '-')
```

Doc-strings are a great habit to get into, particularly if you are working collaboratively. They are understood by most decent Python editing toolsets and are also used by such automated documentation libraries as *Sphinx*. The string-literal doc-string is accessible as the `doc` property of a function or class.

Declaring Variables, let, var

JavaScript uses `let` or `var` to declare variables. Generally speaking, `let` is almost always the right choice.

Strictly speaking, JS statements should be terminated with a semicolon as opposed to Python's newline. You will see examples where the semicolon is dispensed with, and modern browsers will usually do the right thing here. There are few edge-cases that could necessitate the use of a semi-colon (e.g., it can trip up code minifiers and compressors that remove whitespace) but generally I find that the loss of clutter and improvement in readablity are a worthwhile compromise to coding without the semi-colons.

> **TIP**
>
> JavaScript has *variable hoisting*, which means variables declared with `var` are processed before any other code. This means declaring them anywhere in the function is equivalent to declaring them at the top. This can result in weird errors and confusion. Explicitly placing `var`s at the top avoids this but it's better to use the modern `let` and have scoped declarations.

Strings and Numbers

The *name* strings used in the student data (see Section A of Examples 1-3 and 1-4) will be interpreted as UCS-2 (the parent of unicode UTF-16) in JavaScript,[13] and Unicode (UTF-8 by default) in Python 3.[14]

Both languages allow single and double quotes for strings. If you want to include a single or double quote in the string, then enclose with the alternative, like so:

```
pub_name = "The Brewer's Tap"
```

The `scores` in Section A of Example 1-4 are stored as JavaScript's one numeric type, double-precision 64-bit (IEEE 754) floating-point numbers. Although JavaScript has a `parseInt` conversion function, when used with floats,[15] it is really just a rounding operator, similar to `floor`. The type of the parsed `number` is still `number`:

```
var x = parseInt(3.45); // 'cast' x to 3
typeof(x); // "number"
```

Python has three numeric types: the 32-bit `int`,to which the student scores are cast, a `float` equivalent (IEE 754) to JS's `number`, and a `long` for arbitrary precision integer arithmetic. This means that Python can represent any integer, whereas JavaScript is more limited.[16] Python's casting changes type:

```
foo = 3.4 # type(foo) -> float
bar = int(3.4) # type(bar) -> int
```

The nice thing about Python and JavaScript numbers is that they are easy to work with and usually do what you want. If you need something more efficient, Python has the *NumPy* library, which allows fine-grained control of your numeric types (you'll learn more about NumPy in [Link to Come]). In JavaScript, aside from some cutting-edge projects, you're pretty much stuck with 64-bit floats.

Booleans

Python differs from the JavaScript and the C class languages in using named boolean operators. Other than that, they work pretty much as expected. This table gives a comparison:

Python	`bool`	`True`	`False`	`not`	`and`	`or`
JavaScript	`boolean`	`true`	`false`	`!`	`&&`	+`

Python's capitalized `True` and `False` is an obvious trip-up for any JavaScripter and vice versa, but any decent syntax highlighting should catch that, as should your code linter.

Rather than always returning boolean true or false, both Python and JavaScript `and`/`or` expressions return the result of one of the arguments, which may of course be a boolean value. Table 1-1 shows how this works, using Python to demonstrate.

Table 1-1. Python's boolean operators

Operation	Result
x or y	if x is false, then y, else x
x and y	if x is false, then x, else y
not x	if x is false, then `True`, else `False`

This fact allows for some occasionally useful variable assignments:

```
rocket_launch = True
(rocket_launch == True and 'All OK') or 'We have a problem!'
Out:
```

```
'All OK'

rocket_launch = False
(rocket_launch == True and 'All OK') or 'We have a problem!'
Out:
'We have a problem!'
```

Data Containers: Dicts, Objects, Lists, Arrays

Roughly speaking, JavaScript `objects` can be used like Python `dicts`, and Python `lists` like JavaScript arrays. Python also has a tuple container, which functions like an immutable list. Here are some examples:

```
# Python
d = {'name': 'Groucho', 'occupation': 'Ruler of Freedonia'}
l = ['Harpo', 'Groucho', 99]
t = ('an', 'immutable', 'container')

// JavaScript
d = {'name': 'Groucho', 'occupation': 'Ruler of Freedonia'}
l = ['Harpo', 'Groucho', 99]
```

As shown in Section A of Examples 1-3 and 1-4, while Python's `dict` keys must be quote-marked strings (or hashable types), JavaScript allows you to omit the quotes if the property is a valid identifier (i.e., not containing special characters such as spaces and dashes). So in our `studentData` objects, JS implicitly converts the property `'name'` to string form.

The student data declarations look pretty much the same and, in practice, are used pretty much the same, too. The key difference to note is that while the curly-bracketed containers in the JS `studentData` look like Python `dicts`, they are actually a shorthand declaration of JS objects, a somewhat different data container.

In JS data visualization, we tend to use arrays of objects as the chief data container and here, JS objects function much as a Pythonista would expect. In fact, as demonstrated in the following code, we get the advantage of both dot notation and key-string access, the former being preferred where applicable (keys with spaces or dashes needing quoted strings):

```
var foo = {bar:3, baz:5};
foo.bar; // 3
foo['baz']; // 5, same as Python
```

It's good to be aware that although they can be used like Python dictionaries, JavaScript objects are much more than just containers (aside from primitives like strings and numbers, pretty much everything in JavaScript is an object).[17] But in most dataviz examples you see, they are used very much like Python `dicts`.

Table 1-2 converts basic list operations.

Table 1-2. Lists and arrays

JavaScript array (a)	Python list (l)
`a.length`	`len(l)`
`a.push(item)`	`l.append(item)`
`a.pop()`	`l.pop()`
`a.shift()`	`l.pop(0)`
`a.unshift(item)`	`l.insert(0, item)`
`a.slice(start, end)`	`l[start:end]`
`a.splice(start, howMany, i1, …)`	`l[start:end] = [i1, …]`

Functions

Section B of Examples 1-3 and 1-4 shows a function declaration. Python uses `def` to indicate a function:

```
def process_student_data(data, pass_threshold=60,
                         merit_threshold=75):
    """ Perform some basic stats on some student data. """
    ...
```

whereas JavaScript uses `function`:

```
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'?
      passThreshold: 60;
    meritThreshold = typeof meritThreshold !== 'undefined'?
      meritThreshold: 75;
    ...
}
```

Both have a list of parameters. With JS, the function code block is indicated by the curly brackets *{ ... }*; with Python, the code block is defined by a colon and indentation.

JS has an alternative way of defining a function called the *function expression*, which you may see in examples:

```
let processStudentData = function( ...){
```

There is now a shortened form , which is becoming more popular:

```
let processStudentData = ( ...) => {
```

The differences are subtle enough not to worry about for now.[18]

Function parameters is an area where Python's handling is a great deal more sophisticated than JavaScript's. As you can see in `process_student_data` (Section B in Example 1-3), Python allows default arguments for the parameters. In JavaScript, all parameters not used in the function call are declared as *undefined*.

Until recently, in order to set a default value for these, we had to perform a distinctly hacky conditional (ternary) expression. You may see this in older libraries or code-examples:

```
function processStudentData(data, passThreshold, meritThreshold){
    passThreshold = typeof passThreshold !== 'undefined'?
      passThreshold: 60;
    ...
```

The good news for JavaScripters is that recent versions of JavaScript, based on Ecmascript 6, allows Python-like default parameters:

```
function processStudentData(data, passThreshold=60,
  meritThreshold=75){
...
```

Iterating: for Loops and Functional Alternatives

Section C in Examples 1-3 and 1-4 shows our first major departure, demonstrating JavaScript's functional chops.

Python's `for` loops are simple, intuitive, and effective on any iterator, such as arrays and `dicts`. One gotcha with `dicts` is that standard iteration is by key, not items. For example:

```
foo = {'a':3, 'b':2}
for x in foo:
    print(x)
# outputs 'a' 'b'
```

To iterate over the key-value pairs, use the `dict`'s `items` method like so:

```
for x in foo.items():
    print(x)
# outputs key-value tuples ('a', 3) ('b' 2)
```

You can assign the key/values in the `for` statement for convenience. For example:

```
for key, value in foo.items():
```

Because Python's `for` loop works on anything with the correct iterator plumbing, you can do cool things like loop over file lines:

```
for line in open('data.txt'):
    print(line)
```

Coming from Python, JS's for loop is a pretty horrible, unintuitive thing. Here's an example:

```
for(var i in ['a', 'b', 'c']){
  console.log(i)
}
# outputs 1, 2, 3
```

JS's `for .. in` returns the index of the array's items, not the items themselves. To compound matters, for the Pythonista, the order of iteration is not guaranteed, so the indices could be returned in non-consecutive order.

Shifting between Python and JS `for` loops is hardly seamless, demanding you keep on the ball. The good news is that you hardly need to use JS `for` loops these days. In fact, I almost never find the need. That's because JS has recently acquired some very powerful first-class functional abilities, which have more expressive power and less scope for confusion with Python and, once you get used to them, quickly become indispensable.[19]

Section C in Example 1-4 demonstrates `forEach()`, one of the *functional* methods available to modern JavaScript arrays.[20] `forEach()` iterates over the array's items, sending them in turn to an anonymous callback function defined in the first argument, where they can be processed. The true expressive power of these functional methods comes from chaining them (maps, filters, etc.), but already we have a cleaner, more elegant iteration with none of the awkward bookkeeping of old.

The callback function receives index and the original array as an optional second argument.

```
data.forEach(function(currentValue, index){//---
```

Until recently even iterating over an object's key-value pairs was fairly tricky. Unlike Python's `dicts`, objects could have inherited properties from the prototyping chain, so you had to use a `hasOwnProperty` guard to filter these out. You may well come across code like this:

```
var obj = {a:3, b:2, c:4};
for (var prop in obj) {
  if( obj.hasOwnProperty( prop ) ) {
    console.log("o." + prop + " = " + obj[prop]);
  }
}
// out: o.a = 3, o.b = 2, o.c = 4
```

Whereas JS arrays have a set of native functional iterator methods (`map`, `reduce`, `filter`, `every`, `sum`, `reduceRight`), `Objects`—in their guise as pseudo-dictionaries—don't. The good news is that the `Object` class has recently acquired some useful additional methods which fill this gap. So you can iterate through the key-value pairs using the `entries` method:

```
var obj = {a:3, b:2, c:4};
for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`); ❶
}
```

```
// out: a: 3
        b: 2 ...
```

❶ note the string template form `${foo}` for printing variables.

Conditionals: if, else, elif, switch

Section C in Examples 1-3 and 1-4 shows Python and JavaScript conditionals in action. Aside from JavaScript's bracket fetish, the statements are very similar; the only real difference being Python's extra `elif` keyword, a convenient conjunction of `else if`.

Though much requested, Python does not have the `switch` statement found in most high-level languages. JS does, allowing you to do this:

```
switch(expression){
  case value1:
    // execute if expression === value1
    break; // optional end expression
  case value2:
    //...
  default:
    // if other matches fail
```

File Input and Output

Browser-based JavaScript has no real equivalent of file input and output (I/O), but Python's is as simple as could be:

```
# READING A FILE
f = open("data.txt") # open file for reading

for line in f: # iterate over file lines
    print(line)

lines = f.readlines() # grab all lines in file into array
data = f.read() # read all of file as single string

# WRITING TO A FILE
f = open("data.txt", 'w')
# use 'w' to write, 'a' to append to file
f.write("this will be written as a line to the file")
f.close() # explicitly close the file
```

One much recommended best practice is to use Python's `with`, `as` context manager when opening files. This ensures they are closed automatically when leaving the block, essentially providing syntactic sugar for a `try`, `except`, `finally` block. Here's how to open a file using `with`, `as`:

```
with open("data.txt") as f:
    lines = f.readlines()
    ...
```

JavaScript does, however, have the roughly analagous `fetch` method for fetching a resource from the network, based on its URL. So to fetch a dataset from the web-site's server, in the *static/data* directory you do this:

```
fetch('/static/data/nobel_winners.json')
  .then(function(response) {
  console.log(response.json())
})
```

```
Out:
[{name: 'Albert Einstein', category: 'Physics'...
```

The Fetch API is thoroughly documented at Mozilla.

Classes and Prototypes

Possibly the cause of more confusion that any other topic is JavaScript's choice of prototypes rather than classical classes as its chief object-oriented programming (OOP) element. I have come to appreciate the concept of prototypes, if not its JS implementation, which could have been cleaner. Nevertheless, once you get the basic principle, you may find that it is actually a better mental model for much of what we do as programmers than classical OOP paradigms.

I remember, when I first started my forays into more advanced languages like C++, falling for the promise of OOP, particularly class-based inheritance. Polymorphism was all the rage and Shape classes were being subclassed to rectangles and ellipses, which were in turn subclassed to more specialized squares and circles.

It didn't take long to realize that the clean class divisions found in the textbooks were rarely found in real programming and that trying to balance generic and specific APIs quickly became fraught. In this sense, I find composition and mix-ins much more useful as a programming concept than attempts at extended subclassing and often avoid all these by using functional programming techniques, particularly in JavaScript. Nevertheless, the class/prototype distinction is an obvious difference between the two languages, and the more you understand its nuances, the better you'll code.[21]

Python's classes are fairly simple affairs and, as with most of the language, easy to use. I tend to think of them these days as a handy way to encapsulate data with a convenient API, and rarely extend sub-classing beyond one generation. Here's a simple example:

```
class Citizen(object):

    def __init__(self, name, country): ❶
        self.name = name
        self.country = country

     def __str__(self): ❷
        return 'Citizen %s from %s'%(self.name, self.country)

     def print_details(self):
          print('Citizen %s from %s'%(self.name, self.country))


groucho = Citizen('Groucho M.', 'Freedonia') ❸
print(groucho) # or groucho.print_details()
Out:
Citizen Groucho M. from Freedonia
```

❶ Python classes have a number of double-underscored special methods, `__init__` being the most common, called when the class instance is created. All instance methods have a first, explicit `self` argument (you could name it something else, but it's a very bad idea), which refers to the instance. In this case, we use it to set name and country properties.

❷ You can override the class's string method, which is used when the `print` function is called on an instance.

❸ Creates a new `Citizen` instance, initialized with name and country.

Python follows a fairly classical pattern of class inheritance. It's easy to do, which is probably why Pythonistas make a lot of use of it. Let's customize the `Citizen` class to create a (Nobel Prize) `Winner` class with a couple of extra properties:

```
class Winner(Citizen):

    def __init__(self, name, country, category, year):
        super(Winner, self).__init__(name, country) ❶
        self.category = category
        self.year = year

    def __str__(self):
        return 'Nobel winner %s from %s, category %s, year %s'\
        %(self.name, self.country, self.category,\
        str(self.year))

w = Winner('Albert E.', 'Switzerland', 'Physics', 1921)
w.print_details()
Out:
Nobel prize-winner Albert E. from Switzerland, category Physics,
year 1921
```

❶ We want to reuse the superclass `Citizen's __init__` method, using this `Winner` instance as `self`. The `super` method scales the inheritance tree one branch from its first argument, supplying the second as instance to the class-instance method.

I think the best article I have read on the key difference between JavaScript's prototypes and classical classes is Reginald Braithwaite's "OOP, JavaScript, and so-called Classes". This quote sums up the difference between classes and prototypes as nicely as any I've found:

> *The difference between a prototype and a class is similar to the difference between a model home and a blueprint for a home.*

When you instantiate a C++ or Python class, a blueprint is followed, creating an object and calling its various constructors in the inheritance tree. In other words, you start from scratch and build a nice, pristine new class instance.

With JavaScript prototypes, you start with a model home (object) that has rooms (methods). If you want a new living room, you can just replace the old one with something in better colors. If you want a new conservatory, then just make an extension. But rather than building from scratch with a blueprint, you're adapting and extending an existing object.

With that necessary theory out of the way and the reminder that object inheritance is useful to know but hardly ubiquitous in dataviz, let's see a simple JavaScript prototype object in Example 1-5.

Example 1-5. A simple JavaScript object

```
var Citizen = function(name, country){ ❶
  this.name = name; ❷
  this.country = country;
};

Citizen.prototype = { ❸
  logDetails: function(){
    console.log(`Citizen ${this.name} from ${this.country}`);
  }
};

var c = new Citizen('Groucho M.', 'Freedonia'); ❹

c.logDetails();
Out:
```

```
Citizen Groucho M. from Freedonia

typeof(c) # object
```

❶ This function is essentially an initialiser, invoked by the `new` operator.

❷ `this` is an implicit reference to the *calling context* of the function. For now, it behaves as you would expect and even though it looks a little like Python's `self`, the two are quite different, as we'll see.

❸ The methods specified here will both override any prototypical methods up the inheritance chain and be inherited by any objects derived from `Citizen`.

❹ `new` is used to create a new object, set its prototype to that of the `Citizen` constructor function, and then call the `Citizen` constructor function on the new object.

JavaScript has recently acquired some syntactic sugar allowing classes to be declared. This essentially wraps the Object based form (see Example 1-5) in something more familiar to programmers coming from `class` based languages like `Java` and `C#`. I think it's fair to say that classes haven't really taken off in frontend, browser-based JavaScript, having been usurped somewhat by new frameworks with an emphasis on reusable components (e.g. React, Vue, Svelte). Here's how we would implement the *Citizen* `Object` shown in Example 1-5:

```
class Citizen {
  constructor(name, country) {
    this.name = name
    this.country = country
  }

  logDetails() {
    console.log(`Citizen ${this.name} from ${this.country}`)
  }
}

const c = new Citizen('Groucho M.', 'Freedonia')
```

SELF VERSUS THIS

At first glance, it would be easy enough to assume that Python's `self` and JavaScript's `this` are essentially the same, the latter being an implicit version of the former, which is supplied to all class instance methods. Actually, `this` and `self` are significantly different. Let's use our bilingual `Citizen` class to demonstrate.

Python's `self` is a variable supplied to each class method (you can call it anything you like, but it's not advisable), representing the class instance. But `this` is a keyword that refers to the object calling the method. This calling object can be different from the method's object instance, and JavaScript provides the `call`, `bind`, and `apply` function methods to allow you to exploit this fact.

Let's use the `call` method to change the calling object of a `print_details` method and therefore the reference for `this`, used in the method to get the citizen's name:

```
let groucho = new Citizen('Groucho M.', 'Freedonia');
let harpo = new Citizen('Harpo M.', 'Freedonia');

groucho.logDetails.call(harpo);
Out:
"Citizen Harpo M. from Freedonia"
```

So JavaScript's `this` is a much more malleable proxy than Python's `self`, offering more freedom but also the responsibility of tracking calling context and, should you use it, making sure `new` is always used in creating objects.[22]

I included Example 1-5, which shows `new` in JavaScript object instantiation, because you will run into its use a fair deal. But the syntax is already a little awkward and gets quite a bit worse when you try to do inheritance. Ecmascript 5 introduced the `Object.create` method, a better way to create objects and to implement inheritance. I'd recommend using it in your own code, but `new` will probably crop up in some third-party libraries.

Let's use `Object.create` to create a `Citizen` and its `Winner` inheritor. To emphasize, JavaScript has many ways to do this, but Example 1-6 shows the cleanest I have found and my personal pattern.

Example 1-6. Prototypical inheritance with Object.create

```
var Citizen = { ❶
    setCitizen: function(name, country){
        this.name = name;
        this.country = country;
        return this;
    },
    printDetails: function(){
        console.log('Citizen ' + this.name + ' from ',\
        + this.country);
    }
};

let Winner = Object.create(Citizen);

Winner.setWinner = function(name, country, category, year){
    this.setCitizen(name, country);
    this.category = category;
    this.year = year;
    return this;
};

Winner.logDetails = function(){
    console.log('Nobel winner ' + this.name + ' from ' +
    this.country + ', category ' + this.category + ', year ' +
```

```
    this.year);
};

let albert = Object.create(Winner)
    .setWinner('Albert Einstein', 'Switzerland', 'Physics', 1921);

albert.logDetails();
Out:
Nobel winner Albert Einstein from Switzerland, category
Physics, year 1921
```

❶ `Citizen` is now an object rather than a constructor function. Think of this as the base house for any new buildings such as `Winner`.

To reiterate, prototypical inheritance is not seen that often in JavaScript dataviz, particularly the 800-pound gorilla D3 with its emphasis on declarative and functional patterns, with *raw* unencapsulated data being used to stamp its impression on the web page.

The tricky class/prototype comparison concludes this section on basic syntactic differences. Now let's look at some common patterns seen in dataviz work with Python and JS.

Differences in Practice

The syntactic differences between JS and Python are important to know and thankfully outweighed by their syntactic similarities. The meat and potatoes of imperative programming, loops, conditionals, data declaration, and manipulation is much the same. This is all the more so in the specialized domain of data processing and data visualization where the languages' first-class functions allow common idioms.

What follows is a less-than-comprehensive list of some important patterns and idioms seen in Python and JavaScript, from the perspective of a data visualizer. Where possible, a translation between the two languages is given.

Method Chaining

A common JavaScript idiom is *method chaining*, popularized by its most popular library, jQuery, and much used in D3. Method chaining involves returning an object from its own method in order to call another method on the result, using dot notation:

```
let sel = d3.select('#viz')
    .attr('width', '600px') ❶
    .attr('height', '400px')
    .style('background', 'lightgray');
```

❶ The `attr` method returns the D3 selection that called it, which is then used to call another `attr` method.

Method chaining is not much seen in Python, which generally advocates one statement per line in keeping with simplicity and readability.

Enumerating a List

Often it's useful to iterate through a list while keeping track of the item's index. Python has the very handy built-in `enumerate` function for just this reason:

```
names = ['Alice', 'Bob', 'Carol']

for i, n in enumerate(names):
```

```
    print('%d: %s'%(i, n))

Out:
0: Alice
1: Bob
2: Carol
```

JavaScript's list methods, such as `forEach` and the functional `map`, `reduce`, and `filter`, supply the iterated item and its index to the callback function:

```
let names = ['Alice', 'Bob', 'Carol'];

names.forEach(function(n, i){
    console.log(i + ': ' + n);
});

Out:
0: Alice
1: Bob
2: Carol
```

Tuple Unpacking

One of the first cool tricks Python initiates come across uses tuple unpacking to switch variables:

```
(a, b) = (b, a)
```

Note that the brackets are optional. This can be put to more practical purpose as a way of reducing the temporary variables, such as in a Fibonacci function:

```
def fibonacci(n):
    x, y = 0, 1
    for i in range(n):
        print(x)
        x, y = y, x + y
# fibonacci(6) -> 0, 1, 1, 2, 3, 5
```

If you want to ignore one of the unpacked variables, use an underscore:

```
winner = 'Albert Einstein', 'Physics', 1921, 'Swiss'

name, _, _, nationality = winner
```

The JavaScript language is adapting rapidly and has recently acquired some very powerful destructuring abilities. With the addition of the *spread operator* (`...`) this enables some very succinct data manipulation:

```
let a, b, rem ❶

[a, b] = [1, 2]
# swap variables
[a, b] = [b, a]
# using the spread-operator
[a, b, ...rem] = [1, 2, 3, 4, 5, 6,] # rem = [3, 4, 5, 6]
```

❶ Unlike in Python you still need to declare any variables you are going to use.

Collections

One of the most useful Python "batteries" is the `collections` module. This provides some specialized container datatypes to augment Python's standard set. It has a `deque`, which provides a list-like container with fast appends and pops at either end; an `OrderedDict`, which remembers the order entries were added; a `defaultdict`, which provides a factory function to set the dictionary's default; and a `Counter` container for counting hashable objects, among others. I find myself using the last three a lot. Here are a few examples:

```
from collections import Counter, defaultdict, OrderedDict

items = ['F', 'C', 'C', 'A', 'B', 'A', 'C', 'E', 'F']

cntr = Counter(items)
print(cntr)
cntr['C'] -=1
print(cntr)
Out:
Counter({'C': 3, 'A': 2, 'F': 2, 'B': 1, 'E': 1})
Counter({'A': 2, 'C': 2, 'F': 2, 'B': 1, 'E': 1})


d = defaultdict(int) ❶

for item in items:
    d[item] += 1 ❷

d
Out:
defaultdict(<type 'int'>, {'A': 2, 'C': 3, 'B': 1, 'E': 1, 'F': 2})


OrderedDict(sorted(d.items(), key=lambda i: i[1])) ❸
Out:
OrderedDict([('B', 1), ('E', 1), ('A', 2), ('F', 2), ('C', 3)]) ❹
```

❶ Sets the dictionary default to an integer, with value 0 by default.

❷ If the item-key doesn't exist, its value is set to the default of 0 and 1 added to that.

❸ Gets the list of items in the dictionary `d` as key-value tuple pairs, sorts using the integer value, and then creates an `OrderedDict` with the sorted list.

❹ The `OrderedDict` remembers the (sorted) order of the items as they were added to it.

You can get more details on the `collections` module from here.

If you want to replicate some of Python's `collections` function using more conventional JavaScript libraries, underscore (or its functionally identical replacement lodash[23]) is a good place to start. These libraries offer some enhanced functional programming utilities. Let's take a quick look at this very handy tool.

Underscore

Underscore is probably the most popular JavaScript library after the ubiquitous jQuery and offers a bevy of functional programming utilities for the JavaScript dataviz programmer. The easiest way to use underscore is to use a content delivery network (CDN) to load it remotely (these loads will be cached by your browser, making things very efficient for common libraries), like so:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/
             underscore.js/1.13.1/underscore-min.js"></script>
```

Underscore has loads of useful functions. There is, for example, a `countBy` method, which serves the same purpose as the Python's `collections` counter just discussed:

```
var items = ['F', 'C', 'C', 'A', 'B', 'A', 'C', 'E', 'F'];

_.countBy(items) ❶
Out:
Object {F: 2, C: 3, A: 2, B: 1, E: 1}
```

❶ Now you see why the library is called underscore.

As we'll now see, the inclusion in modern JavaScript of native functional methods (`map`, `reduce`, `filter`) and a `forEach` iterator for arrays has made underscore slightly less indispensable, but it still has some great utilities to augment vanilla JS. With a little chaining, you can produce extremely terse but very powerful code. Underscore was my gateway drug to functional programming in JavaScript, and the idioms are just as addictive today. Check out underscore's repertoire of utilities here.

Let's have a look at underscore in action, tackling a more involved task:

```
journeys = [
  {period:'morning', times:[44, 34, 56, 31]},
  {period:'evening', times:[35, 33],},
  {period:'morning', times:[33, 29, 35, 41]},
  {period:'evening', times:[24, 45, 27]},
  {period:'morning', times:[18, 23, 28]}
];

var groups = _.groupBy(journeys, 'period');
var mTimes = _.pluck(groups['morning'], 'times');
mTimes = _.flatten(mTimes); ❶
var average = function(l){
  var sum = _.reduce(l, function(a,b){return a+b},0);
  return sum/l.length;
};
console.log('Average morning time is ' + average(mTimes));
Out:
Average morning time is 33.81818181818182
```

❶ Our array of morning times arrays ([[44, 34, 56, 31], [33…]]) needs to be *flattened* into a a single array of numbers.

Functional Array Methods and List Comprehensions

I find myself using underscore a lot less since the addition, with Ecmascript 5, of functional methods to JavaScript arrays. I don't think I've used a conventional `for` loop since then, which, given the ugliness of JS `for` loops, is a very good thing.

Once you get used to processing arrays functionally, it's hard to consider going back. Combined with JS's anonymous functions, it makes for very fluid, expressive programming. It's also an area where method chaining seems very natural. Let's look at a highly contrived example:

```
let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let sum = nums.filter(function(o){ return o%2 }) ❶
  .map(function(o){ return o * o}) ❷
  .reduce(function(total, current){return total + next}); ❸

console.log('Sum of the odd squares is ' + sum);
```

❶ Filters the list for odd numbers (i.e., returning `1` for the modulus (%) 2 operation).

❷ `map` produces a new list by applying a function to each member (i.e., [1, 3, 5…] → [1, 9, 25…]).

❸ `reduce` processes the resultant mapped list in sequence, providing the current (in this case, summed) value (total) and the item value (next). By default, the initial value of the first argument (total) is 0.

Python's powerful list comprehensions can emulate the previous example easily enough:

```
nums = range(10) ❶

odd_squares = [x * x for x in nums if x%2] ❷
sum(odd_squares) ❸
Out:
165
```

❶ Python has a handy built-in `range` function, which can also take a start, end, and step (e.g., range(2, 8, 2) → [2, 4, 6])

❷ The `if` condition tests for oddness of `x`, and any numbers passing this filter are squared and inserted into the list.

❸ Python also has a built-in and often used `sum` statement.

> **TIP**
>
> Python's list comprehensions can use recursive control structures, such as applying a second `for`/`if` expression to the iterated items. Although this can create terse and powerful lines of code, it goes against the grain of Python's readability and I discourage its use. Even simple list comprehensions are less than intuitive and, as much as it appeals to the leet hacker in all of us, you risk creating incomprehensible code.

Python's list comprehensions work well for basic filtering and mapping. They do lack the convenience of JavaScript's anonymous functions (which are fully fledged, with their own scope, control blocks, exception handling, etc.), but there are arguments against the use of anonymous functions. For example, they are not reusable and, being unnamed, they make it hard to follow exceptions and debug. See here for some persuasive arguments. Having said that, for libraries like D3, replacing the small, throwaway anonymous functions used to set DOM attributes and properties with named ones would be far too onerous and would just add to the boilerplate.

Python does have functional lambda expressions, which we'll look at in the next section, but for full functional processing in Python by necessity and JavaScript for best practice, we can use named functions to increase our control scope. For our simple odd-squares example, named functions are a contrivance—but note that they increase the first-glance readability of the list comprehension, which becomes much more important as your functions get more complex.

```
items = [1, 2, 3, 4, 5]

def is_odd(x):
    return x%2

def sq(x):
    return x * x

sum([sq(x) for x in items if is_odd(x)])
```

With JavaScript, a similar contrivance can also increase readability and facilitate DRY code:[24]

```
var isOdd = function(x){ return x%2; };

sum = l.filter(isOdd)
...
```

Map, Reduce, and Filter with Python's Lambdas

Although Python lacks anonymous functions, it does have *lambdas*, which are nameless expressions that take arguments. Though lacking the bells and whistles of JavaScript's anonymous functions, these are a powerful addition to Python's functional programming repertoire, especially when combined with its functional methods.

> **NOTE**
>
> Python's functional built-ins (`map`, `reduce`, `filter` methods, and lambda expressions) have a checkered past. It's no secret that the creator of Python wanted to remove them from the language. The clamor of disapproval led to their reluctant preservation. With the recent trend toward functional programming, this looks like a very good thing. They're not perfect but are far better than nothing. And given JavaScript's strong functional emphasis, they're a good way to leverage skills acquired in that language.

Python's lambdas take a number of parameters and return an operation on them, using a colon separator to define the function block, in much the same way that standard Python functions only pared to the bare essentials and with an implicit return. The following example shows a few lambdas employed in functional programming:

```
from functools import reduce # if using Python 3+

nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

odds = filter(lambda x: x % 2, nums)
odds_sq = map(lambda x: x * x, odds)
reduce(lambda x, y: x + y, odds_sq) ❶
Out:
165
```

❶ Here, the `reduce` method provides two arguments to the lambda, which uses them to return the expression after the colon.

JavaScript Closures and the Module Pattern

One of the key concepts in JavaScript is that of the *closure*, which is essentially a nested function declaration that uses variables declared in an outer (but not global) scope that are *kept alive* after the function is returned. Closures allow for a number of very useful programming patterns and are a common feature of the language.

Let's look at possibly the most common usage of closures and one we've already seen exploited in our module pattern (Example 1-2): exposing a limited API while having access to essentially private member variables.

A simple example of a closure is this little counter:

```
function Counter(inc) {
  let count = 0;
  let add = function() { ❶
    count += inc;
    console.log('Current count: ' + count);
  }
  return add;
}
```

```
let inc2 = Counter(2); ❷
inc2(); ❸
Out:
Current count: 2
inc2();
Out:
Current count: 4
```

❶ The `add` function gets access to the essentially private, outer-scope `count` and `inc` variables.

❷ This returns an `add` function with the closure variables, `count` (0) and `inc` (2).

❸ Calling `inc2` calls `add`, updating the *closed* `count` variable.

We can extend the `Counter` to add a little API. This technique is the basis of JavaScript modules and many simple libraries, particularly when using script-based JavaScript [25]. In essence, it selectively exposes public methods while hiding private methods and variables, which is generally seen as good practice in the programming world:

```
function Counter(inc) {
  let count = 0;
  let api = {};
  api.add = function() {
    count += inc;
    console.log('Current count: ' + count);
  }
  api.sub = function() {
    count -= inc;
    console.log('Current count: ' + count)
  }
  api.reset = function() {
    count = 0;
    console.log('Count reset to 0')
  }

  return api;
}

cntr = Counter(3);
cntr.add() // Current count: 3
cntr.add() // Current count: 6
cntr.sub() // Current count: 3
cntr.reset() // Count reset to 0
```

Closures have all sorts of uses in JavaScript and I'd recommend getting your head around them—you'll see them a lot as you start investigating other people's code. These are three particularly good web articles that provide a lot of good use cases for closures:

- Mozilla's introduction
- "JavaScript Module Pattern: In-Depth" by Ben Cherry
- "Use Cases for JavaScript Closures" by Juriy Zaytsev

Python has closures, but they are not used nearly as much as JavaScript's, perhaps because of a few quirks that, though surmountable, make for some slightly awkward code. To demonstrate, Example 1-7 tries to replicate the previous JavaScript counter.

Example 1-7. A first-pass attempt at a Python counter closure

```
def get_counter(inc):
    count = 0
```

```
    def add():
        count += inc
        print('Current count: ' + str(count))
    return add
```

If you create a counter with `get_counter` (Example 1-7) and try to run it, you'll get an `UnboundLocalError`:

```
cntr = get_counter(2)
cntr()
Out:
...
UnboundLocalError: local variable 'count' referenced before
assignment
```

Interestingly, although we can read the value of `count` within the `add` function (comment out the `count += inc` line to try it), attempts to change it throw an error. This is because attempts to assign a value to something in Python assume it is local in scope. There is no `count` local to the `add` function and so an error is thrown.

In Python 3, we can get around the error in Example 1-7 by using the `nonlocal` keyword to tell Python that `count` is in a nonlocal scope:

```
...
def add():
    nonlocal count
    count += inc
...
```

If you are obliged to use Python 2+ (please try and upgrade), we can use a little dictionary hack to allow mutation of our closed variables:

```
def get_counter(inc):
    vars = {'count': 0}
    def add():
        vars['count'] += inc
        print('Current count: ' + str(vars['count']))
    return add
```

This *hack* works because we are not assigning a new value to `vars` but are instead mutating an existing container, which is perfectly valid even if it is out of local scope.

As you can see, with a bit of effort, JavaScripters can transfer their closure skills to Python. The use cases are similar, but Python, being a richer language with lots of useful batteries included, has more options to apply to the same problem. Probably the most common use of closures is in Python's decorators.

Decorators are essentially function wrappers that extend the function's utility without having to alter the function itself. They're a relatively advanced concept, but you can find a user-friendly introduction on The Code Ship website.

This Is That

One JavaScript hack you'll see a lot of is a consequence of closures and the slippery `this` keyword. If you wish to refer to the outer-scoped `this` in a child function, you must use a proxy because the child's `this` will be bound according to context. The convention is to use `that` to refer to `this`. The code is less confusing than the explanation:

```
function outer(bar){
   this.bar = bar;
   var that = this;
```

```
function inner(baz){
    this.baz = baz * that.bar; ❶
    // ...
```

❶ `that` refers to the outer function's `this`.

This concludes my cherry-picked selection of patterns and hacks that I find myself using a lot in dataviz work. You'll doubtless acquire your own, but I hope these give you a leg up.

A Cheat Sheet

As a handy reference guide, Figures 1-2 to 1-7 include a set of cheat sheets to translate basic operations between Python and JavaScript.

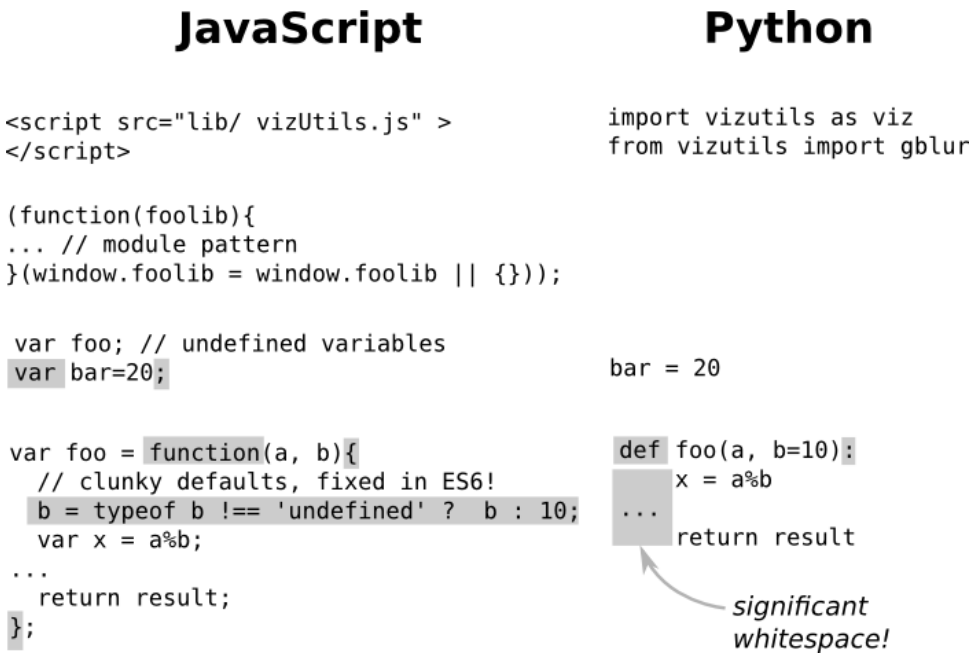


Figure 1-2. Some basic syntax

Figure 1-3. Booleans

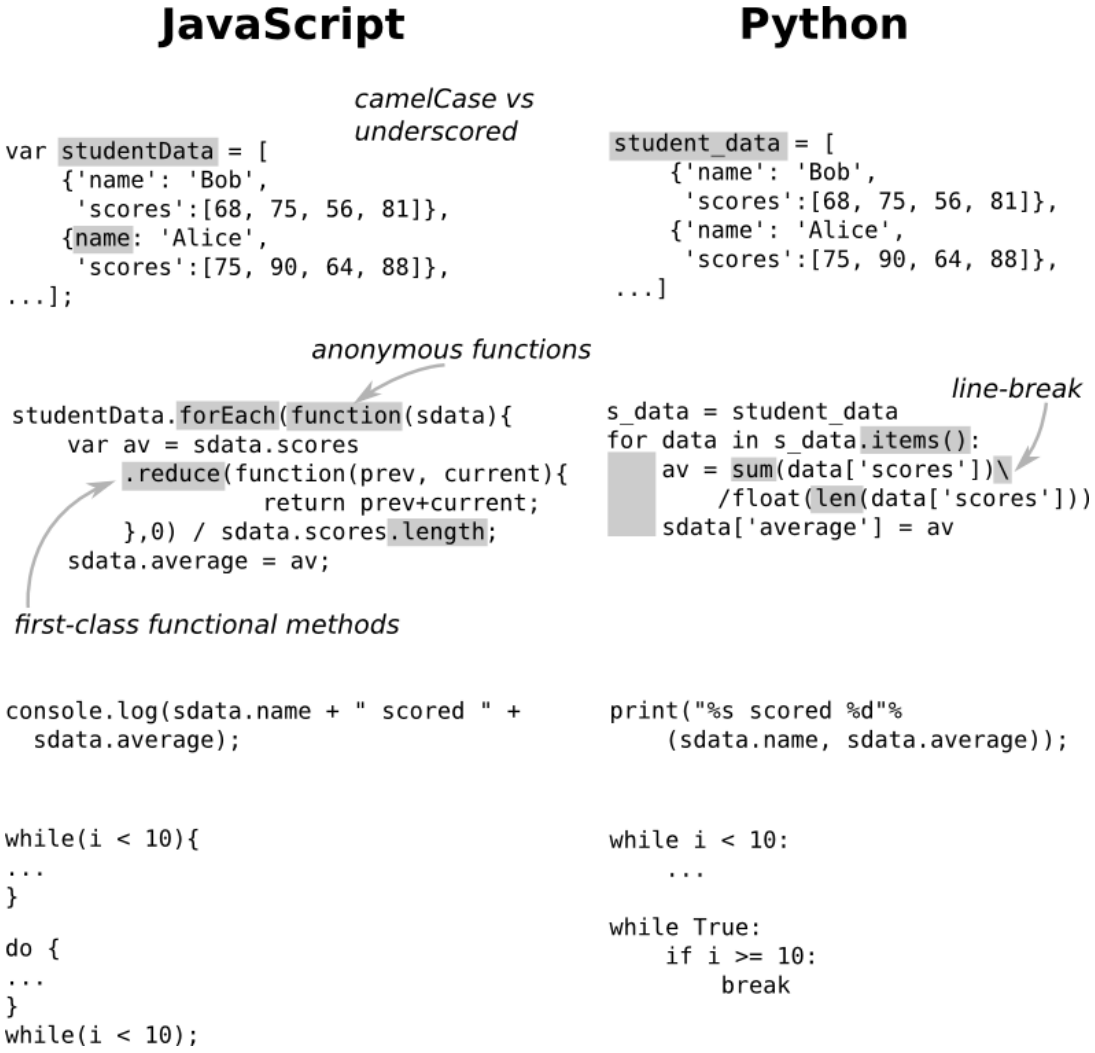


Figure 1-4. Loops and iterations

JavaScript

```
if(x === 'foo'){
    ...}
else if(x === 'bar'){
    ...}
else{
    ...}

if(x === foo && y !== bar){...

if(['foo', 'bar', 'baz']
     .indexOf(s) != -1){...

switch(foo){
  case bar:
    ...
    break;
  case baz: ...
  default:
    return false;
}
```

Python

```
if x == 'foo':
    ...
elif x == 'bar':
    ...
else:
    ...

if x == foo and y != bar:
    ...
if s in ['foo', 'bar', 'baz']:
    ...
```

Figure 1-5. Conditionals

JavaScript

```
var l = [1, 2, 3, 4];
l.push('foo'); // [...4, 'foo']
l.pop(); // 'foo', l=[..., 4]
l.slice(1,3) // [2, 3]
l.slice(-3, -1) // [2, 3]


l.map(function(o){ return o*o;})
// [1, 4, 9, 16]


d = {a:1, b:2, c:3};
d.a === d['a'] // 1
d.z // undefined

// OLD BROWSERS
for(key in d){
  if(d.hasOwnProperty(key){
    var item = d[key];

// NEW AND BETTER
Object.keys(d).forEach(key, i){
  var item = d[key];
```

Python

```
l = [1, 2, 3, 4]
l.append('foo') # [...4, 'foo']
l.pop() # 'foo', l=[..., 4]
l[1:3] # [2, 3]
l[-3:-1] # [2, 3]
l[0:4:2] # [1, 3] (stride of 2)

[o*o for o in l]
// [1, 4, 9, 16]


d = {'a':1, 'b':2, 'c':3}
d['a'] # 1
d.get('z') # NoneType
d['z'] # KeyError!

for key, value in d.items(): ...
for key in d:
for value in d.values():...
```

Figure 1-6. Containers

JavaScript

```
var Foo = {
  initFoo: function(bar){
    this.bar = bar;
    return this;
  }
};

var Baz = Object.create(Foo);

Baz.initBaz = function(bar, qux){
  this.initFoo(bar);
  this.qux = qux;
  return this;
};

var baz = Object.create(Baz)
            .initBaz('answer', 42);
```

Python

```
class Foo(object):
    def __init__(self, bar):
        self.bar = bar

class Baz(Foo):
    def __init__(self, bar, qux):
        super(Baz).__init__(bar)
        self.qux = qux

baz = Baz('answer', 42)
baz.bar # 'answer'
```

Figure 1-7. Classes and prototypes

Summary

I hope this chapter has shown that JavaScript and Python have a lot of common syntax and that most common idioms and patterns from one of the languages can be expressed in the other without too much fuss. The meat and potatoes of programming, iteration, conditionals, and basic data manipulation is simple in both languages, and the translation of functions is straightforward. If you can program in one to any degree of competency, the threshold to entry for the other is low. That's the huge appeal of these simple scripting languages, which have a lot of common heritage.

I provided a list of patterns, hacks, and idioms I find myself using frequently in dataviz work. I'm sure this list has its idiosyncrasies, but I've tried to tick the obvious boxes.

Treat this as part tutorial, part reference for the chapters to come. Anything not covered here will be dealt with when introduced.

---

1 One particularly annoying little gotcha is that while Python uses `pop` to remove a list item, it uses `append`—not `push`—to add an item. JavaScript uses `push` to add an item, whereas `append` is used to concatenate arrays.

2 The ascent of node.js has extended JavaScript to the server.

3 This is changing with libraries like TensorFlowJS and DanfoJS (a JavaScript Pandas-alike based on TensorFlow), but JS is still well behind Python, R, and others.

4 This version is based on the Qt GUI library.

5 At the cost of a running a Python interpreter on the server.

6 The constraint of having to deliver JS scripts over the Web via HTTP is largely responsible for this.

7 This means any blocking-script-loading calls occur after the page's HTML has rendered.

8 You can eliminate the possibility of a missing `var` by using the Ecmascript 5 `'use strict'` directive.

9 This is a good thing for reasons outlined in PEP 3105.

10 This is actually done by JavaScript compressors to reduce the file size of downloaded web pages.

11 The soft versus hard tab debate generates controversy, with much heat and little light. *PEP 8* stipulates spaces, which is good enough for me.

12 It could be two or even three spaces, but this number must be consistent throughout the module.

13 The quite fair assumption that JavaScript uses UTF-16 has been the cause of much bug-driven misery. See here for an interesting analysis.

14 The change to Unicode strings in Python 3 is a big one. Given the confusion that often attends Unicode de/encoding, it's worth reading a little bit about it. Python 2 used strings of bytes.

15 `parseInt` can do quite a bit more than round. For example, `parseInt(12.5px)` gives 12, first removing the *px* and then casting the string to a number. It also has a second `radix` argument to specify the base of the cast. See here for the specifics.

16 Because all numbers in JavaScript are floating point, it can only support 53-bit integers. Using larger integers (such as the commonly used 64 bit) can result in discontinuous integers. See *http://www.2ality.com/2012/07/large-integers.html* for further information.

17 This makes iterating over their properties a little trickier than it might be. See here for more details.

18 For the curious, there's a nice summation here.

19 This is one area where JS beats Python hands down and which finds many of us wishing for similar functionality in Python.

20 Added with Ecmascript 5 and available on all modern browsers.

21 I mentioned to a talented programmer friend that I was faced with the challenge of explaining prototypes to Python programmers and he pointed out that most JavaScripters could probably do with some pointers too. There's a lot of truth in this and many JSers do manage to be productive by using prototypes in a *classy* way, hacking their way around the edge cases.

22 This is another reason to use Ecmascript 5's `'use strict;'` injunction, which calls attention to such mistakes.

23 My personal choice for performance reasons.

24 Don't Repeat Yourself (DRY) is a solid coding convention.

25 Modern JavaScript has proper modules which can import and export encapsulated variables. There is an overhead to using these as they currently require a build-phase to make ready for the browser.

Chapter 2. Reading and Writing Data with Python

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the third chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

One of the fundamental skills of any data visualizer is the ability to move data around. Whether your data is in an SQL database, a comma-separated-value (CSV) file, or in some more esoteric form, you should be comfortable reading the data, converting it, and writing it into a more convenient form if need be. One of Python's great strengths is how easy it makes manipulating data in this way. The focus of this chapter is to bring you up to speed with this essential aspect of our dataviz toolchain.

This chapter is part tutorial, part reference, and sections of it will be referred to in later chapters. If you know the fundamentals of reading and writing Python data, you can cherry-pick parts of the chapter as a refresher.

Easy Does It

I remember when I started programming back in the day (using low-level languages like C) how awkward data manipulation was. Reading from and

writing to files was an annoying mixture of boilerplate code, hand-rolled kludges, and the like. Reading from databases was equally difficult, and as for serializing data, the memories are still painful. Discovering Python was a breath of fresh air. It wasn't a speed demon, but opening a file was pretty much as simple as it could be:

```
file = open('data.txt')
```

Back then, Python made reading from and writing to files refreshingly easy, and its sophisticated string processing made parsing the data in those files just as easy. It even had an amazing module called Pickle that could serialize pretty much any Python object.

In the years since, Python has added robust, mature modules to its standard library that make dealing with CSV and JSON files, the standard for web dataviz work, just as easy. There are also some great libraries for interacting with SQL databases such as SQLAlchemy, my thoroughly recommended go-to. The newer NoSQL databases are also well served. MongoDB is the most popular of these newer document-based databases, and Python's `pymongo` library, which is demonstrated later in the chapter, makes interacting with it a relative breeze.

Passing Data Around

A good way to demonstrate how to use the key data-storage libraries is to pass a single data packet among them, reading and writing it as we go. This will give us an opportunity to see in action the key data formats and databases employed by data visualizers.

The data we'll be passing around is probably the most commonly used in web visualizations, a list of dictionary-like objects (see Example 2-1). This dataset is transferred to the browser in JSON form, which is, as we'll see, easily converted from a Python dictionary.

Example 2-1. Our target list of data objects

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
 {'category': 'Physics',
  'name': 'Paul Dirac',
  'nationality': 'British',
  'sex': 'male',
  'year': 1933},
 {'category': 'Chemistry',
  'name': 'Marie Curie',
  'nationality': 'Polish',
  'sex': 'female',
  'year': 1911}
]
```

We'll start by creating a CSV file from the Python list shown in Example 2-1 as a demonstration of reading (opening) and writing system files.

The following sections assume you're in a work directory with a *data* subdirectory at hand. You can run the code from a Python interpreter or file.

Working with System Files

In this section, we'll create a CSV file from a Python list of dictionaries (Example 2-1). Typically, you would do this using the `csv` module, which we'll demonstrate after this section, so this is just a way of demonstrating basic Python file manipulation.

First let's open a new file, using `w` as a second argument to indicate we'll be writing data to it.

```
f = open('data/nobel_winners.csv', 'w')
```

Now we'll create our CSV file from the `nobel_winners` dictionary (Example 2-1):

```
cols = nobel_winners[0].keys() ❶
cols = sorted(cols) ❷
```

```
with open('data/nobel_winners.csv', 'w') as f: ❸
    f.write(','.join(cols) + '\n') ❹

    for o in nobel_winners:
        row = [str(o[col]) for col in cols] ❺
        f.write(','.join(row) + '\n')
```

❶ Gets our data columns from the keys of the first object (i.e., `['category', 'name', ... ]`).

❷ Sorts the columns in alphabetical order.

❸ Uses Python's `with` statement to guarantee the file is closed on leaving the block or if any exceptions occur.

❹ `join` creates a concatenated string from a list of strings (`cols` here), joined by the initial string (i.e., "category,name,..").

❺ Creates a list using the column keys to the objects in `nobel_winners`.

Now that we've created our CSV file, let's use Python to read it and make sure everything is correct:

```
with open('data/nobel_winners.csv') as f:
    for line in f.readlines():
        print(line)

Out:
category,name,nationality,sex,year
Physics,Albert Einstein,Swiss,male,1921
Physics,Paul Dirac,British,male,1933
Chemistry,Marie Curie,Polish,female,1911
```

As the previous output shows, our CSV file is well formed. Let's use Python's built-in `csv` module to first read it and then create a CSV file the right way.

CSV, TSV, and Row-Column Data Formats

Comma-separated values (CSV) or their tab-separated cousins (TSV) are probably the most ubiquitous file-based data formats and, as a data visualizer, these will often be the forms you'll receive to work your magic with. Being able to read and write CSV files and their various quirky variants, such as pipe- or semicolon-separated or those using `` ` `` in place of the standard double quotes, is a fundamental skill; Python's `csv` module is capable of doing pretty much all your heavy lifting here. Let's put it through its paces reading and writing our `nobel_winners` data:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Writing our `nobel_winners` data (see Example 2-1) to a CSV file is a pretty simple affair. `csv` has a dedicated `DictWriter` class that will turn our dictionaries into CSV rows. The only piece of explicit bookkeeping we have to do is write a header to our CSV file, using the keys of our dictionaries as fields (i.e., "category, name, nationality, sex"):

```
import csv

with open('data/nobel_winners.csv', 'w') as f:
    fieldnames = nobel_winners[0].keys() ❶
    fieldnames = sorted(fieldnames) ❷
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader() ❸
    for w in nobel_winners:
        writer.writerow(w)
```

❶ You need to explicitly tell the writer which `fieldnames` (in this case, the `'category'`, `'name'`, etc., keys) to use.

❷ We'll sort the CSV header fields alphabetically for readability.

❸ Writes the CSV-file header ("category,name,…").

You'll probably be reading CSV files more often than writing them.[1] Let's read back the *nobel_winners.csv* file we just wrote.

If you just want to use `csv` as a superior and eminently adaptable file line-reader, a couple of lines will produce a handy iterator, which can deliver your CSV rows as lists of strings:

```
with open('data/nobel_winners.csv') as f:
    reader = csv.reader(f)
    for row in reader: ❶
        print(row)

Out:
['category', 'name', 'nationality', 'sex', 'year']
['Physics', 'Albert Einstein', 'Swiss', 'male', '1921']
['Physics', 'Paul Dirac', 'British', 'male', '1933']
['Chemistry', 'Marie Curie', 'Polish', 'female', '1911']
```

❶ Iterates over the `reader` object, consuming the lines in the file.

Note that the numbers are read in string form. If you want to manipulate them numerically, you'll need to convert any numeric columns to their respective type, which is integer years in this case.

A more convenient way to consume CSV data is to convert the rows into Python dictionaries. This *record* form is also the one we are using as our conversion target (a `list` of `dicts`). `csv` has a handy `DictReader` for just this purpose:

```
import csv

with open('data/nobel_winners.csv') as f:
    reader = csv.DictReader(f)
    nobel_winners = list(reader) ❶
```

```
nobel_winners

Out:
[{'category': 'Physics', 'nationality': 'Swiss', \
 'year': '1921', 'name': 'Albert Einstein', 'sex': 'male'},
{'category': 'Physics', 'nationality': 'British', \
 'year': '1933', 'name': 'Paul Dirac', 'sex': 'male'},
{'category': 'Chemistry', 'nationality': 'Polish', \
 'year': '1911', 'name': 'Marie Curie', 'sex': 'female'}]
```

❶ Inserts all of the `reader` items into a list.

As the output shows, we just need to cast the `dicts` year attributes to integers to make `nobel_winners` conform to the chapter's target data (Example 2-1), thus:

```
for w in nobel_winners:
    w['year'] = int(w['year'])
```

The `csv` readers don't infer datatypes from your file, and instead interpret everything as a string. Pandas, Python's preeminent data-hacking library, will try to guess the correct type of the data columns, usually successfully. We'll see this in action in the later dedicated Pandas chapters.

`csv` has a few useful arguments to help parse members of the CSV family:

`dialect`

By default, `'excel'`; specifies a set of dialect-specific parameters. `excel-tab` is a sometimes used alternative.

`delimiter`

Files are usually comma-separated, but they could use `|`, `:` or `' '` instead.

`quotechar`

By default, double quotes are used, but you occasionally find `|` or `` ` `` instead.

You can find the full set of `csv` parameters in the online Python docs.

Now that we've successfully written and read our target data using the `csv` module, let's pass on our CSV-derived `nobel_winners` `dict` to the `json` module.

JSON

In this section we'll write and read our `nobel_winners` data using Python's `json` module. Let's remind ourselves of the data we're using:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

For data primitives such as strings, integers, and floats, Python dictionaries are easily saved (or *dumped*, in the JSON vernacular) into JSON files, using the `json` module. The `dump` method takes a Python container and a file pointer, saving the former to the latter:

```
import json

with open('data/nobel_winners.json', 'w') as f:
     json.dump(nobel_winners, f)

open('data/nobel_winners.json').read()

Out: '[{"category": "Physics", "name": "Albert Einstein",
"sex": "male", "person_data": {"date of birth": "14th March
1879", "date of death": "18th April 1955"}, "year": 1921,
"nationality": "Swiss"}, {"category": "Physics",
```

```
"nationality": "British", "year": 1933, "name": "Paul Dirac",
"sex": "male"}, {"category": "Chemistry", "nationality":
"Polish", "year": 1911, "name": "Marie Curie", "sex":
"female"}]'
```

Reading (or loading) a JSON file is just as easy. We just pass the opened JSON file to the `json` module's `load` method:

```
import json

with open('data/nobel_winners.json') as f:
    nobel_winners = json.load(f)

nobel_winners
Out:
[{u'category': u'Physics',
  u'name': u'Albert Einstein',
  u'nationality': u'Swiss',
  u'sex': u'male',
  u'year': 1921}, ❶
...
```

❶ Note that, unlike in our CSV file conversion, the integer type of the year column is preserved.

`json` has the methods `loads` and `dumps`, which are counterparts to the file access methods, loading JSON strings to Python containers and dumping Python containers to JSON strings.

Dealing with Dates and Times

Trying to dump a `datetime` object to `json` produces a `TypeError`:

```
from datetime import datetime

json.dumps(datetime.now())
Out:
...
TypeError: datetime.datetime(2021, 9, 13, 10, 25, 52, 586792)
is not JSON serializable
```

When serializing simple datatypes such as strings or numbers, the default `json` encoders and decoders are fine. But for more specialized data such as dates, you will need to do your own encoding and decoding. This isn't as hard as it sounds and quickly becomes routine. Let's first look at encoding your Python `datetimes` into sensible JSON strings.

The easiest way to encode Python data containing `datetimes` is to create a custom encoder like the one shown in Example 2-2, which is provided to the `json.dumps` method as a `cls` argument. This encoder is applied to each object in your data in turn and converts dates or datetimes to their ISO-format string (see "Dealing with Dates, Times, and Complex Data").

Example 2-2. Encoding a Python datetime to JSON

```
import datetime
from dateutil import parser
import json


class JSONDateTimeEncoder(json.JSONEncoder): ❶
    def default(self, obj):
        if isinstance(obj, (datetime.date, datetime.datetime)): ❷
            return obj.isoformat()
        else:
            return json.JSONEncoder.default(self, obj)

def dumps(obj):
    return json.dumps(obj, cls=JSONDateTimeEncoder) ❸
```

❶ Subclasses a `JSONEncoder` in order to create customized date-handling one.

❷ Tests for a `datetime` object and if true, returns the `isoformat` of any dates or datetimes (e.g., *2021-11-16T16:41:14.650802*).

❸ Uses the `cls` argument to set a custom date encoder.

Let's see how our new `dumps` method copes with some `datetime` data:

```
now_str = dumps({'time': datetime.datetime.now()})
now_str
```

```
Out:
'{"time": "2021-11-16T16:41:14.650802"}'
```

The `time` field is correctly converted into an ISO-format string, ready to be decoded into a JavaScript `Date` object (see "Dealing with Dates, Times, and Complex Data" for a demonstration).

While you could write a generic decoder to cope with date strings in arbitrary JSON files,[2] it's probably not advisable. Date strings come in so many weird and wonderful varieties that this is a job best done by hand on what is pretty much always a known dataset.

The venerable `strptime` method, part of the `datetime.datetime` package, is good for the job of turning a time string in a known format into a Python `datetime` instance:

```
In [0]: from datetime import datetime

In [1]: time_str = '2021/01/01 12:32:11'

In [2]: dt = datetime.strptime(time_str, '%Y/%m/%d %H:%M:%S') ❶

In [3]: dt
Out[2]: datetime.datetime(2021, 1, 1, 12, 32, 11)
```

❶ `strptime` tries to match the time string to a format string using various directives such as `%Y` (year with century) and `%H` (hour as a zero-padded decimal number). If successful, it creates a Python `datetime` instance. See the Python docs for a full list of the directives available.

If `strptime` is fed a time string that does not match its format, it throws a handy `ValueError`:

```
dt = datetime.strptime('1/2/2021 12:32:11', '%Y/%m/%d %H:%M:%S')
---------------------------------------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-111-af657749a9fe> in <module>()
----> 1 dt = datetime.strptime('1/2/2021 12:32:11',\
```

```
    '%Y/%m/%d %H:%M:%S')
...
ValueError: time data '1/2/2021 12:32:11' does not match
            format '%Y/%m/%d %H:%M:%S'
```

So to convert date fields of a known format into `datetimes` for a `data` list of dictionaries, you could do something like this:

```
data = [
    {'id': 0, 'date': '2020/02/23 12:59:05'},
    {'id': 1, 'date': '2021/11/02 02:32:00'},
    {'id': 2, 'date': '2021/23/12 09:22:30'},
]

for d in data:
     try:
         d['date'] = datetime.strptime(d['date'],\
           '%Y/%m/%d %H:%M:%S')
     except ValueError:
         print('Oops! - invalid date for ' + repr(d))
Out:
Oops! - invalid date for {'id': 2, 'date': '2021/23/12 09:22:30'}
```

Now that we've dealt with the two most popular data file formats, let's shift to the big guns and see how to read our data from and write our data to SQL and NoSQL databases.

SQL

For interacting with an SQL database, SQLAlchemy is the most popular and, in my opinion, best Python library. It allows you to use raw SQL instructions if speed and efficiency is an issue, but also provides a powerful object-relational mapping (ORM) that allows you to operate on SQL tables using a high-level, Pythonic API, treating them essentially as Python classes.

Reading and writing data using SQL while allowing the user to treat that data as a Python container is a complicated process, and though SQLAlchemy is considerably more user-friendly than a low-level SQL

engine, it is still a fairly complex library. I'll cover the basics here, using our data as a target, but encourage you to spend a little time reading some of the rather excellent documentation on SQLAlchemy. Let's remind ourselves of the `nobel_winners` dataset we're aiming to write and read:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Let's first write our target data to an SQLite file using SQLAlchemy, starting by creating the database engine.

Creating the Database Engine

The first thing you need to do when starting an SQLAlchemy session is to create a database engine. This engine will establish a connection with the database in question and perform any conversions needed to the generic SQL instructions generated by SQLAlchemy and the data being returned.

There are engines for pretty much every popular database, as well as a *memory* option, which holds the database in RAM, allowing fast access for testing.[3] The great thing about these engines is that they are interchangeable, which means you could develop your code using the convenient file-based SQLite database and then switch during production to something a little more industrial, such as Postgresql, by changing a single config string. Check SQLAlchemy for the full list of engines available.

The form for specifying a database URL is:

```
dialect+driver://username:password@host:port/database
```

So, to connect to a `'nobel_winners'` MySQL database running on localhost requires something like the following. Note that

`create_engine` does not actually make any SQL requests at this point, but merely sets up the framework for doing so.[4]

```
engine = create_engine(\
           'mysql://kyran:mypsswd@localhost/nobel_winners')
```

We'll use a file-based SQLite database, setting the `echo` argument to `True`, which will output any SQL instructions generated by SQLAlchemy. Note the use of three backslashes after the colon:

```
from sqlalchemy import create_engine

engine = create_engine(\
            'sqlite:///data/nobel_winners.db', echo=True)
```

SQLAlchemy offers various ways to engage with databases, but I recommend using the more recent declarative style unless there are good reasons to go with something more low-level and fine-grained. In essence, with declarative mapping, you subclass your Python SQL-table classes from a base, and SQLAlchemy introspects their structure and relationships. See SQLAlchemy for more details.

Defining the Database Tables

We first create a `Base` class using `declarative_base`. This base will be used to create table classes, from which SQLAlchemy will create the database's table schemas. You can use these table classes to interact with the database in a fairly Pythonic fashion:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

Note that most SQL libraries require you to formally define table schemas. This is in contrast to such schema-less NoSQL variants as MongoDB. We'll take a look at the Dataset library later in this chapter, which enables schemaless SQL.

Using this `Base`, we define our various tables—in our case, a single `Winner` table. Example 2-3 shows how to subclass `Base` and use SQLAlchemy's datatypes to define a table schema. Note the `__tablename__` member, which will be used to name the SQL table and as a keyword to retrieve it, and the optional custom `__repr__` method, which will be used when printing a table row.

Example 2-3. Defining an SQL database table

```
from sqlalchemy import Column, Integer, String, Enum
// ...

class Winner(Base):
    __tablename__ = 'winners'
    id = Column(Integer, primary_key=True)
    category = Column(String)
    name = Column(String)
    nationality = Column(String)
    year = Column(Integer)
    sex = Column(Enum('male', 'female'))
    def __repr__(self):
        return "<Winner(name='%s', category='%s', year='%s')>"\
%(self.name, self.category, self.year)
```

Having declared our `Base` subclass in Example 2-3, we supply its `metadata` `create_all` method with our database engine to create our database.[5] Because we set the `echo` argument to `True` when creating the engine, we can see the SQL instructions generated by SQLAlchemy from the command line:

```
Base.metadata.create_all(engine)

2021-11-16 17:58:34,700 INFO sqlalchemy.engine.Engine BEGIN
(implicit)
...
CREATE TABLE winners (
        id INTEGER NOT NULL,
        category VARCHAR,
        name VARCHAR,
        nationality VARCHAR,
        year INTEGER,
        sex VARCHAR(6),
        PRIMARY KEY (id)
```

```
)...
2021-11-16 17:58:34,742 INFO sqlalchemy.engine.Engine COMMIT
```

With our new `winners` table declared, we can start adding winner instances to it.

Adding Instances with a Session

Now that we have created our database, we need a session to interact with:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

We can now use our `Winner` class to create instances and table rows and add them to the session:

```
albert = Winner(**nobel_winners[0]) ❶
session.add(albert)
session.new ❷
Out:
IdentitySet([<Winner(name='Albert Einstein', category='Physics',
             year='1921')>])
```

❶ Python's handy ** operator unpacks our first `nobel_winners` member into key-value pairs: `(name='Albert Einstein', category='Physics'...)`.

❷ `new` is the set of any items that have been added to this session.

Note that all database insertions and deletions take place in Python. It's only when we use the `commit` method that the database is altered.

> **TIP**
>
> Use as few commits as possible, allowing SQLAlchemy to work its magic behind the scenes. When you commit, your various database manipulations should be summarized by SQLAlchemy and communicated in an efficient fashion. Commits involve establishing a database handshake and negotiating transactions, which is often a slow process and one you want to limit as much as possible, leveraging SQLAlchemy's bookkeeping abilities to full advantage.

As the `new` method shows, we have added a `Winner` to the session. We can remove the object using `expunge`, leaving an empty `IdentitySet`:

```
session.expunge(albert) ❶
session.new
Out:
IdentitySet([])
```

❶ Remove the instance from the session (there is an `expunge_all` method that removes all new objects added to the session).

At this point, no database insertions or deletions have taken place. Let's add all the members of our `nobel_winners` list to the session and commit them to the database:

```
winner_rows = [Winner(**w) for w in nobel_winners]
session.add_all(winner_rows)
session.commit()
Out:
INFO:sqlalchemy.engine.base.Engine:BEGIN (implicit)
...
INFO:sqlalchemy.engine.base.Engine:INSERT INTO winners (name,
category, year, nationality, sex) VALUES (?, ?, ?, ?, ?)
INFO:sqlalchemy.engine.base.Engine:(u'Albert Einstein',
u'Physics', 1921, u'Swiss', u'male')
...
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

Now that we've committed our `nobel_winners` data to the database, let's see what we can do with it and how to recreate the target list in

Example 2-1.

Querying the Database

To access data, you use the `session`'s `query` method, the result of which can be filtered, grouped, and intersected, allowing the full range of standard SQL data retrieval. You can check out available querying methods in the SQLAlchemy docs. For now, I'll quickly run through some of the most common queries on our Nobel dataset.

Let's first count the number of rows in our winners' table:

```
session.query(Winner).count()
Out:
3
```

Next, let's retrieve all Swiss winners:

```
result = session.query(Winner).filter_by(nationality='Swiss') ❶
list(result)
Out:
[<Winner(name='Albert Einstein', category='Physics',\
  year='1921')>]
```

❶ `filter_by` uses keyword expressions; its SQL expressions counterpart is `filter`—for example, `filter(Winner.nationality == Swiss)`.

Now let's get all non-Swiss Physics winners:

```
result = session.query(Winner).filter(\
            Winner.category == 'Physics', \
            Winner.nationality != 'Swiss')
list(result)
Out:
[<Winner(name='Paul Dirac', category='Physics', year='1933')>]
```

Here's how to get a row based on ID number:

```
session.query(Winner).get(3)
Out:
<Winner(name='Marie Curie', category='Chemistry', year='1911')>
```

Now let's retrieve winners ordered by year:

```
res = session.query(Winner).order_by('year')
list(res)
Out:
[<Winner(name='Marie Curie', category='Chemistry',\
year='1911')>,
 <Winner(name='Albert Einstein', category='Physics',\
year='1921')>,
 <Winner(name='Paul Dirac', category='Physics', year='1933')>]
```

To reconstruct our target list requires a little effort converting the `Winner` objects returned by our session query into Python `dicts`. Let's write a little function to create a `dict` from an SQLAlchemy class. We'll use a little table introspection to get the column labels (see Example 2-4).

Example 2-4. Converts an SQLAlchemy instance to a dict

```
def inst_to_dict(inst, delete_id=True):
    dat = {}
    for column in inst.__table__.columns: ❶
        dat[column.name] = getattr(inst, column.name)
    if delete_id:
        dat.pop('id') ❷
    return dat
```

❶ Accesses the instance's table class to get a list of column objects.

❷ If `delete_id` is true, remove the SQL primary ID field.

We can use Example 2-4 to reconstruct our `nobel_winners` target list:

```
winner_rows = session.query(Winner)
nobel_winners = [inst_to_dict(w) for w in winner_rows]
nobel_winners
Out:
[{'category': u'Physics',
  'name': u'Albert Einstein',
  'nationality': u'Swiss',
```

```
  'sex': u'male',
  'year': 1921},
  ...
]
```

You can update database rows easily by changing the property of their reflected objects:

```
marie = session.query(Winner).get(3) ❶
marie.nationality = 'French'
session.dirty ❷
Out:
IdentitySet([<Winner(name='Marie Curie', category='Chemistry',
year='1911')>])
```

❶ Fetches Marie Curie, nationality Polish.

❷ `dirty` shows any changed instances not yet committed to the database.

Let's commit `Marie`'s changes and check that her nationality has changed from Polish to French:

```
session.commit()
Out:
INFO:sqlalchemy.engine.base.Engine:UPDATE winners SET
nationality=? WHERE winners.id = ?
INFO:sqlalchemy.engine.base.Engine:('French', 3)
...

session.dirty
Out:
IdentitySet([])

session.query(Winner).get(3).nationality
Out:
'French'
```

In addition to updating database rows, you can delete the results of a query:

```
session.query(Winner).filter_by(name='Albert Einstein').delete()
Out:
```

```
INFO:sqlalchemy.engine.base.Engine:DELETE FROM winners WHERE
winners.name = ?
INFO:sqlalchemy.engine.base.Engine:('Albert Einstein',)
1

list(session.query(Winner))
Out:
[<Winner(name='Paul Dirac', category='Physics', year='1933')>,
 <Winner(name='Marie Curie', category='Chemistry',\
 year='1911')>]
```

You can also drop the whole table if required, using the declarative class's `__table__` attribute:

```
Winner.__table__.drop(engine)
```

In this section, we've dealt with a single winners table, without any foreign keys or relationship to any other tables, akin to a CSV or JSON file. SQLAlchemy adds the same level of convenience in dealing with many-to-one, one-to-many, and other database table relationships as it does to basic querying using implicit joins, by supplying the `query` method with more than one table class or explicitly using the query's `join` method. Check out the examples in the SQLAlchemy docs for more details.

Easier SQL with Dataset

One library I've found myself using a fair deal recently is Dataset, a module designed to make working with SQL databases a little easier and more Pythonic than existing powerhouses like SQLAlchemy.[6] Dataset tries to provide the same degree of convenience you get when working with schema-less NoSQL databases such as MongoDB by removing a lot of the formal boilerplate, such as schema definitions, which are demanded by the more conventional libraries. Dataset is built on top of SQLAlchemy, which means it works with pretty much all major databases and can exploit the power, robustness, and maturity of that best-of-breed library. Let's see how it deals with reading and writing our target dataset (from Example 2-1).

Let's use the SQLite *nobel_winners.db* database we've just created to put Dataset through its paces. First we connect to our SQL database, using the same URL/file format as SQLAlchemy:

```
import dataset

db = dataset.connect('sqlite:///data/nobel_winners.db')
```

To get our list of winners, we grab a table from our `db` database, using its name as a key, and then use the `find` method without arguments to return all winners:

```
wtable = db['winners']
winners = wtable.find()
winners = list(winners)
winners
Out:
[OrderedDict([(u'id', 1), (u'name', u'Albert Einstein'),
 (u'category', u'Physics'), (u'year', 1921), (u'nationality',
 u'Swiss'), (u'sex', u'male')]), OrderedDict([(u'id', 2),
 (u'name', u'Paul Dirac'), (u'category', u'Physics'),
 (u'year', 1933), (u'nationality', u'British'), (u'sex',
 u'male')]), OrderedDict([(u'id', 3), (u'name', u'Marie
 Curie'), (u'category', u'Chemistry'), (u'year', 1911),
 (u'nationality', u'Polish'), (u'sex', u'female')])]
```

Note that the instances returned by Dataset's `find` method are `OrderedDicts`. These useful containers are an extension of Python's `dict` class and behave just like dictionaries except that they remember the order in which items were inserted, meaning you can guarantee the result of iteration, pop the last item inserted, and more. This is a very handy additional functionality.

> **TIP**
>
> One of the most useful Python "batteries" for data-mungers is `collections`, which is where `Dataset`'s `OrderedDicts` come from. The `defaultdict` and `Counter` classes are particularly useful. Check out what's available in the Python docs.

Let's recreate our winners table with `Dataset`, first dropping the existing one:

```
wtable = db['winners']
wtable.drop()

wtable = db['winners']
wtable.find()
Out:
[]
```

To recreate our dropped winners table, we don't need to define a schema as with SQLAlchemy (see "Defining the Database Tables"). Dataset will infer that from the data we add, doing all the SQL creation implicitly. This is the kind of convenience one is used to when working with collection-based NoSQL databases. Let's use our `nobel_winners` dataset (Example 2-1) to insert some winner dictionaries. We use a database transaction and the `with` statement to efficiently insert our objects and then commit them.[7]

```
with db as tx: ❶
     tx['winners'].insert_many(nobel_winners)
```

❶ Use the `with` statement to guarantee the transaction `tx` is committed to the database.

Let's check that everything has gone well:

```
list(db['winners'].find())
Out:
[OrderedDict([(u'id', 1), (u'name', u'Albert Einstein'),
(u'category', u'Physics'), (u'year', 1921), (u'nationality',
u'Swiss'), (u'sex', u'male')]),
...
]
```

The winners have been correctly inserted and their order of insertion preserved by the `OrderedDict`.

Dataset is great for basic SQL-based work, particularly retrieving data you might wish to process or visualize. For more advanced manipulation, it allows you to drop down into SQLAlchemy's core API using the `query` method.

Now that we've covered the basics of working with SQL databases, let's see how Python makes working with the most popular NoSQL database just as painless.

MongoDB

Document-centric datastores like MongoDB offer a lot of convenience to data wranglers. As with all tools, there are good and bad use cases for NoSQL databases. If you have data that has already been refined and processed and don't anticipate needing SQL's powerful query language based on optimized table joins, MongoDB will probably prove easier to work with initially. MongoDB is a particularly good fit for web dataviz because it uses binary JSON (BSON) as its data format. An extension of JSON, BSON can deal with binary data and `datetime` objects, and plays very nicely with JavaScript.

Let's remind ourselves of the target dataset we're aiming to write and read:

```
nobel_winners = [
 {'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'sex': 'male',
  'year': 1921},
  ...
]
```

Creating a MongoDB collection with Python is the work of a few lines:

```
from pymongo import MongoClient

client = MongoClient() ❶
db = client.nobel_prize ❷
coll = db.winners ❸
```

❶ Creates a Mongo client, using the default host and ports.

❷ Creates or accesses the `nobel_prize` database.

❸ If a *winners* collection exists, this will retrieve it; otherwise (as in our case), it creates it.

USING CONSTANTS FOR MONGODB ACCESS

Accessing and creating a MongoDB database with Python involves the same operation, using dot notation and square-bracket key access:

```
db = client.nobel_prize
db = client['nobel_prize']
```

This is all very convenient, but it means a single spelling mistake, such as `noble_prize`, could both create an unwanted database and cause future operations to fail to update the correct one. For this reason, I advise using constant strings to access your MongoDB databases and collections:

```
DB_NOBEL_PRIZE = 'nobel_prize'
COLL_WINNERS = 'winners'

db = client[DB_NOBEL_PRIZE]
coll = db[COLL_WINNERS]
```

MongoDB databases run on localhost port 27017 by default but could be anywhere on the Web. They also take an optional username and password. Example 2-5 shows how to create a simple utility function to access our database, with standard defaults.

Example 2-5. Accessing a MongoDB database

```
from pymongo import MongoClient

def get_mongo_database(db_name, host='localhost',\
```

```
                    port=27017, username=None, password=None):
    """ Get named database from MongoDB with/out authentication """
    # make Mongo connection with/out authentication
    if username and password:
        mongo_uri = 'mongodb://%s:%s@%s/%s'%\ ❶
        (username, password, host, db_name)
        conn = MongoClient(mongo_uri)
    else:
        conn = MongoClient(host, port)

    return conn[db_name]
```

❶ We specify the database name in the MongoDB URI (Uniform Resource Identifier) as the user may not have general privileges for the database.

We can now create a Nobel Prize database and add our target dataset (Example 2-1). Let's first get a winners collection, using the string constants for access:

```
db = get_mongo_database(DB_NOBEL_PRIZE)
coll = db[COLL_WINNERS]
```

Inserting our Nobel Prize dataset is then as easy as can be:

```
coll.insert_many(nobel_winners)
coll.find()
Out:
[{'_id': ObjectId('61940b7dc454e79ffb14cd25'),
  'category': 'Physics',
  'name': 'Albert Einstein',
  'nationality': 'Swiss',
  'year': 1921,
  'sex': 'male'},
 {'_id': ObjectId('61940b7dc454e79ffb14cd26'),
 ...
```

The resulting array of `ObjectIds` can be used for future retrieval, but MongoDB has already left its stamp on our `nobel_winners` list, adding a hidden `id` property.[8]

TIP

MongoDB's `ObjectId`s have quite a bit of hidden functionality, being a lot more than a simple random identifier. You can, for example, get the generation time of the `ObjectId`, which gives you access to a handy timestamp:

```
import bson
oid = bson.ObjectId()
oid.generation_time
Out: datetime.datetime(2015, 11, 4, 15, 43, 23...
```

Find the full details in the MongoDB BSON documentation.

Now that we've got some items in our winners collection, MongoDB makes finding them very easy, with its `find` method taking a dictionary query:

```
res = coll.find({'category':'Chemistry'})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d6'),
  u'category': u'Chemistry',
  u'name': u'Marie Curie',
  u'nationality': u'Polish',
  u'sex': u'female',
  u'year': 1911}]
```

There are a number of special dollar-prefixed operators that allow for sophisticated querying. Let's find all the winners after 1930 using the `$gt` (greater-than) operator:

```
res = coll.find({'year': {'$gt': 1930}})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d5'),
  u'category': u'Physics',
  u'name': u'Paul Dirac',
  u'nationality': u'British',
  u'sex': u'male',
  u'year': 1933}]
```

You can also use Boolean expression, for instance, to find all winners after 1930 or all female winners:

```
res = coll.find({'$or':[{'year': {'$gt': 1930}},\
{'sex':'female'}]})
list(res)
Out:
[{u'_id': ObjectId('55f8326f26a7112e547879d5'),
  u'category': u'Physics',
  u'name': u'Paul Dirac',
  u'nationality': u'British',
  u'sex': u'male',
  u'year': 1933},
 {u'_id': ObjectId('55f8326f26a7112e547879d6'),
  u'category': u'Chemistry',
  u'name': u'Marie Curie',
  u'nationality': u'Polish',
  u'sex': u'female',
  u'year': 1911}]
```

You can find the full list of available query expressions in the MongoDB documentation.

As a final test, let's turn our new *winners* collection back into a Python list of dictionaries. We'll create a utility function for the task:

```
def mongo_coll_to_dicts(dbname='test', collname='test',\
                        query={}, del_id=True, **kw): ❶

    db = get_mongo_database(dbname, **kw)
    res = list(db[collname].find(query))

    if del_id:
        for r in res:
            r.pop('_id')

    return res
```

❶ An empty `query dict {}` will find all documents in the collection. `del_id` is a flag to remove MongoDB's `ObjectIds` from the items by default.

We can now create our target dataset:

```
mongo_coll_to_dicts(DB_NOBEL_PRIZE, COLL_WINNERS)
Out:
[{u'category': u'Physics',
  u'name': u'Albert Einstein',
  u'nationality': u'Swiss',
  u'sex': u'male',
  u'year': 1921},
  ...
]
```

MongoDB's schema-less databases are great for fast prototyping in solo work or small teams. There will probably come a point, particularly with large code bases, when a formal schema becomes a useful reference and sanity check; and when you are choosing a data model, the ease with which document forms can be adapted is a bonus. Being able to pass Python dictionaries as queries to PyMongo and having access to client-side generated `ObjectId`s are a couple of other conveniences.

We've now passed the `nobel_winners` data in Example 2-1 through all our required file formats and databases. Let's consider the special case of dealing with dates and times before summing up.

Dealing with Dates, Times, and Complex Data

The ability to deal comfortably with dates and times is fundamental to dataviz work but can be quite tricky. There are many ways to represent a date or datetime as a string, each one requiring a separate encoding or decoding. For this reason it's good to settle on one format in your own work and encourage others to do the same. I recommend using the International Standard Organization (ISO) 8601 time format as your string representation for dates and times, and using the Coordinated Universal Time (UTC) form.[9] Here's a few examples of ISO 8601 date and datetime strings:

2021-09-23	A date (Python/C format code `'%Y-%m-%d'`)
2021-09-23T16:32:35Z	A UTC (*Z* after time) date and time (`'T%H:%M:%S'`)
2021-09-23T16:32+02:00	A positive two-hour (+02:00) offset from UTC (e.g., Central European Time)

Note the importance of being prepared to deal with different time zones. These are not always on lines of longitude (see Wikipedia's Time Zone entry), and often the best way to derive an accurate time is by using UTC time plus a geographic location.

ISO 8601 is the standard used by JavaScript and is easy to work with in Python. As web data visualizers, our key concern is in creating a string representation that can be passed between Python and JavaScript using JSON and encoded and decoded easily at both ends.

Let's take a date and time in the shape of a Python `datetime`, convert it into a string, and then see how that string can be consumed by JavaScript.

First we produce our Python `datetime`:

```
from datetime import datetime

d = datetime.now()
d.isoformat()
Out:
'2021-11-16T22:55:48.738105'
```

This string can then be saved to JSON or CSV, read by JavaScript, and used to create a `Date` object:

```
d = new Date('2021-11-16T22:55:48.738105')
> Tue Nov 16 2021 22:55:48 GMT+0000 (Greenwich Mean Time)
```

We can return the datetime to ISO 8601 string form with the `toISOString` method:

```
d.toISOString()
> '2021-11-16T22:55:48.738Z'
```

Finally, we can read the string back into Python.

If you know that you're dealing with an ISO-format time string, Python's `dateutil` module should do the job.[10] But you'll probably want to sanity-check the result:

```
from dateutil import parser

d = parser.parse('2021-11-16T22:55:48.738Z')
d
Out:
datetime.datetime(2021, 11, 16, 22, 55, 48, 738000,\
tzinfo=tzutc())
```

Note that we've lost some resolution in the trip from Python to JavaScript and back again, the latter dealing in milliseconds, not microseconds. This is unlikely to be an issue in any dataviz work but is good to bear in mind just in case some strange temporal errors occur.

Summary

This chapter aimed to make you comfortable using Python to move data around the various file formats and databases that a data visualizer might expect to bump into. Using databases effectively and efficiently is a skill that takes a while to learn, but you should now be comfortable with basic reading and writing for the large majority of dataviz use cases.

Now that we have the vital lubrication for our dataviz toolchain, let's get up to scratch on the basic web development skills you'll need for the chapters ahead.

1 I recommend using JSON over CSV as your preferred data format.

2 The Python module `dateutil` has a parser that will parse most dates and times sensibly, and might be a good basis for this.

3 On a cautionary note, it is probably a bad idea to use different database configurations for testing and production.

4 See details on SQLAlchemy of this *lazy initialization*.

5 This assumes the database doesn't already exist. If it does, `Base` will be used to create new insertions and to interpret retrievals.

6 Dataset's official motto is "databases for lazy people." It is not part of the standard Anaconda package, so you'll want to install it using pip from the command line: `$ pip install dataset`.

7 See this documentation for further details of how to use transactions to group updates.

8 One of the cool things about MongoDB is that the `ObjectId`s are generated on the client side, removing the need to quiz the database for them.

9 To get the actual local time from UTC, you can store a time zone offset or, better still, derive it from a geocoordinate; this is because time zones do not follow lines of longitude very exactly.

10 To install, just run `pip install python-dateutil`. `dateutil` is a pretty powerful extension of Python's `datetime`; check it out on Read the Docs.

Chapter 3. Visualizing Data with Matplotlib

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

As a data visualizer, one of the best ways to come to grips with your data is to visualize it interactively, using the full range of charts and plots that have evolved to summarize and refine datasets. Conventionally, the fruits of this exploratory phase are then presented as static figures, but increasingly they are used to construct more engaging interactive web-based charts, such as the cool D3 visualizations you have probably seen (one of which we'll be building in [Link to Come]).

Python's Matplotlib and its family of extensions (such as the statistically focused Seaborn) form a mature and very customizable plotting ecosystem. Matplotlib plots can be used interactively by IPython (the Qt and Notebook versions), providing a very powerful and intuitive way of finding interesting nuggets in your data. In this chapter we'll introduce Matplotlib and one of its great extensions, Seaborn.

Pyplot and Object-Oriented Matplotlib

Matplotlib can be more than a little confusing, especially if you start randomly sampling examples online. The main complicating factor is that there are two main ways to create plots, which are similar enough to be confused but different enough to lead to a lot of frustrating errors. The first way uses a global state machine to interact directly with Matplotlib's `pyplot` module. The second, object-oriented approach uses the more familiar notion of figure and axes classes to provide a programmatic alternative. I'll clarify their differences in the sections ahead, but as a rough rule of thumb, if you're working interactively with single plots, `pyplot`'s global state is a convenient shortcut. For all other occasions, it makes sense to explicitly declare your figures and axes using the object-oriented approach.

Starting an Interactive Session

We will be using a Jupyter notebook for our interactive visualization. Use the following command to start a session:

```
$ jupyter notebook
```

You can then use one of the Matplotlib magic commands within the IPython session to enable interactive Matplotlib. On its own, `%matplotlib` will use the default GUI backend to create a plotting window, but you can specify the backend directly. The following should work on standard and Qt console IPython:[1]

```
%matplotlib [qt | osx | wx ...]
```

To get inline graphics in the Notebook or Qt console, you can use the `inline` directive. Note that with inline plots, you can't amend them after creation, unlike the standalone Matplotlib window:

```
%matplotlib inline
```

Whether you are using Matplotlib interactively or in Python programs, you'll use similar imports:

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

> **NOTE**
>
> You will find many examples of Matplotlib using `pylab`. Pylab is a convenience module that bulk-imports `matplotlib.pyplot` (for plotting) and NumPy in a single namespace. Pylab is pretty much deprecated now, but even were it not, I'd still recommend avoiding this namespace and merging and importing `pyplot` and `numpy` explicitly.

While NumPy and Pandas are not mandatory, Matplotlib is designed to play well with them, handling NumPy `arrays` and, by association, Pandas `Series`.

The ability to create inline plots is key to enjoyable interaction with Matplotlib, and we achieve this in IPython with the following "magic"[2] injunction:

```
In [0]: %matplotlib inline
```

Your Matplotlib plots will now be inserted into your IPython workflow. This works with Qt and Notebook versions. In the Notebooks, the plots are incorporated into the active cell.

> **AMENDING PLOTS**
>
> In inline mode, after an IPython cell or (multi-line) input has been run, the drawing context is flushed. This means you cannot change the plot from a previous cell or input using the `gcf` (get current figure) method but have to repeat all the plot commands with any additions or amendments in a new input/cell.

Interactive Plotting with Pyplot's Global State

The `pyplot` module provides a global state that you can manipulate interactively.[3] This is intended for use in interactive data exploration and is best when you are creating simple plots, usually containing single figures. `pyplot` is convenient and many of the examples you'll see use it, but for more complex plotting Matplotlib's object-oriented API (which we'll see shortly) comes into its own. Before demoing use of the global plot, let's create some random data to display, courtesy of Panda's useful `period_range` method:

```
from datetime import datetime

x = pd.period_range(datetime.now(), periods=200, freq='d') ❶
x = x.to_timestamp().to_pydatetime() ❷
y = np.random.randn(200, 3).cumsum(0) ❸
```

❶ Creates a Pandas `datetime` index with 200 day (`d`) elements, starting from the current time (`datetime.now()`).

❷ Converts `datetime` index to Python `datetimes`.

❸ Creates three 200-element random arrays summed along the 0 axis.

We now have a y-axis with 200 time slots and three random arrays for the complementary x values. These are provided as separate arguments to the `(line)plot` method:

```
plt.plot(x, y)
```

This gives us the not particularly inspiring chart shown in Figure 3-1. Note how Matplotlib deals naturally with a multidimensional NumPy line array.

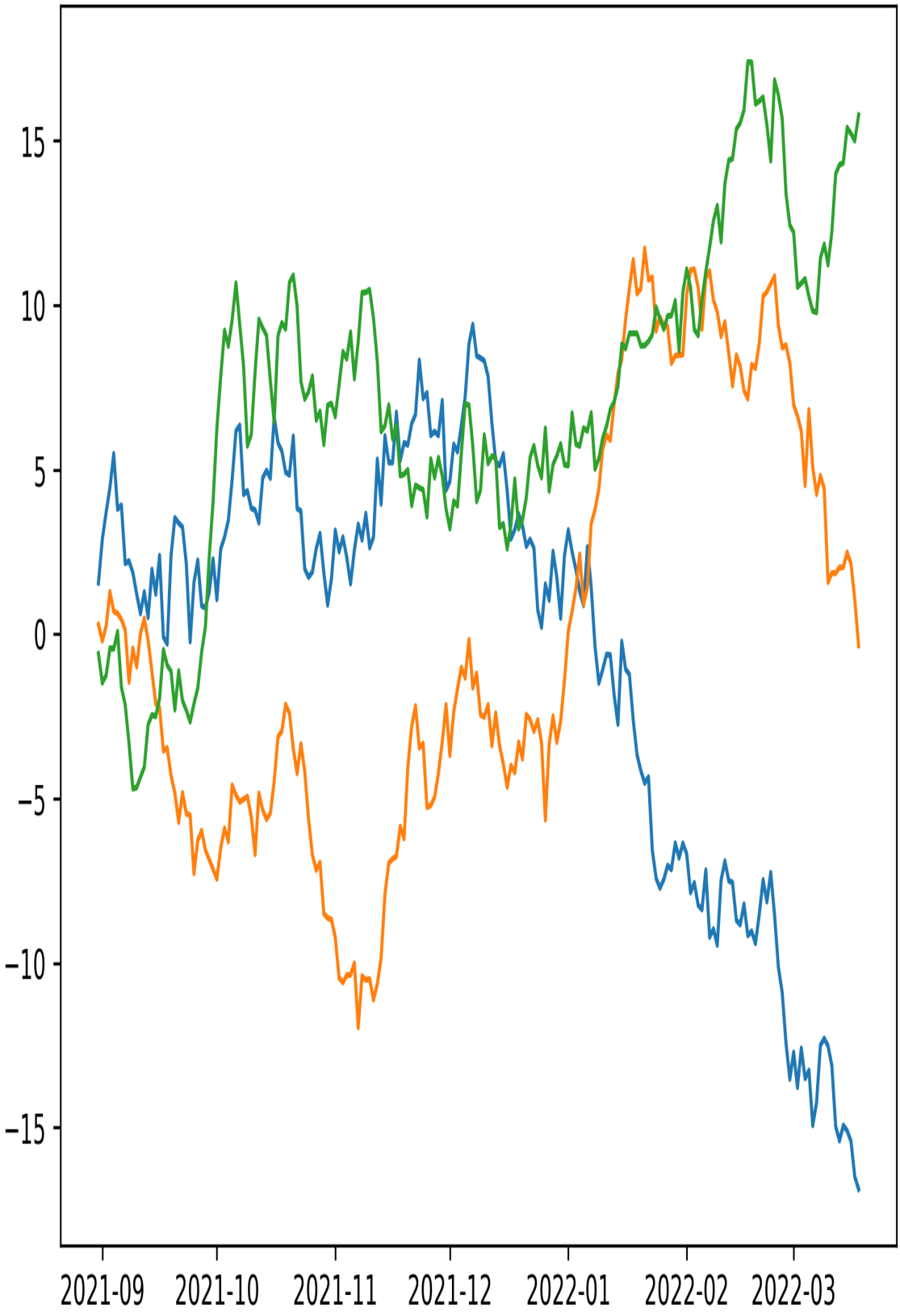

15
10
5
0
−5
−10
−15
2021-09
2021-10
2021-11
2021-12
2022-01
2022-02
2022-03

Figure 3-1. Default line plot

Although Matplotlib's defaults are, by general consensus, less than ideal, one of its strengths is the sheer amount of customization you can perform. This is why there is a rich ecosystem of chart libraries that wrap Matplotlib with better defaults, more attractive color schemes, and more. Let's see some of this customization in action by using vanilla Matplotlib to tailor our default plot.

Configuring Matplotlib

Matplotlib provides a wide range of configurations, which can be specified in a `matplotlibrc` file or dynamically, through the dictionary-like `rcParams` variable. Here we change the width and default color of our plot lines:

```
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r' # red
```

You can find a sample `matplotlibrc` file at the main site.

As well as using the `rcParams` variable, you can use the `gcf` (get current figure) method to grab the currently active figure and manipulate it directly.

Let's see a little example of configuration, setting the current figure's size.

Setting the Figure's Size

If your plot's default readability is poor or the width-to-height ratio suboptimal, you will want to change its size. By default, Matplotlib uses inches for its plotting size. This makes sense when you consider the many backends (often vector-graphic-based) that Matplotlib can save to. Here we use `pyplot` to set the figure size to eight by four inches, using `rcParams` and `gcf`:

```
# set figure size to 8 by 4 inches
plt.rcParams['figure.figsize'] = (8,4)
plt.gcf().set_size_inches(8, 4)
```

Points, Not Pixels

Matplotlib uses points, not pixels, to measure the size of its figures. This is the accepted measure for print-quality publications, and Matplotlib is used to deliver publication-quality images.

By default a point is approximately 1/72 of an inch wide, but Matplotlib allows you to adjust this by changing the dots-per-inch (dpi) for any figures generated. The higher this number, the better the quality of the image. For the purpose of the inline figures shown interactively during IPython sessions, the resolution is usually a product of the backend engine being used to generate the plots (e.g., Qt, WXAgg, tkinter). See here for an explanation of backends.

Labels and Legends

Figure 3-1 needs, among other things, to tell us what the lines mean. Matplotlib has a handy legend box for line labeling, which, like most things Matplotlib, is heavily configurable. Labeling our three lines involves a little indirection as the `plot` method only takes one label, which it applies to all lines generated. Usefully, the plot command returns all `Line2D` objects created. These can be used by the `legend` method to set individual labels.

```
plots = plt.plot(x,y)
plots
Out:
[<matplotlib.lines.Line2D at 0x9b31a90>,
 <matplotlib.lines.Line2D at 0x9b4da90>,
 <matplotlib.lines.Line2D at 0x9b4dcd0>]
```

The `legend` method can set labels, suggest a location for the legend box, and configure a number of other things:

```
plt.legend(plots, ('foo', 'bar', 'baz'), ❶
           loc='best', ❷
           framealpha=0.5, ❸
           prop={'size':'small', 'family':'monospace'}) ❹
```

❶ Sets the labels for our three plots.

❷ Using the `best` location should avoid obscuring lines.

❸ Sets the legend's transparency.

❹ Here we adjust the font properties of the legend.[4]

Titles and Axes Labels

Adding a title and label for your axes is as easy as can be:

```
plt.title('Random trends')
plt.xlabel('Date')
plt.ylabel('Cum. sum')
```

You can add some text with the `figtext` method:[5]

```
plt.figtext(0.995, 0.01, ❶
            u'© Acme designs 2021',
            ha='right', va='bottom') ❷
```

❶ The location of the text proportionate to figure size.

❷ Horizontal (`ha`) and vertical (`va`) alignment.

The complete code is shown in Example 3-1 and the resulting chart in Figure 3-2.

Example 3-1. Customized line chart

```
plots = plt.plot(x, y)
plt.legend(plots, ('foo', 'bar', 'baz'), loc='best,
                  framealpha=0.25,
```

```
                    prop={'size':'small', 'family':'monospace'})
plt.gcf().set_size_inches(8, 4)
plt.title('Random trends')
plt.xlabel('Date')
plt.ylabel('Cum. sum')
plt.grid(True) ❶
plt.figtext(0.995, 0.01, u'\u00a9 Acme Designs 2021',
ha='right', va='bottom')
plt.tight_layout() ❷
```

❶ This will add a dotted grid to the figure, marking the axis ticks.

❷ The `tight_layout` method should guarantee that all your plot elements are within the figure box. Otherwise, you might find tick-labels or legends truncated.

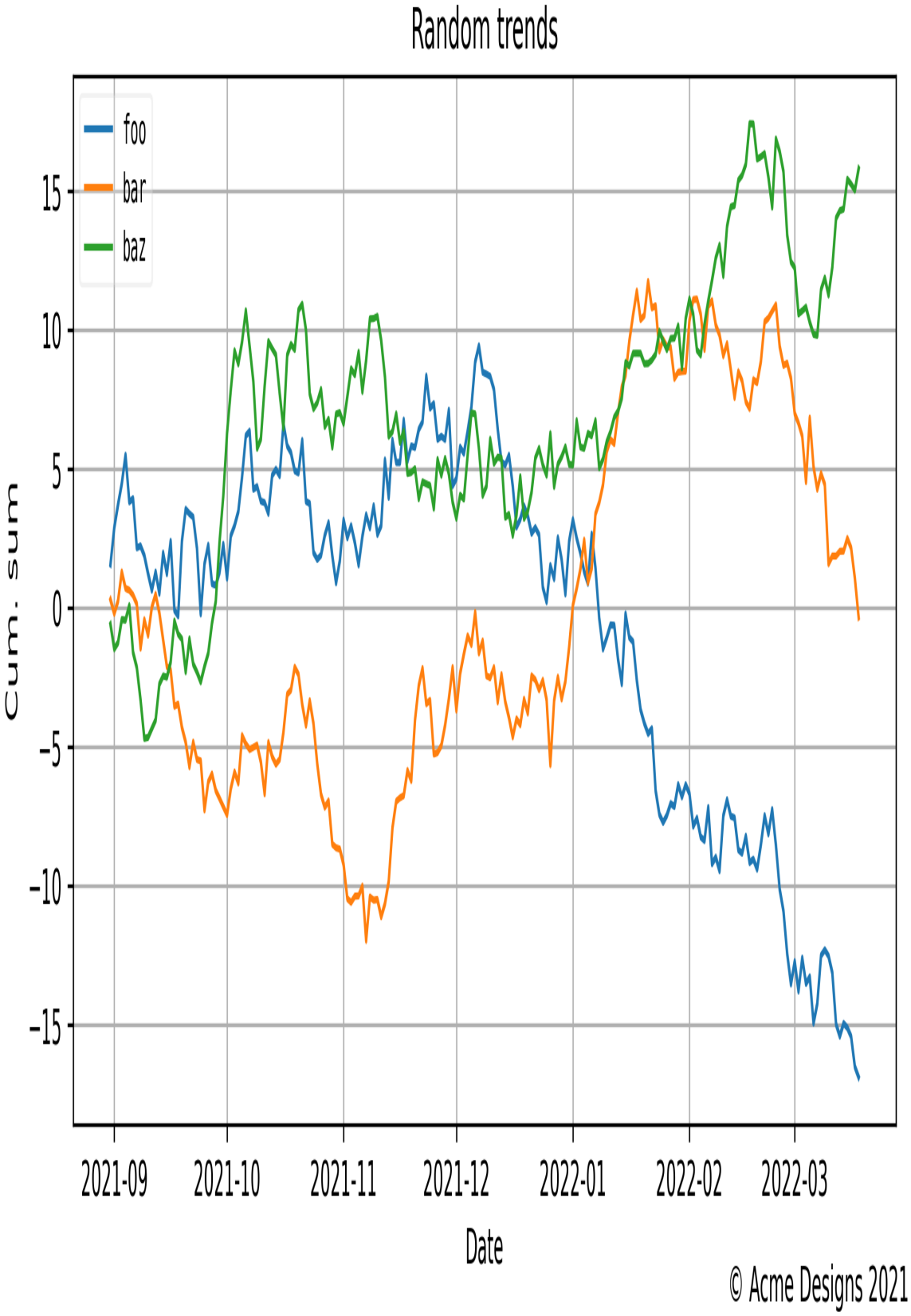

Random trends
foo
bar
baz
15
10
5
0
−5
−10
−15
Cum. sum
2021-09
2021-10
2021-11
2021-12
2022-01
2022-02
2022-03
Date
© Acme Designs 2021

Figure 3-2. Customized line chart

We used the `tight_layout` method in Example 3-1 to prevent plot elements from being obscured or truncated. `tight_layout` has been known to cause problems with some systems, particularly OS X. If you have any problems, this issue thread may help. As of now, the best advice is to use the `set_tight_layout` method on the current figure:

```
plt.gcf().set_tight_layout(True)
```

Saving Your Charts

One area where Matplotlib shines is in saving your plots, providing many output formats.[6] The available formats depend on the backends available, but generally PNG, PDF, PS, EPS, and SVG are supported.

Saving is as simple as this:

```
plt.tight_layout() # force plot into figure dimensions
plt.savefig('mpl_3lines_custom.svg')
```

You can set the format explicitly using `format="svg"`, but Matplotlib understands the *.svg* suffix. To avoid truncated labels, use the `tight_layout` method.[7]

Figures and Object-Oriented Matplotlib

As just shown, interactively manipulating Pyplot's global state works fine for quick data sketching and single-plot work. However, if you want to have more control over your charts, Matplotlib's `figure` and `axes` OOP approach is the way to go. Most of the more advanced plotting demos you see will be done this way.

In essence, with OOP Matplotlib we are dealing with a `figure`, which you can think of as a drawing area with one or more axes (or plots) embedded in it. Both `figures` and `axes` have properties that can be independently

specified. In this sense, the interactive `pyplot` route discussed earlier was plotting to a single axis of a global figure.

We can create a figure by using Pyplot's `figure` method:

```
fig = plt.figure(
          figsize=(8, 4), # figure size in inches
          dpi=200, # dots per inch
          tight_layout=True, # fit axes, labels, etc. to canvas
          linewidth=1, edgecolor='r' # 1 pixel wide, red border
          )
```

As you can see, figures share a subset of properties with the global `pyplot` module. These can be set on creation of the figure or through similar methods (i.e., `fig.text()` as opposed to `plt.fig_text()`). Each `figure` can have multiple axes, each of which is analogous to the single, global plot state but with the considerable advantage that multiple axes can exist on one figure, each with independent properties.

Axes and Subplots

The `figure.add_axes` method allows precise control over the position of axes within a figure (e.g., enabling you to embed a smaller plot within the main). Positioning of plot elements uses a 0 → 1 coordinate system, where 1 is the width or height of the figure. You can specify the position using a four-element list or tuple to set bottom-left and top-right bounds [bottom(h*0.2), left(w*0.2), top(h*0.8), right(w*0.8)]:

```
fig.add_axes([0.2, 0.2, 0.8, 0.8])
```

Example 3-2 shows the code needed to insert smaller axes into larger ones, using our random test data. The result is shown in Figure 3-3.

Example 3-2. A plot insert with figure.axes

```
fig = plt.figure(figsize=(8,4))
# --- Main Axes
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.set_title('Main Axes with Insert Child Axes')
ax.plot(x, y[:,0]) ❶
```

```
ax.set_xlabel('Date')
ax.set_ylabel('Cum. sum')
# --- Inserted Axes
ax = fig.add_axes([0.15, 0.15, 0.3, 0.3])
ax.plot(x, y[:,1], color='g') # 'g' for green
ax.set_xticks([]); ❷
```

❶ This selects the first column of our random NumPy y-data.

❷ Removes the x ticks and labels from our embedded plot.

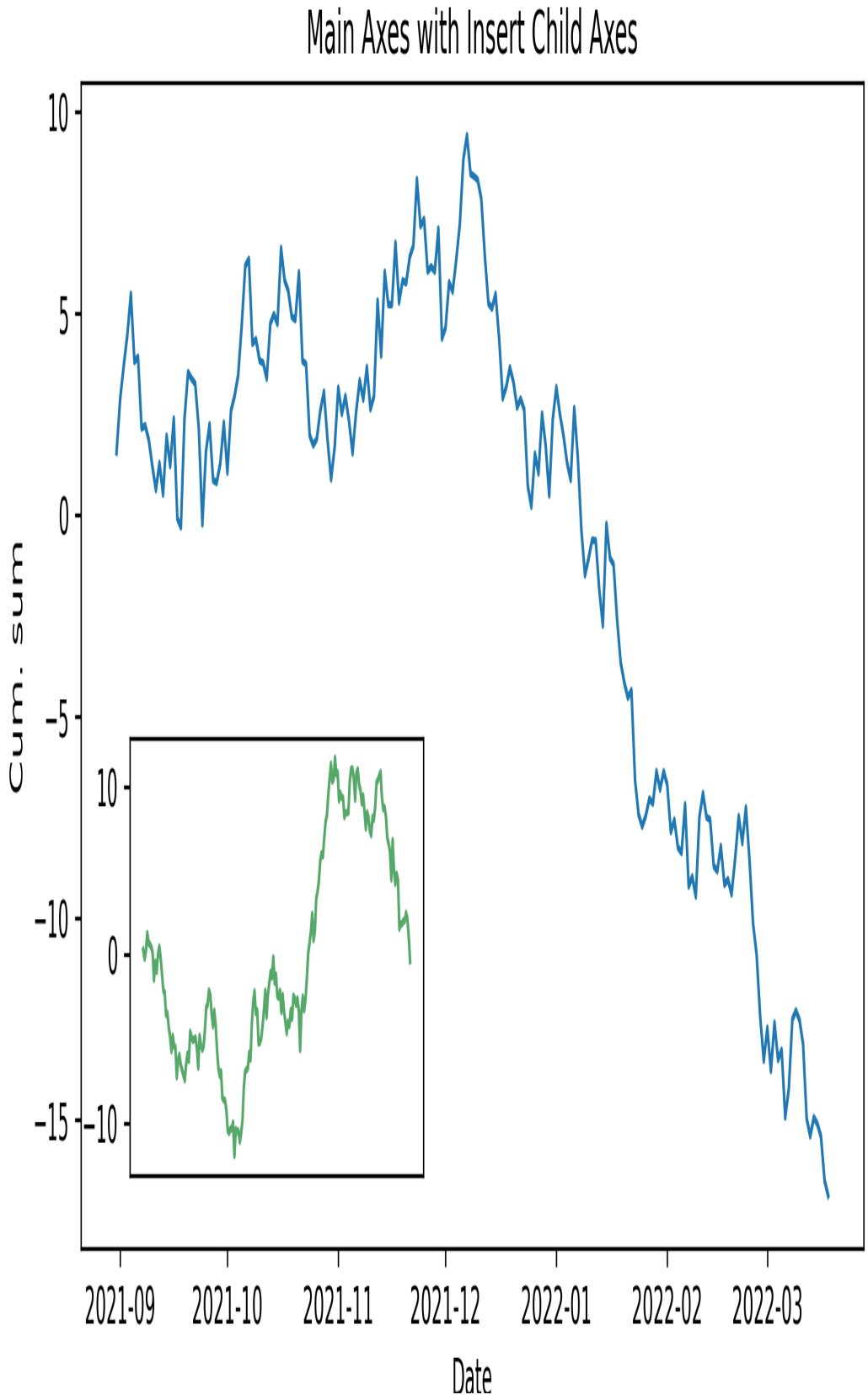

Main Axes with Insert Child Axes
Cum. sum
Date
10
5
0
−5
−10
−15
2021-09
2021-10
2021-11
2021-12
2022-01
2022-02
2022-03
10
0
−10
−10

Figure 3-3. Inserted plot with figure.add_axes

Although `add_axes` gives us a lot of scope for fine-tuning the appearance of our charts, most of the time Matplotlib's built-in grid-layout system makes life much easier.[8] The simplest option is to use `figure.subplots`, which allows you to specify row-column layouts of equal-sized plots. If you want a grid with different-sized plots, the `gridspec` module is your go-to.

Calling `subplots` without arguments returns a figure with single axes. This is closest in use to using the Pyplot state machine. Example 3-3 shows the figure and axes equivalent to the `pyplot` demo in Example 3-1, producing the chart in Figure 3-2. Note the use of "setter" methods for `figure` and `axes`.

Example 3-3. Plotting with single figure and axes

```
figure, ax = plt.subplots()
plots = ax.plot(x, y, label='')
figure.set_size_inches(8, 4)
ax.legend(plots, ('foo', 'bar', 'baz'), loc='best',
framealpha=0.25,
prop={'size':'small', 'family':'monospace'})
ax.set_title('Random trends')
ax.set_xlabel('Date')
ax.set_ylabel('Cum. sum')
ax.grid(True)
figure.text(0.995, 0.01, u'\u00a9 Acme Designs 2021',
ha='right', va='bottom')
figure.tight_layout()
```

Calling `subplots` with arguments for number of rows (`nrows`) and columns (`ncols`) (as shown in Example 3-4) allows multiple plots to be placed on a grid layout (see the results in Figure 3-4). The call to `subplots` returns the figure and an array of axes, in row-column order. In the example, we specify one column so `axes` is a single array of three stacked axes. We make use of Python's handy `zip` method to produce three dictionaries with line data. `zip` takes lists or tuples of length *n* and returns *n* lists, formed by matching the elements by order:

```
letters = ['a', 'b']
numbers = [1, 2]
zip(letters, numbers)
Out:
[('a', 1), ('b', 2)]
```

In the `for` loop, we use `enumerate` to supply an index `i`, which we use to select an axis by row, using our zipped `labelled_data` to provide the plot properties.

Note the shared x- and y-axes specified in the `subplots` call. This allows easy comparison of the three charts, particularly on the now normalized y-axis. To avoid redundant x labels, we only call `set_xlabel` on the last row, using Python's handy negative indexing.

Example 3-4. Using subplots

```
fig, axes = plt.subplots(
                    nrows=3, ncols=1, ❶
                    sharex=True, sharey=True, ❷
                    figsize=(8, 8))
labelled_data = zip(y.transpose(), ❸
                    ('foo', 'bar', 'baz'), ('b', 'g', 'r'))
fig.suptitle('Three Random Trends', fontsize=16)
for i, ld in enumerate(labelled_data):
    ax = axes[i]
    ax.plot(x, ld[0], label=ld[1], color=ld[2])
    ax.set_ylabel('Cum. sum')
    ax.legend(loc='upper left', framealpha=0.5,
              prop={'size':'small'})
axes[-1].set_xlabel('Date') ❹
```

❶ Specifies a subplot grid of three rows by one column.

❷ We want to share x- and y-axes, automatically adjusting limits for easy comparison.

❸ Switch y to row-column and zip the line data, labels, and line colors together.

❹ Labels the last of the shared x-axes.

Now that we've covered the two ways in which IPython and Matplotlib engage interactively, using the global state (accessed through `plt`) and the object-oriented API, let's look at a few of the common plot types you'll use to explore your datasets.

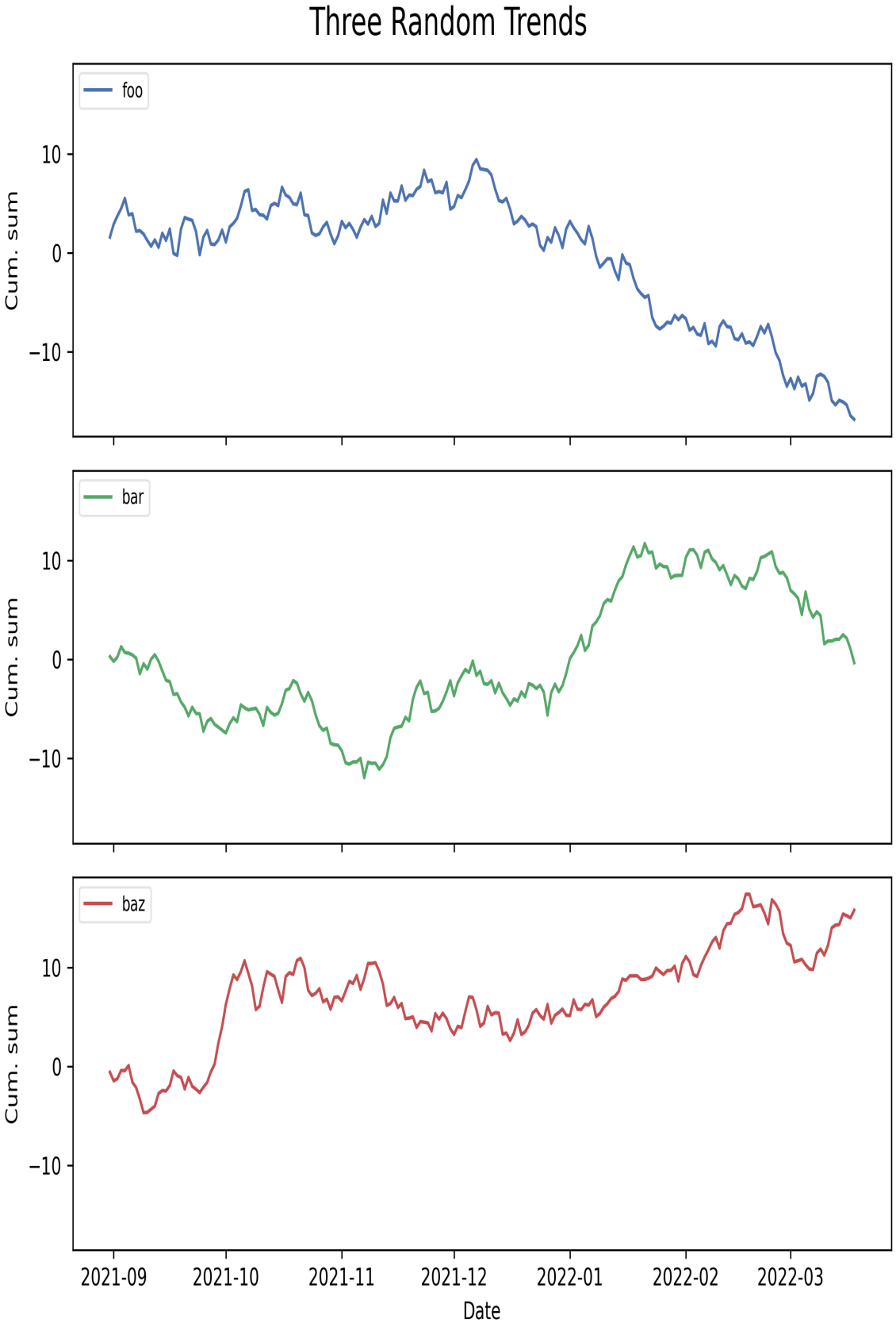

Three Random Trends
foo
bar
baz
Cum. sum
10
0
−10
2021-09
2021-10
2021-11
2021-12
2022-01
2022-02
2022-03
Date

Figure 3-4. Three subplots

Plot Types

As well as the line plot just demonstrated, Matplotlib has a number of plot types available. I'll now demonstrate a few of the ones commonly used in exploratory data visualization.

Bar Charts

The humble bar chart is a staple for a lot of visual data exploration. As with most of Matplotlib charts, there's a good deal of customization possible. We'll now run through a few variants to give you the gist.

The code in Example 3-5 produces the bar chart in Figure 3-5. Note that you have to specify your own bar and label locations. This kind of flexibility is beloved by hardcore Matplotlibbers and is pretty easy to get the hang of. Nevertheless, it's the sort of thing that can get tedious. It's trivial to write some helper methods here, and there are many libraries that wrap Matplotlib and make things a little more user-friendly. As we'll see in Chapter 4, Panda's built-in Matplotlib-based plots are quite a bit simpler to use.

Example 3-5. A simple bar chart

```
labels = ["Physics", "Chemistry", "Literature", "Peace"]
foo_data =   [3, 6, 10, 4]

bar_width = 0.5
xlocations = np.array(range(len(foo_data))) + bar_width ❶
plt.bar(xlocations, foo_data, width=bar_width)
plt.yticks(range(0, 12))
plt.xticks(xlocations, labels) ❷
plt.title("Prizes won by Fooland")
plt.gca().get_xaxis().tick_bottom()
plt.gca().get_yaxis().tick_left()
plt.gcf().set_size_inches((8, 4))
```

❶ Here we create the middle bar locations, two `bar_width`'s apart.

❷ This places tick labels at the middle of the bars.

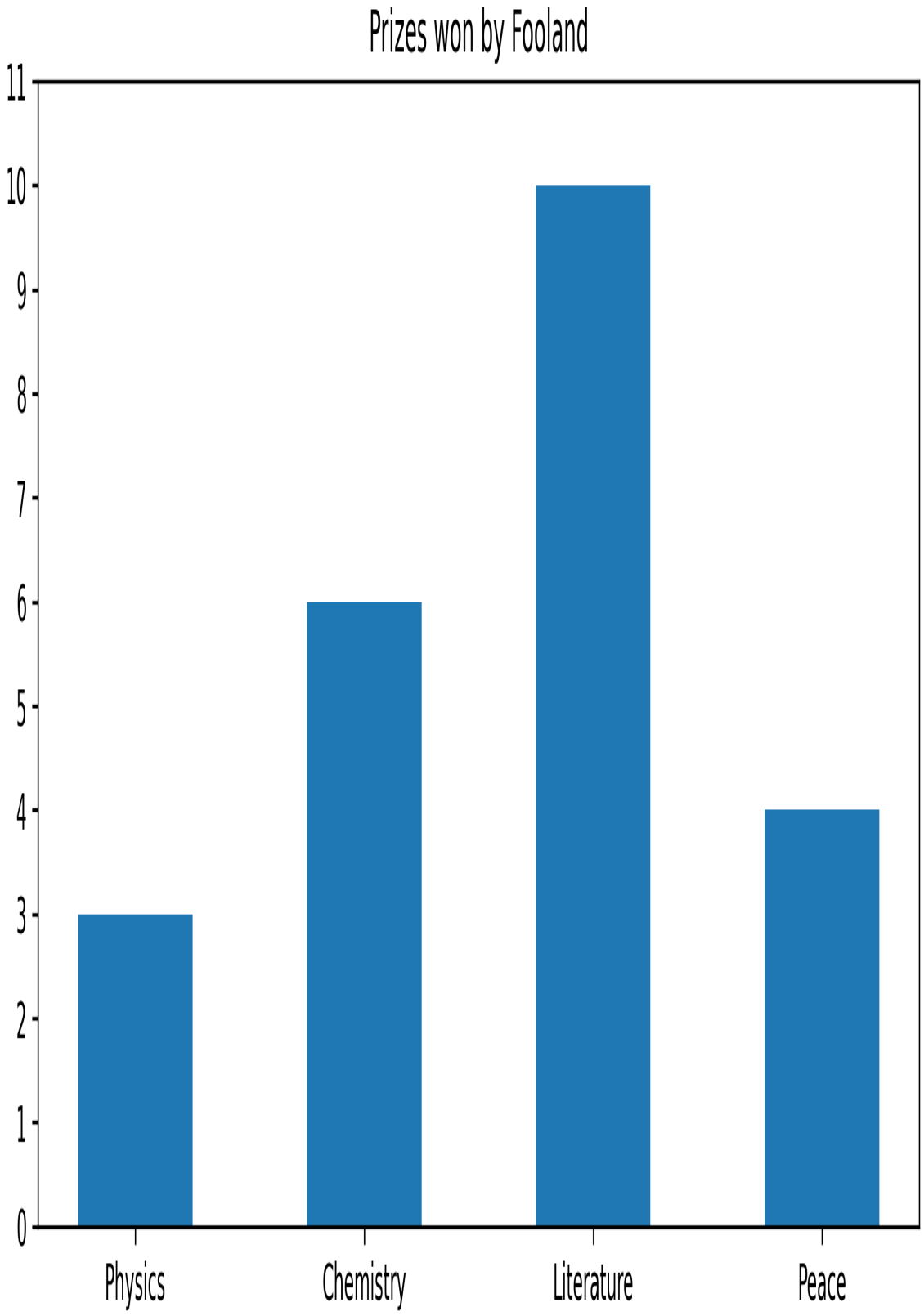

Prizes won by Fooland
11
10
9
8
7
6
5
4
3
2
1
0
Physics
Chemistry
Literature
Peace

Figure 3-5. A simple bar chart

Bar charts with multiple groups are particularly useful. In Example 3-6, we add some more country data (for a mythical Barland) and use the `subplots` method to produce grouped bar charts (see Figure 3-6). Once again we specify the bar locations manually, adding two bar groups—this time with `ax.bar`. Note that our axes' x-limits are automatically rescaled in a sensible fashion, at increments of 0.5:

```
ax.get_xlim()
# Out: (-0.5, 3.5)
```

Use the respective setter methods (`set_xlim`, in this case) if autoscaling doesn't achieve the desired look.

Example 3-6. Creating a grouped bar chart

```
labels = ["Physics", "Chemistry", "Literature", "Peace"]
foo_data = [3, 6, 10, 4]
bar_data = [8, 3, 6, 1]

fig, ax = plt.subplots(figsize=(8, 4))
bar_width = 0.4 ❶
xlocs = np.arange(len(foo_data))
ax.bar(xlocs-bar_width, foo_data, bar_width,
       color='#fde0bc', label='Fooland') ❷
ax.bar(xlocs, bar_data, bar_width, color='peru', label='Barland')
#--- ticks, labels, grids, and title
ax.set_yticks(range(12))
ax.set_xticks(ticks=range(len(foo_data)))
ax.set_xticklabels(labels)
ax.yaxis.grid(True)
ax.legend(loc='best')
ax.set_ylabel('Number of prizes')
fig.suptitle('Prizes by country')
fig.tight_layout(pad=2) ❸
fig.savefig('mpl_barchart_multi.png', dpi=200) ❹
```

❶ With a width of `1` for our two-bar groups, this bar width gives 0.1 bar padding.

❷ Matplotlib supports standard HTML colors, taking hex values or a name.

❸ We use the `pad` argument to specify padding around the figure as a fraction of the font size.

❹ This saves the figure at the high resolution of 200 dots per inch.

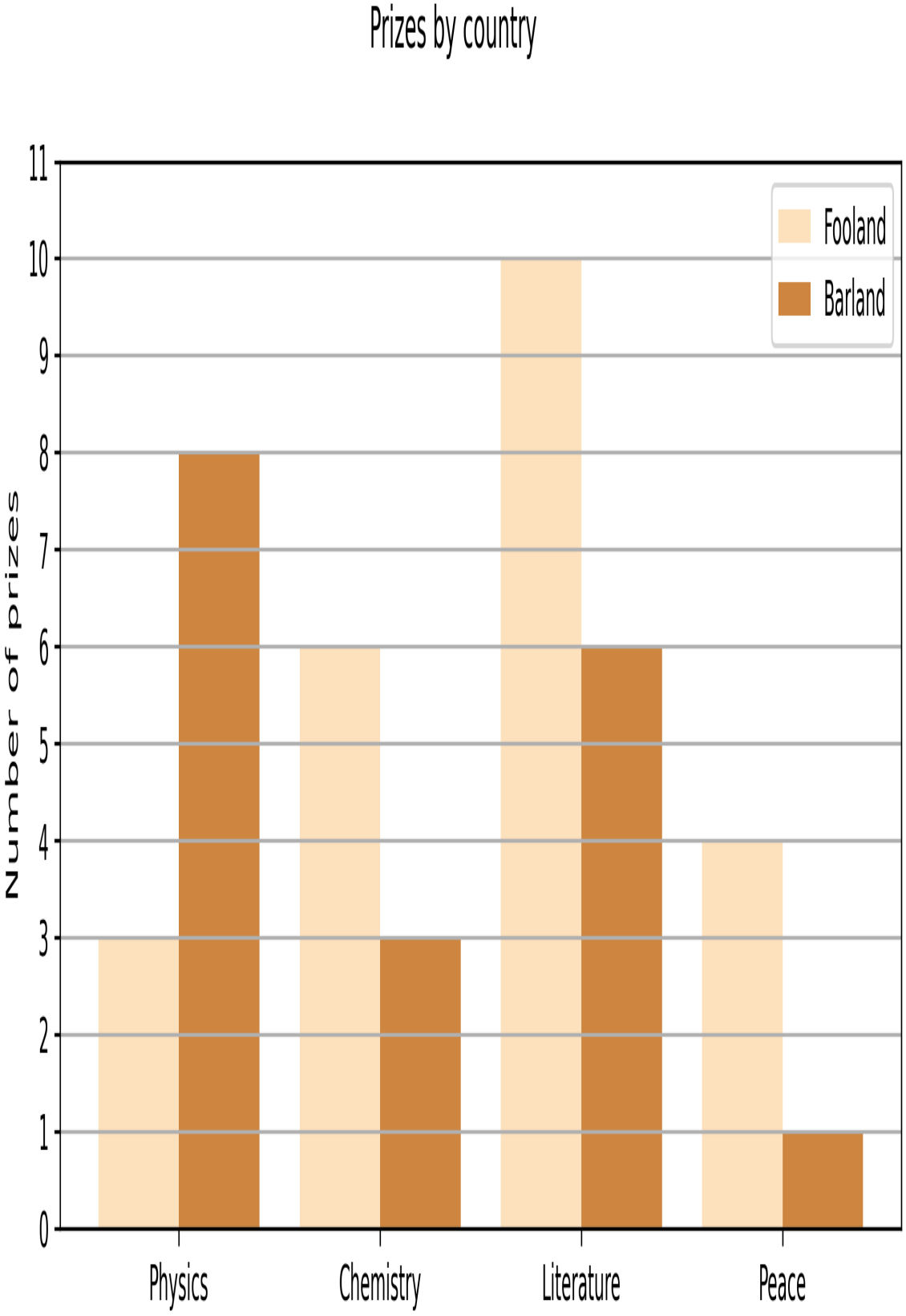

Prizes by country
Number of prizes
0
1
2
3
4
5
6
7
8
9
10
11
Fooland
Barland
Physics
Chemistry
Literature
Peace

Figure 3-6. Grouped bar charts

It's often useful to use horizontal bars, particularly if there are a lot of them and/or you are using tick labels, which are likely to run into one another if placed on the same line. Turning Figure 3-6 on its side is easy enough, requiring only that we replace the `bar` method with its horizontal counterpart `barh` and switch the axis labels and limits (see Figure 3-7).

Example 3-7. Converting Example 3-6 to horizontal bars

```
# ...
ylocs = np.arange(len(foo_data))
ax.barh(ylocs-bar_width, foo_data, bar_width, color='#fde0bc',
        label='Fooland') ❶
ax.barh(ylocs, bar_data, bar_width, color='peru', label='Barland')
# --- labels, grids and title, then save
ax.set_xticks(range(12)) ❷
ax.set_yticks(ticks=ylocs-bar_width/2)
ax.set_yticklabels(labels)
ax.xaxis.grid(True)
ax.legend(loc='best')
ax.set_xlabel('Number of prizes')
# ...
```

❶ To create a horizontal bar chart, we use `barh` in place of `bar`.

❷ A horizontal chart necessitates swapping the horizontal and vertical axes.

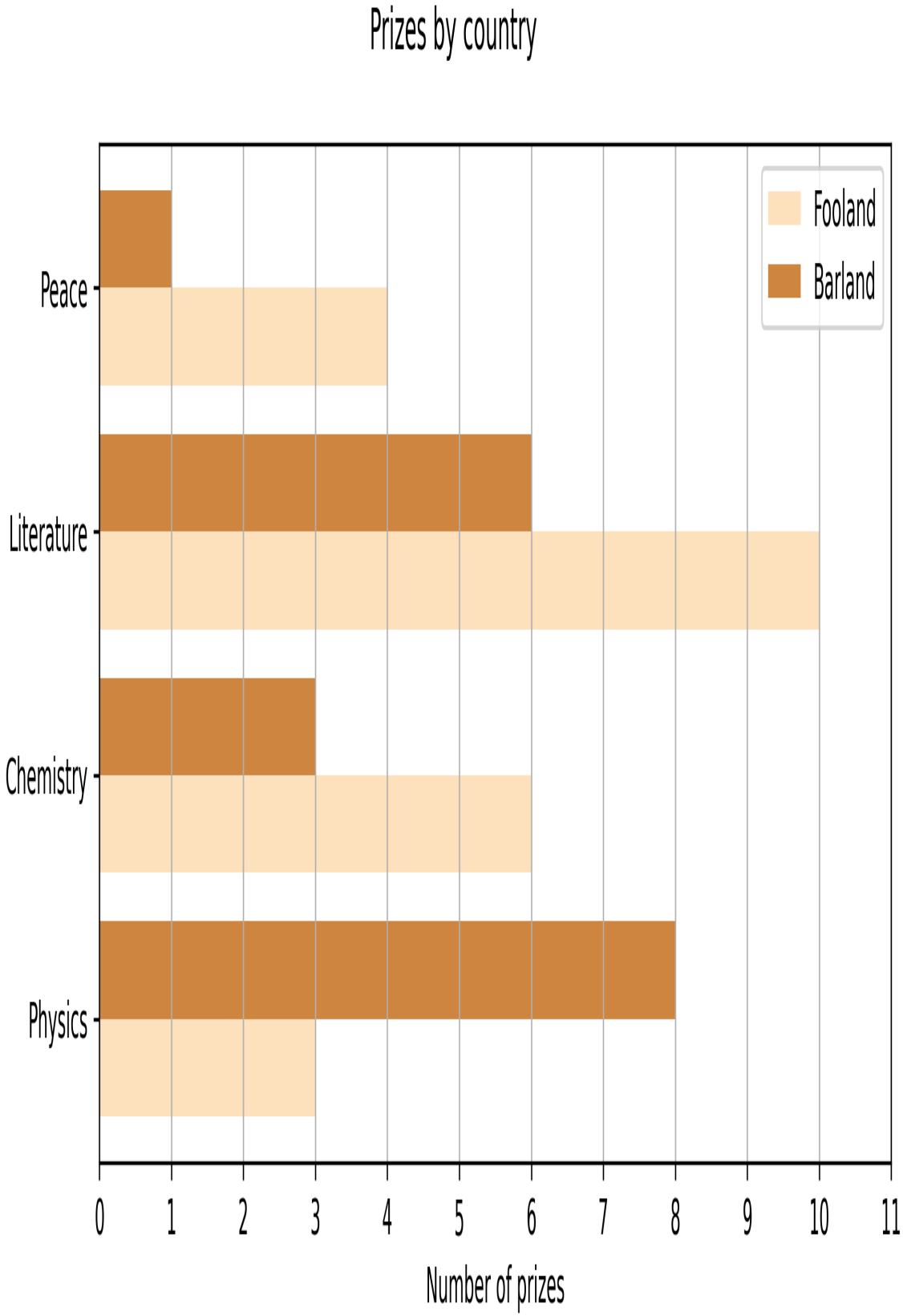

Prizes by country
Fooland
Barland
Peace
Literature
Chemistry
Physics
0
1
2
3
4
5
6
7
8
9
10
11
Number of prizes

Figure 3-7. Turning the bars on their side

Stacked bars are easy to achieve in Matplotlib.[9] Example 3-8 converts Figure 3-6 to a stacked form; Figure 3-8 shows the result. The trick is to use the bottom argument to `bar` to set the bottom of the raised bars as the top of the previous group.

Example 3-8. Converting Example 3-6 to stacked bars

```
# ...
bar_width = 0.8
xlocs = np.arange(len(foo_data))
ax.bar(xlocs, foo_data, bar_width, color='#fde0bc', ❶
       label='Fooland')
ax.bar(xlocs, bar_data, bar_width, color='peru',    ❷
       label='Barland', bottom=foo_data)
# --- labels, grids and title, then save
ax.set_yticks(range(18))
ax.set_xticks(ticks=xlocs)
ax.set_xticklabels(labels)
# ...
```

❶ The `foo_data` and `bar_data` bar groups share the same x-locations.

❷ The bottom of the `bar_data` group is the top of the `foo_data`, providing stacked bars.

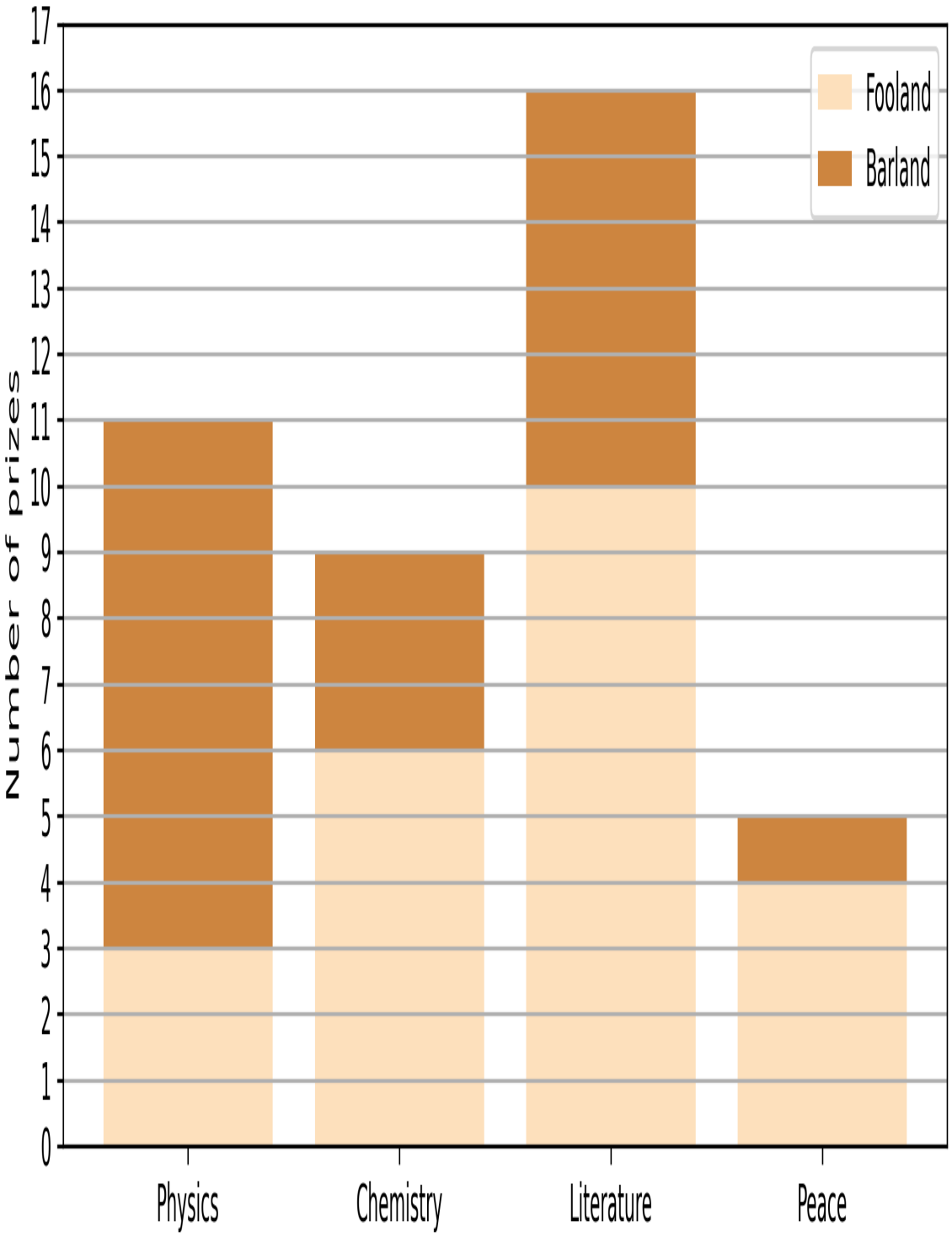

Prizes by country
Number of prizes
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Fooland
Barland
Physics
Chemistry
Literature
Peace

Figure 3-8. Stacked bar chart

Scatter Plots

Another useful chart is the scatter plot, which takes 2D arrays of points with options for point size, color, and more.

Example 3-9 shows the code for a quick scatter plot, using Matplotlib autoscaling for x and y limits. We create a noisy line by adding normally distributed random numbers (sigma of 10). Figure 3-9 shows the resulting chart.

Example 3-9. A simple scatter plot

```
num_points = 100
gradient = 0.5
x = np.array(range(num_points))
y = np.random.randn(num_points) * 10 + x*gradient ❶
fig, ax = plt.subplots(figsize=(8, 4))
ax.scatter(x, y) ❷

fig.suptitle('A Simple Scatterplot')
```

❶ `randn` gives normally distributed random numbers, which we scale to be within 0 and 10 and to which we then add an x-dependent value.

❷ The equally sized x and y arrays provide the point coordinates.

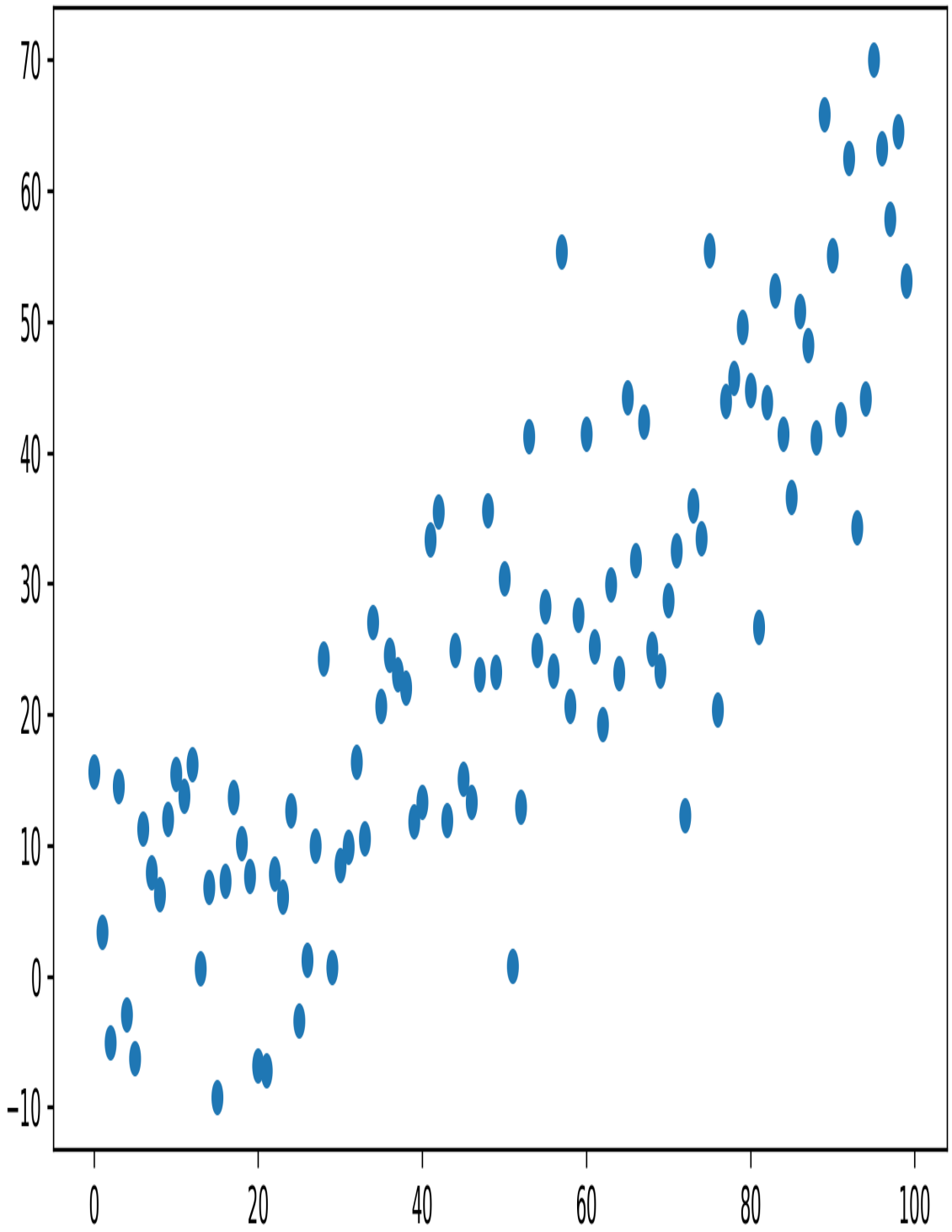

A Simple Scatterplot
70
60
50
40
30
20
10
0
−10
0
20
40
60
80
100

Figure 3-9. A simple scatter plot

We can adjust the size and color of individual points by passing an array of marker sizes and color indices to the current default colormap. One thing to note, which can be confusing, is that we are specifying the area of the markers' bounding boxes, not the circles' diameters. This means if we want points to double the diameter of the circles, we must increase the size by a factor of four.[10] In Example 3-10, we add size and color information to our simple scatter plot, producing Figure 3-10.

Example 3-10. Adjusting point size and color

```
num_points = 100
gradient = 0.5
x = np.array(range(num_points))
y = np.random.randn(num_points) * 10 + x*gradient
fig, ax = plt.subplots(figsize=(8, 4))
colors = np.random.rand(num_points) ❶
size = np.pi * (2 + np.random.rand(num_points) * 8) ** 2 ❷
ax.scatter(x, y, s=size, c=colors, alpha=0.5) ❸
fig.suptitle('Scatterplot with Color and Size Specified')
```

❶ This produces 100 random color values between 0 and 1 for the default colormap.

❷ We use the power notation `**` to square values between 2 and 10, the width range for our markers.

❸ We use the `alpha` argument to make our markers half-transparent.

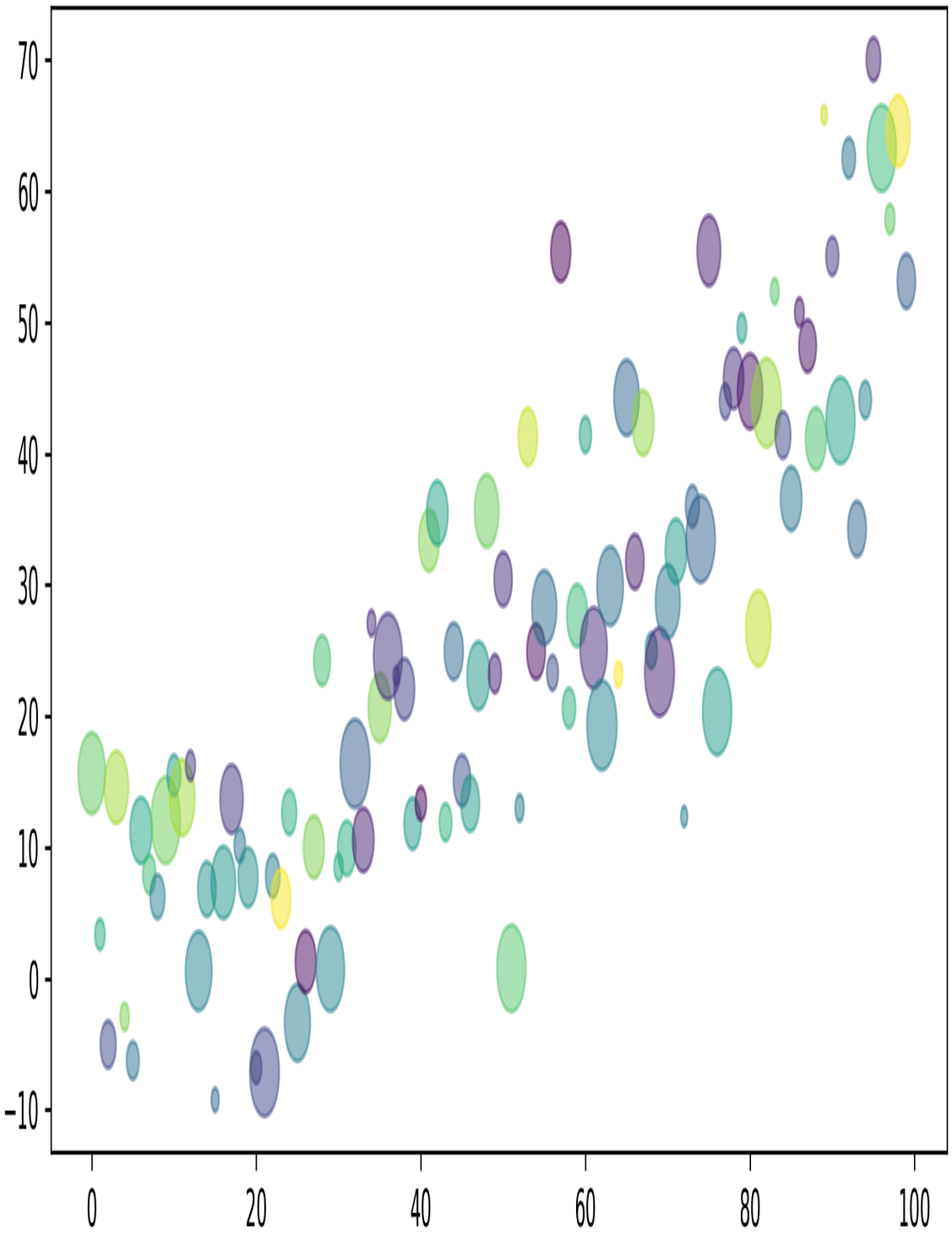

A Simple Scatterplot
70
60
50
40
30
20
10
0
−10
0
20
40
60
80
100

Figure 3-10. Adjusting point size and color

> **MATPLOTLIB COLORMAPS**
>
> Matplotlib has a huge variety of colormaps available, the choice of which can significantly improve the quality of your visualization. See the colormap docs for details.

Adding a regression line

A regression line is a simple predictive model of the correlation between two variables, in this case the x and y coordinates of our scatter plot. The line is essentially a best fit through the points of the plot, and adding one to a scatter plot is a useful dataviz technique and a good way to demo Matplotlib, NumPy interaction.

In Example 3-11 NumPy's very useful `polyfit` function is used to generate the gradient and constant of a best-fit line for the points defined by the x and y arrays. We then plot this line on the same axes as the scatter plot (see Figure 3-11).

Example 3-11. Scatter plot with regression line

```
num_points = 100
gradient = 0.5
x = np.array(range(num_points))
y = np.random.randn(num_points) * 10 + x*gradient
fig, ax = plt.subplots(figsize=(8, 4))
ax.scatter(x, y)
m, c = np.polyfit(x, y ,1) ❶
ax.plot(x, m*x + c) ❷
fig.suptitle('Scatterplot With Regression-line')
```

❶ We use NumPy's `polyfit` in 1D to get a line gradient (`m`) and constant (`c`) for a best-fit line through our random points.

❷ Use the gradient and constant to plot a line on the scatter plot's axes (`y = mx + c`).

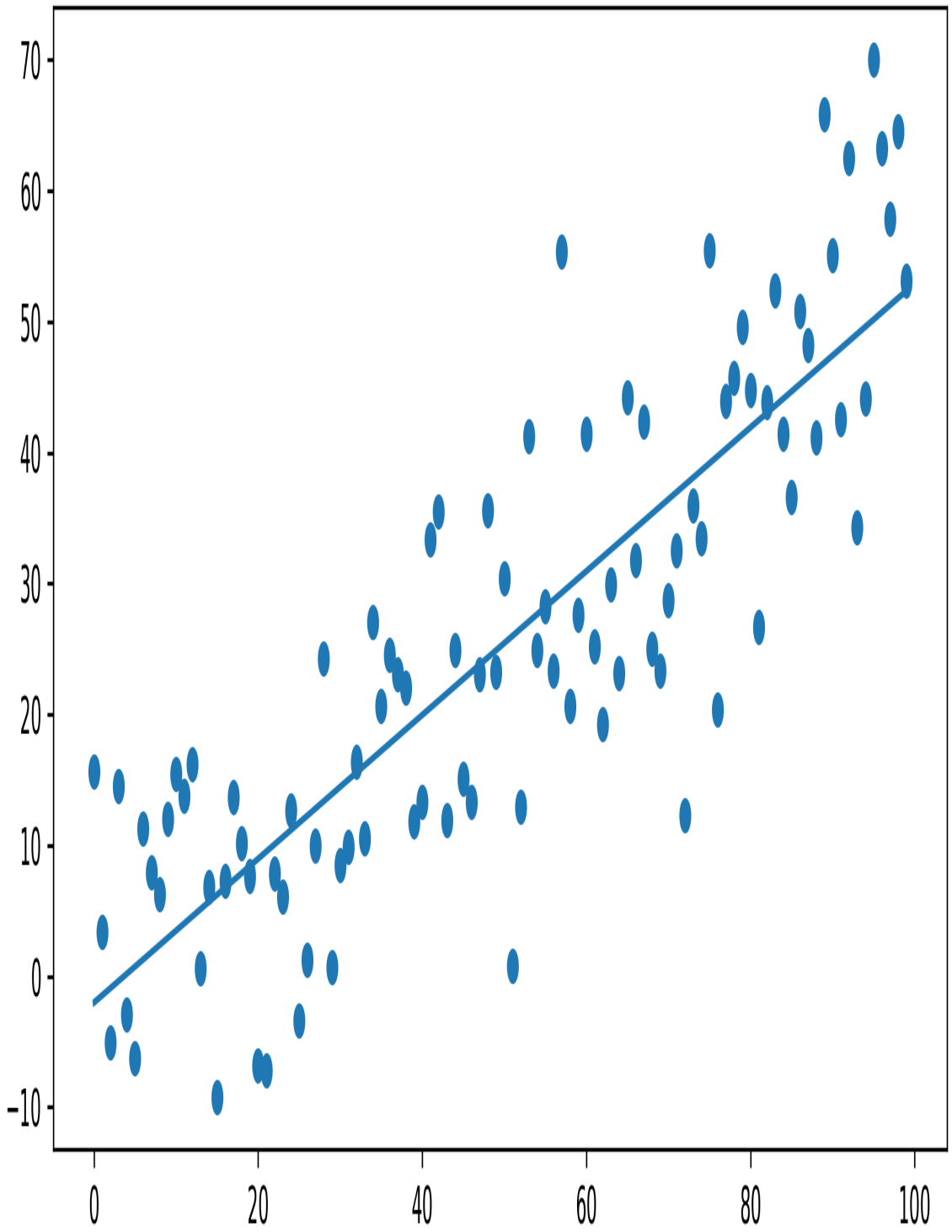

Scatterplot With Regression-line
70
60
50
40
30
20
10
0
−10
0
20
40
60
80
100

Figure 3-11. Scatter plot with regression line

It's generally a good idea to plot confidence intervals when doing line regression. This gives an idea of how reliable the line fit is, based on the number and distribution of the points. Confidence intervals can be achieved with Matplotlib and NumPy, but it is a little awkward. Luckily, there is a library built on Matplotlib that has extra, specialized functions for statistical analysis and data visualization and, in the opinion of many, looks a lot better than Matplotlib's defaults. That library is Seaborn, which we are going to take a quick look at now.

Seaborn

There are a number of libraries that wrap the powerful plotting abilities of Matplotlib in a more user-friendly guise[11] and, as important for us data visualizers, play nicely with Pandas:

- *Bokeh* is an interactive visualization library with the Web in mind, producing browser-rendered output and therefore playing very nicely with IPython Notebook. It's a great achievement, with a design philosophy similar to D3's.[12]

But for the kind of interactive, exploratory dataviz necessary to get a feel for your data and suggest visualizations, I recommend Seaborn. Seaborn extends Matplotlib with some powerful statistical plots and is well integrated with the PyData stack, playing nicely with NumPy, Pandas, and the statistical routines found in Scipy and Statsmodels.

One of the nice things about Seaborn is that it doesn't hide the Matplotlib API, allowing you to tweak your charts with Matplotlib's extensive tools. In this sense, it's not a replacement for Matplotlib and the relevant skills, but a very impressive extension.

To work with Seaborn, simply extend your standard Matplotlib imports:

```
import numpy as np
import pandas as pd
```

```
import seaborn as sns # relies on matplotlib
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Matplotlib provides a number of plotting styles which can be invoked by calling a `use` method with a style key. Let's set the current style to Seaborn's default, which will provide a subtle gray grid to the charts:

```
matplotlib.style.use('seaborn')
```

You can checkout all available styles and their visual effects here.

Many of Seaborn's functions are designed to accept a Pandas `DataFrame`, allowing you to specify, for example, the column values describing 2D scattered points. Let's take our existing x and y arrays from Example 3-9 and use them to make some dummy data.

```
data = pd.DataFrame({'dummy x':x, 'dummy y':y})
```

We now have some `data` with columns of x (`'dummy_x'`) and y (`'dummy_y'`) values. Example 3-12 demonstrates the use of Seaborn's dedicated linear regression plot `lmplot`, which produces the chart in Figure 3-12. Note that for some Seaborn plots, to adjust figure size we pass a size (height) in inches and an aspect ratio (width/height). Note also that Seaborn shares `pyplot`'s global context.

Example 3-12. Linear regression plot with Seaborn

```
data = pd.DataFrame({'dummy x':x, 'dummy y':y})
sns.lmplot(data=data, x='dummy x', y='dummy y', ❶
           height=4, aspect=2) ❷
plt.tight_layout() ❸
plt.savefig('mpl_scatter_seaborn.png') ❸
```

❶ The `x` and `y` arguments specify the column names of the `DataFrame` data which define the coordinates of the plot points.

❷ To set figure size, we provide the height in inches and an aspect ratio of width/height.

❸ Seaborn shares the `pyplot` global context, allowing you to save its plots as you would Matplotlib's.

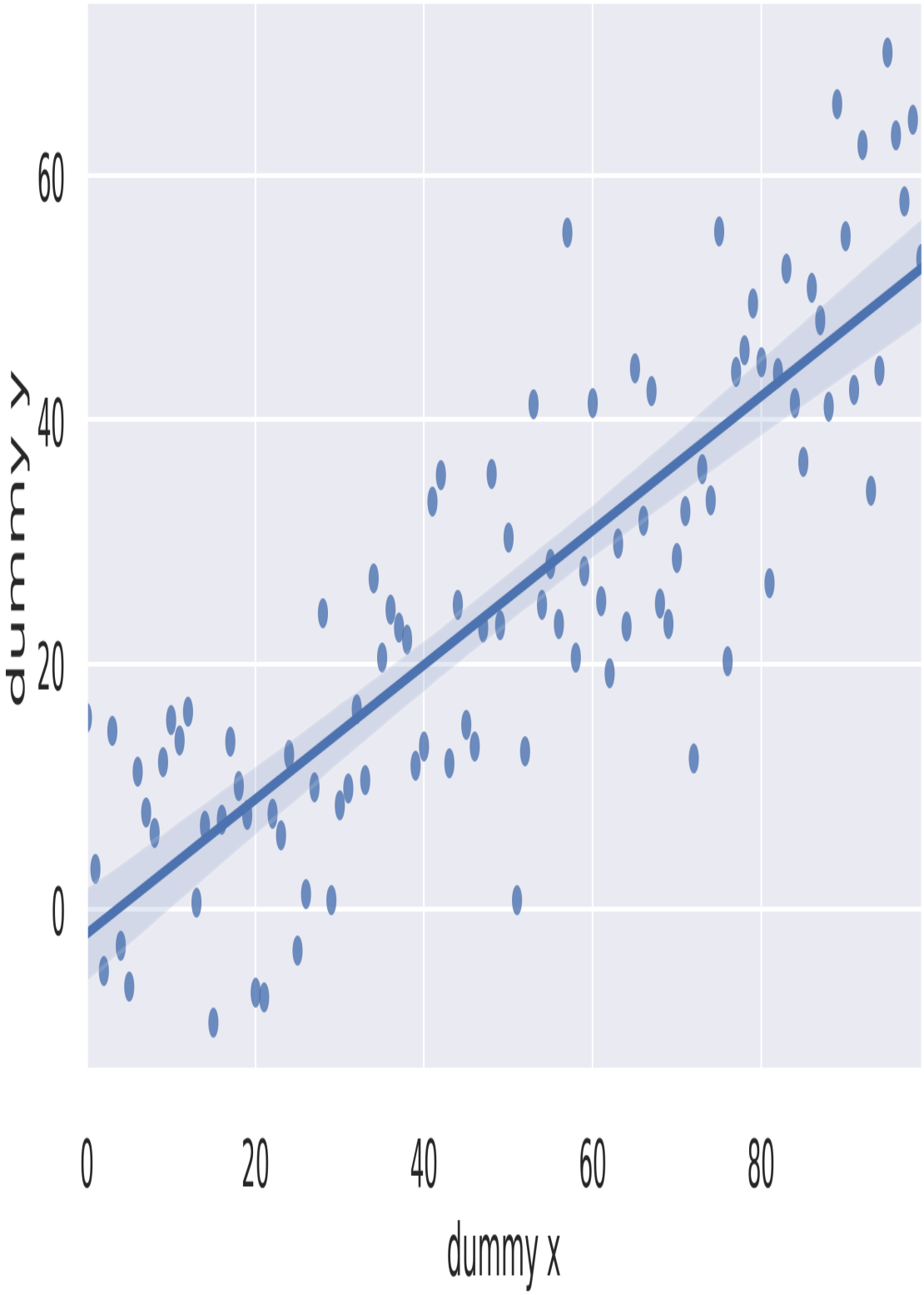

60
40
20
0
dummy y
0
20
40
60
80
dummy x

Figure 3-12. Linear regression plot with Seaborn

As you would expect from a library that places an emphasis on attractive-looking plots, Seaborn allows a lot of visual customization. Let's make a few changes to the look of Figure 3-12 and adjust the confidence interval to the standard error estimate of 68% (see Figure 3-13 for the result):

```
sns.lmplot(data=data, x='dummy x', y='dummy y', height=4,
aspect=2,
           scatter_kws={"color": "slategray"}, ❶
           line_kws={"linewidth": 2, "linestyle":'--', ❷
                     "color": "seagreen"},
           markers='D', ❸
           ci=68) ❹
```

❶ Provide the scatter plot component's keyword arguments, setting our points' color to slate gray.

❷ Provide the line plot component's keyword arguments, setting line width and style.

❸ Sets the plot markers to diamonds using Matplotlib marker code *D*.

❹ We set a confidence interval of 68%, the standard error estimate.

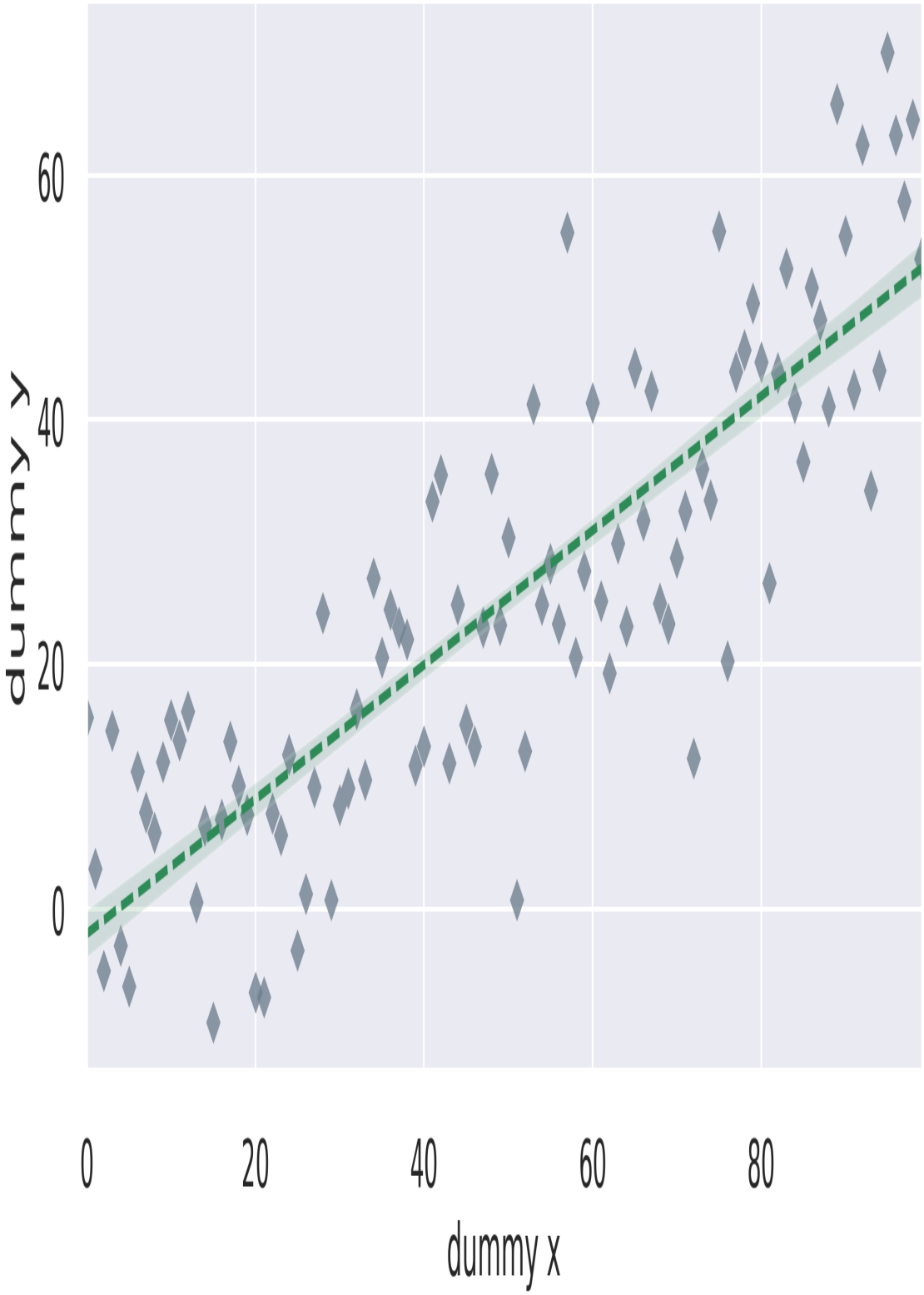

60
40
20
0
dummy y
0
20
40
60
80
dummy x

Figure 3-13. Customizing the Seaborn scatter plot

Seaborn offers a number of useful plots beyond Matplotlib's basic set. Let's take a look at one of the most interesting, using Seaborn's `FacetGrid` to plot reflections of multidimensional data.

FacetGrids

Often referred to as "lattice" or "trellis" plotting, the ability to draw multiple instances of the same plot on different subsets of your dataset is a good way to get a bird's-eye view of your data. Large amounts of information can be presented in one plot, and relationships between the different dimensions can be quickly apprehended. This technique is related to the small multiples popularized by Edward Tufte.

`FacetGrid`s require the data to be in the form of a Pandas `DataFrame` (see [Link to Come]) and in a form referred to by Hadley Whickam, creator of ggplot, as "tidy," meaning each column in the `DataFrame` should be a variable and each row an observation.

Let's use Tips, one of Seaborn's test datasets,[13] to show a `FacetGrid` in action. Tips is a small set of data showing the distribution of tips by various dimensions, such as day of the week or whether the customer was a smoker. First let's load our Tips dataset into a Pandas `DataFrame` using the `load_dataset` method:

```
In [0]: tips = sns.load_dataset('tips')
Out[0]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
...
```

To create a `FacetGrid`, we specify the `tips` `DataFrame` and a column of interest, such as the smoking status of the customer. This column will be used to create our plot groups; in this case, `'smoker=Yes'` and

`'smoker=No'`. We then use the grid's `map` method to create multiple scatter plots of tip size against total bill.

```
g = sns.FacetGrid(tips, col="smoker", height=4, aspect=1)
g.map(plt.scatter, "total_bill", "tip") ❶
```

❶ `map` takes a plot class, in this case `scatter`, and two (`tips`) dimensions required for this scatter plot.

This produces the two scatter plots shown in Figure 3-14, one for each smoker status, with tips and total bills correlated.

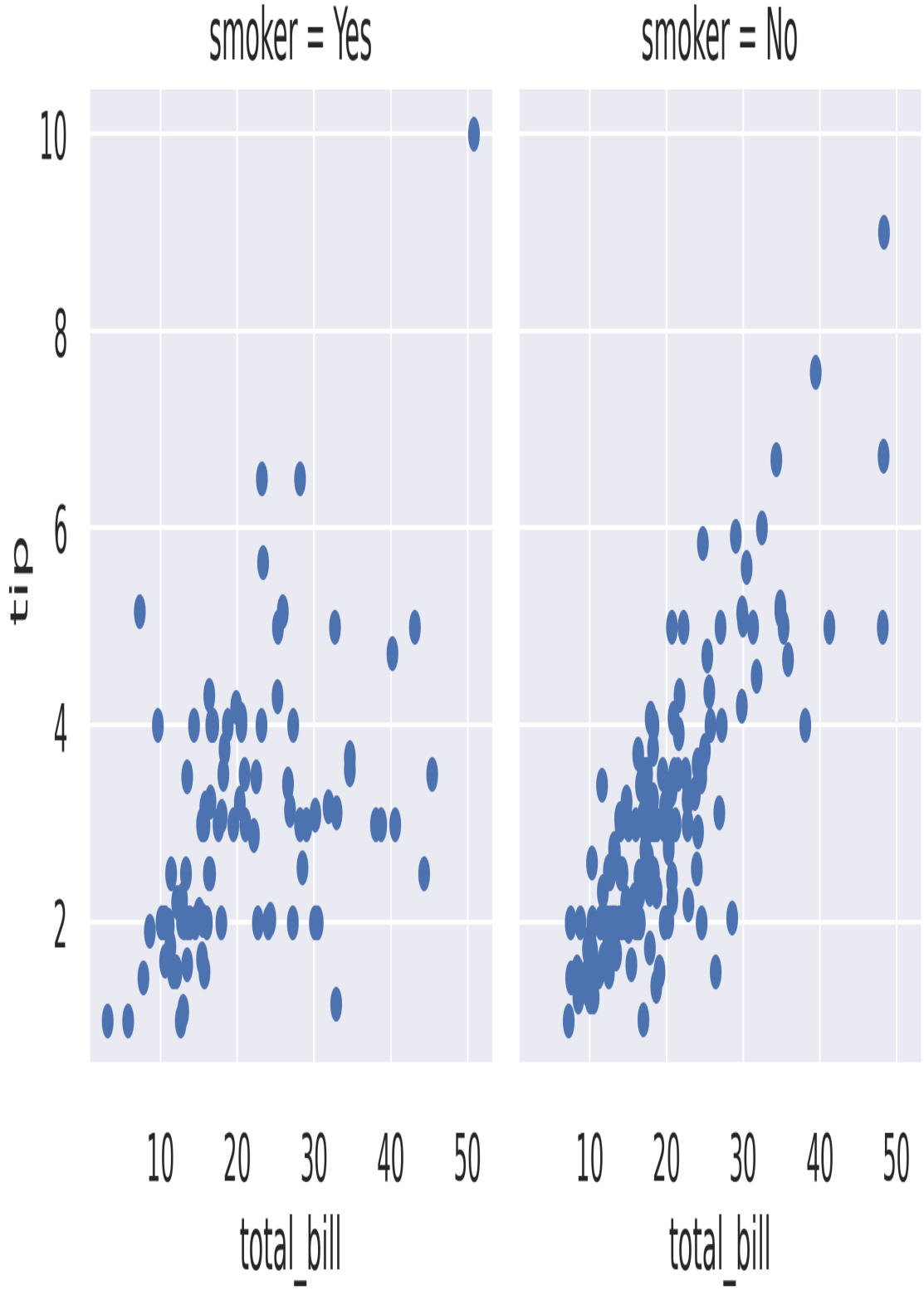

smoker = Yes
smoker = No
tip
10
8
6
4
2
10
20
30
40
50
total_bill
10
20
30
40
50
total_bill

Figure 3-14. A Seaborn FacetGrid using scatter plots

We can include another dimension of the `tips` data by specifying the marker to be used in our scatter plots. Let's make it a red diamond for females and a blue square for males:

```
pal = dict(Female='red', Male='blue')
g = sns.FacetGrid(tips, col="smoker",
                  hue="sex", hue_kws={"marker": ["D", "s"]}, ❶
                  palette=pal, height=4, aspect=1, )
g.map(plt.scatter, "total_bill", "tip", alpha=.4)
g.add_legend();
```

❶ Adds a marker color (`hue`) for the `sex` dimension with diamond (`D`) and square (`s`) shapes, and uses our color palette (`pal`) to make them red and blue.

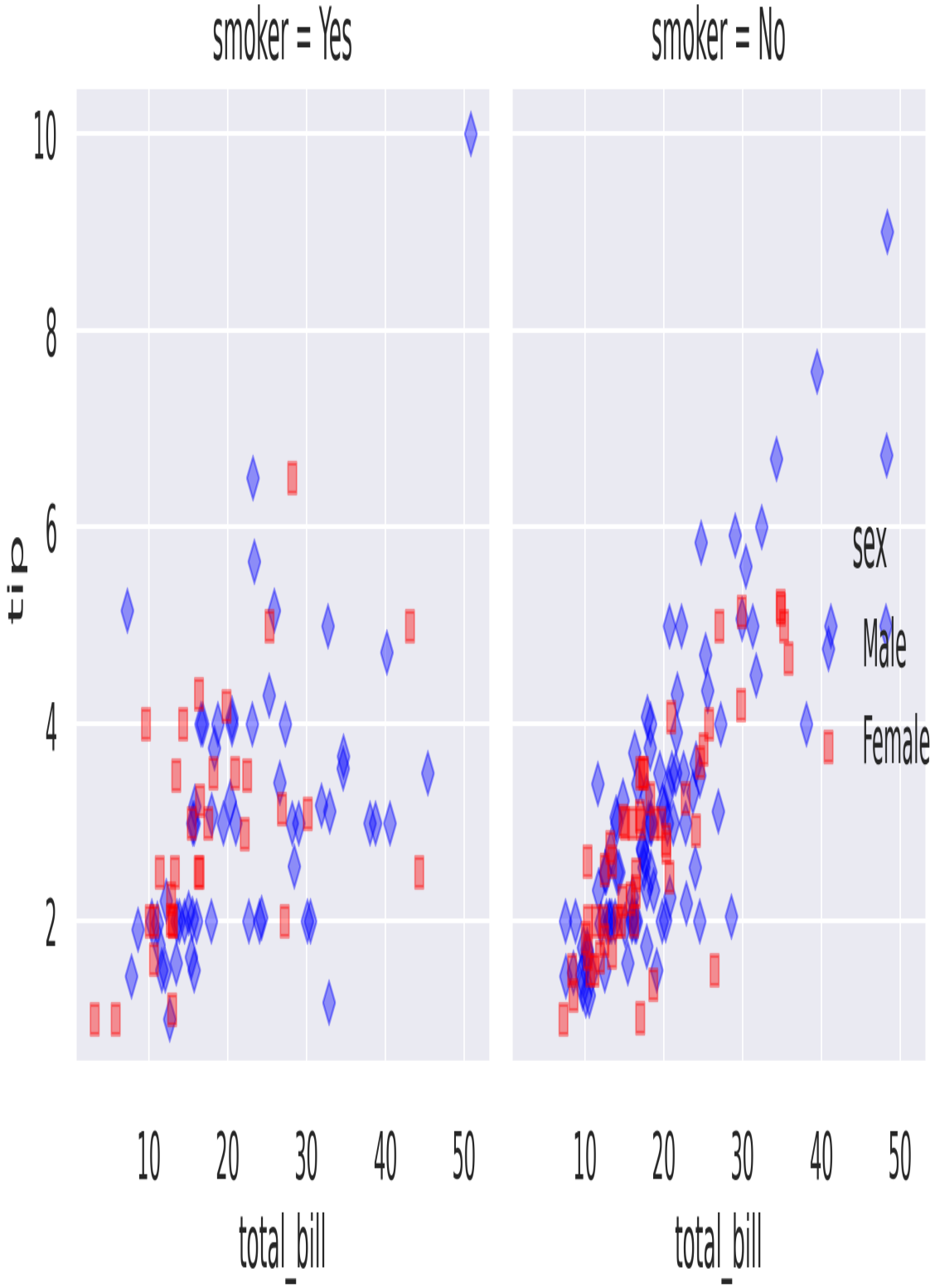

smoker = Yes
smoker = No
tip
total_bill
total_bill
sex
Male
Female
10
8
6
4
2
10
20
30
40
50

Figure 3-15. Scatter plot with diamond and square markers for sex

We can use rows as well as columns to create subsets of the data by dimension. Combining the two allows, with the help of a `regplot`,[14] five dimensions to be explored:

```
pal = dict(Female='red', Male='blue')
g = sns.FacetGrid(tips, col="smoker", row="time", ❶
                  hue="sex", hue_kws={"marker": ["D", "s"]},
                  palette=pal, height=4, aspect=1, )
g.map(sns.regplot, "total_bill", "tip", alpha=.4)
g.add_legend();
```

❶ Adds a time row to separate tips by lunch and dinner.

Figure 3-16 shows four `regplots` producing a linear-regression model fit with confidence intervals for Female and Male hue-groups. The plot titles show the data subset being used, each row having the same time and smoker status.

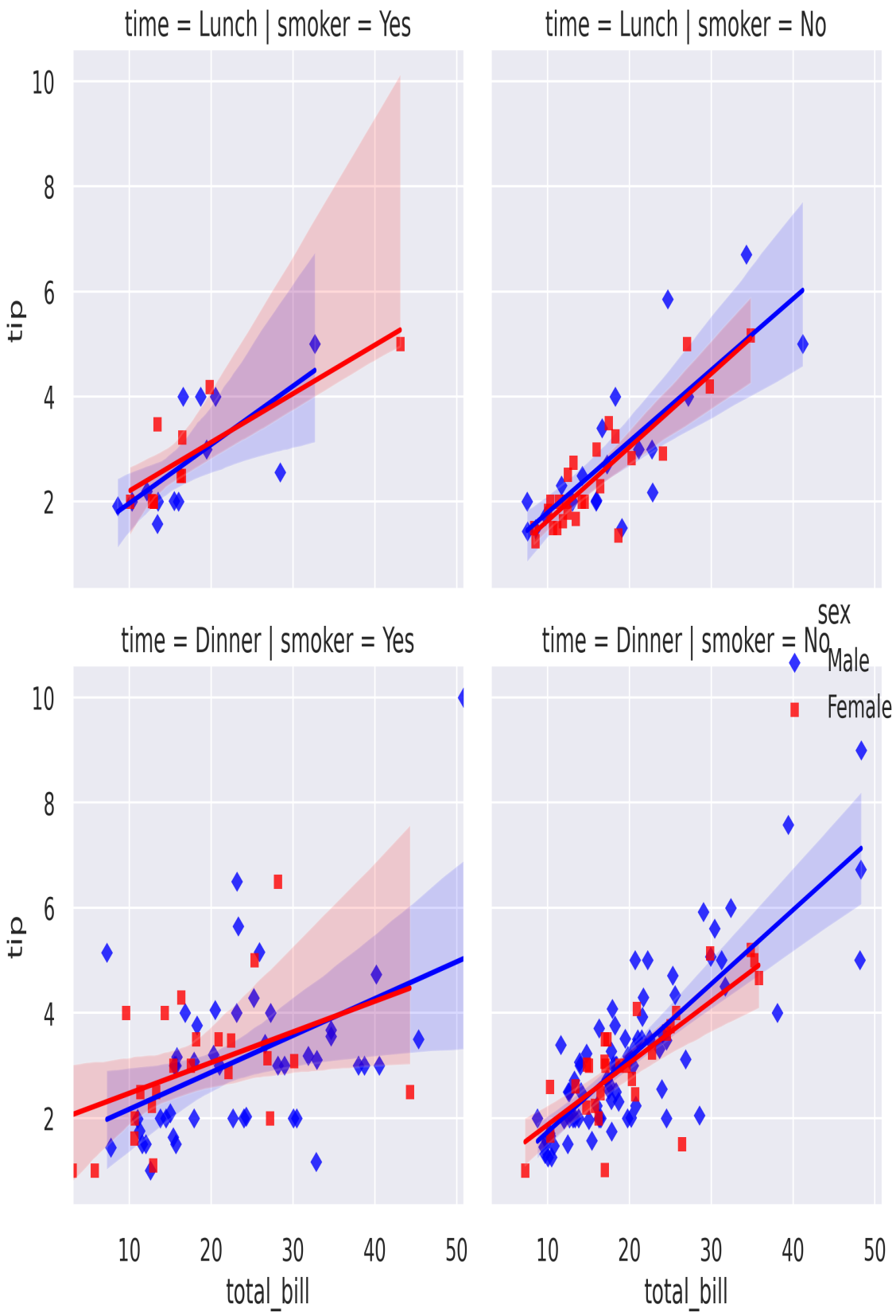

time = Lunch | smoker = Yes
time = Lunch | smoker = No
time = Dinner | smoker = Yes
time = Dinner | smoker = No
tip
total_bill
sex
Male
Female
2
4
6
8
10
10
20
30
40
50

Figure 3-16. Visualizing five dimensions

We can achieve the same effect using the `lmplot` we saw in Example 3-12, which wraps the functionality of `FacetGrid` and `regplot` for convenience. The following code produces Figure 3-16.

```
pal = dict(Female='red', Male='blue')
sns.lmplot(x="total_bill", y="tip", hue="sex",\
markers=["D", "s"], ❶
           col="smoker", row="time", data=tips, palette=pal,
           height=4, aspect=1
           );
```

❶ Note the use of a `markers` keyword as opposed to the `kws_hue` dictionary we used with the `FacetGrid` plot.

`lmplot` offers a nice shortcut to producing `FacetGrid` `regplots`, but `FacetGrid`'s `map` allows you to use the full panoply of Seaborn and Matplotlib charts to create plots on dimensional subsets. It's a very powerful technique and a great way to drill down into your data.

Pairgrids

Pairgrids are another rather cool Seaborn plot type that provide a way to quickly assess multidimensional data. Unlike with `FacetGrids`, you don't divide the dataset into subsets that are then compared by designated dimensions. With Pairgrids, the dataset's dimensions are all compared pair-wise in a square grid. By default all dimensions are compared, but you can specify which ones get plotted by providing a list to the `vars` parameter when declaring the Pairgrid.[15]

Let's demonstrate the utility of this pair-wise comparison by using the classic Iris dataset, showing some vital statistics for a set containing members of three Iris species. First we'll load the example dataset:

```
In [0]: iris = sns.load_dataset('iris')
In [1]: iris.head()
```

```
Out[1]:
   sepal_length sepal_width  petal_length  petal_width  species
0           5.1         3.5           1.4          0.2   setosa
1           4.9         3.0           1.4          0.2   setosa
2           4.7         3.2           1.3          0.2   setosa
...
```

To capture the relationship between petal and sepal dimensions by species, we first create a `PairGrid` object, set its hue to `species`, and then use its mapping methods to create plots on and off the diagonal of the pair-wise grid, producing the charts in Figure 3-17.

```
sns.set_theme(font_scale=1.5) ❶
g = sns.PairGrid(iris, hue="species") ❷
g.map_diag(plt.hist) ❸
g.map_offdiag(plt.scatter) ❹
g.add_legend();
```

❶ Tweaks the font size using Seaborn's `set_theme` method (see here to see the full list of available tweaks).

❷ Sets the markers and subbars to be colored by species.

❸ Places histograms of the species' dimensions on the grid's diagonal.

❹ Uses standard scatter plots to compare the dimensions of the diagonal.

As you can see in Figure 3-17, a few lines of Seaborn goes a long way in creating a richly informative set of plots correlating the different Iris metrics. This plot is known as a scatter-plot matrix and is a great way of finding linear correlations between pairs of variables in a multivariate set. As it stands, there is redundancy in the grid: for example, plots for `sepal_width-petal_length` and `petal_length-septal_width`. `PairGrid` gives you the opportunity to use the redundant plots above or below the main diagonal to provide a different reflection of the data. Check out some of the examples at the Seaborn docs for more info.[16]

I've covered a few of the Seaborn plots in this section, and you'll be seeing a few more when we explore our Nobel Prize dataset in the next chapter. But Seaborn has a lot of other very handy and very powerful plotting tools, mainly of a statistical nature. For further investigation, I'd recommend starting with the main Seaborn documentation. There are some nice examples, a well-documented API, and some good tutorials that should complement what you've learned in this chapter.

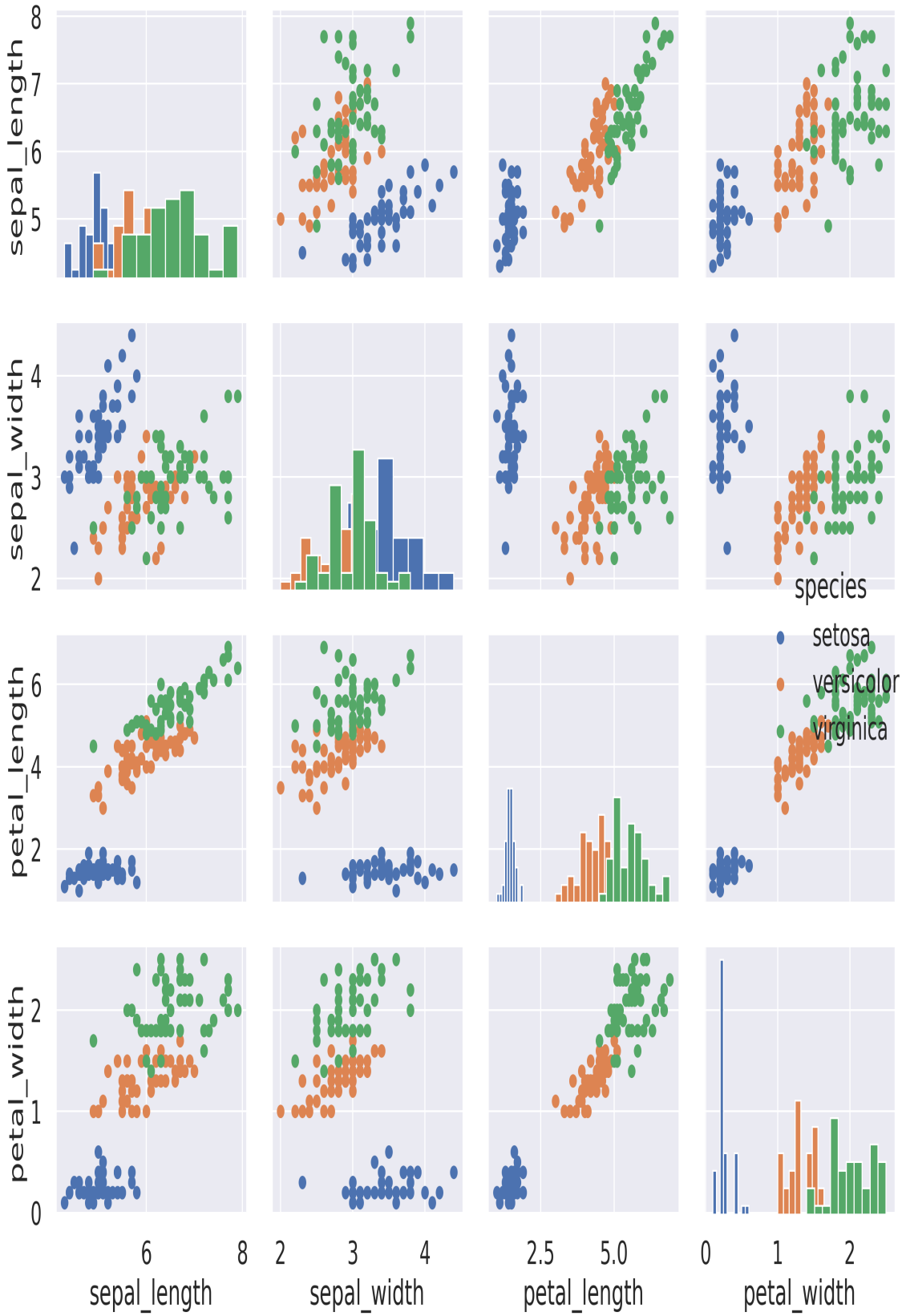

sepal_length
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica

Figure 3-17. Pairgrid summation of Iris measures

Summary

This chapter introduced Matplotlib, Python's plotting powerhouse. It's a big, mature library with lots of documentation and an active community. If you have a particular customization in mind, chances are there's an example out there somewhere. I'd recommend firing up a Jupyter Notebook and playing around with a dataset.

We saw how Seaborn extends Matplotlib with some useful statistical methods and that it has what many consider to be superior aesthetics. It also allows access to the Matplotlib figure and axes internals, allowing full customization if required.

In the next chapter we'll use Matplotlib along with Pandas to explore our freshly scraped and cleaned Nobel dataset. We'll use some of the plot types demonstrated in this chapter and see a few useful new ones.

---

1 If you have errors trying to start a GUI session, try changing the backend setting (e.g., if using OS X and `%matplotlib qt` doesn't work, try `%matplotlib osx`).

2 IPython has a large number of such functions to enable a whole slew of useful extras to the vanilla Python interpreter. Check them out on the IPython website.

3 This was inspired by Matlab.

4 See the docs for more details.

5 See the Matplotlib website for details.

6 As well as providing many formats, it also understands LaTex math mode, which means you can use mathematical symbols in the titles, legends, and the like. This is one of the reasons Matplotlib is much beloved by academics, as it is quite capable of journal-quality images.

7 More details are available on the Matplotlib website.

8 The handy `tight_layout` option assumes grid-layout subplots.

9 It's questionable whether stacked bar charts are a particularly good way of appreciating groups of data. See Solomon Messing's blog for a nice discussion and one example of "good" use.

10 Setting marker size, rather than width or radius, is actually a good default, making it proportional to whatever value we are trying to reflect.

11 It's generally agreed that Matplotlib's defaults aren't that great and making them better is an easy win for any wrapper.

12 Both D3 and Bokeh tip their hats to the classic visualization text, Leland Wilkinson's *The Grammar of Graphics* (Springer).

13 Seaborn has a number of handy datasets, which you can find on GitHub.

14 `regplot` is equivalent to `lmplot`, used in Example 3-12. The latter combines `regplot` and `FacetGrid` for convenience.

15 There are also `x_vars` and `y_vars` parameters enabling you to specify nonsquare grids.

16 For the curious, there's a D3 example which builds a scatter-plot matrix at the bl.ocks.org site.

Chapter 4. Exploring Data with Pandas

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

In the previous chapter, we cleaned the Nobel Prize dataset that we scraped from Wikipedia in [Link to Come]. Now it's time to start exploring our shiny new dataset, looking for interesting patterns, stories to tell, and anything else that could form the basis for an interesting visualization.

First off, let's try to clear our minds and take a long, hard look at the data to hand to get a broad idea of the visualizations suggested. Example 4-1 shows the form of the Nobel dataset, with categorical, temporal, and geographical data.

Example 4-1. Our cleaned Nobel Prize dataset

```
[{
 'category': u'Physiology or Medicine',
 'date_of_birth': u'8 October 1927',
 'date_of_death': u'24 March 2002',
 'gender': 'male',
 'link': u'http://en.wikipedia.org/wiki/C%C3%A9sar_Milstein',
 'name': u'C\xe9sar Milstein',
 'country': u'Argentina',
```

```
'place_of_birth': u'Bah\xeda Blanca ,  Argentina',
'place_of_death': u'Cambridge , England',
'year': 1984,
'born_in': NaN
},
...
}]
```

The data in Example 4-1 suggests a number of *stories* we might want to investigate, among them:

- Gender disparities among the prize winners
- National trends (e.g., which country has most prizes in Economics)
- Details about individual winners, such as their average age on receiving the prize or life expectancy
- Geographical journey from place of birth to adopted country using the `born_in` and `country` fields

These investigative lines form the basis for the coming sections, which will probe the dataset by asking questions of it, such as "How many women other than Marie Curie have won the Nobel Prize for Physics?," "Which countries have the most prizes per capita rather than absolute?," and "Is there a historical trend to prizes by nation, a changing of the guard from old (science) world (big European nations) to new (US and upcoming Asians)?" Before beginning our explorations, let's ready our tools and load our Nobel Prize dataset.

Starting to Explore

To start our exploration, let's fire up a Jupyter notebook from the command line:

```
$ jupyter notebook
```

We'll use the *magic* `matplotlib` command to enable inline plotting:

```
%matplotlib inline
```

Then import the standard set of data exploration modules:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json
import matplotlib
import seaborn as sns
```

Now we'll make a few adjustments to the plotting parameters and the general look and feel of the charts. Make sure to change the style *before* adjusting figure sizes, fonts and the rest:

```
matplotlib.style.use('seaborn') ❶

plt.rcParams['figure.figsize'] = (8, 4) ❷
plt.rcParams['font.size'] = '12'
```

❶ We'll use the seaborn theming for our charts, arguably more attractive than Matplotlib's default.

❷ Sets the default plotting size to eight inches by four.

At the end of [Link to Come], we saved our clean dataset as a `json` file. Let's load the clean data into a Pandas `DataFrame`, ready to begin exploring.

```
df = pd.read_json(open('data/nobel_winners_cleaned.json'))
```

Let's get some basic information about our dataset's structure:

```
df.info()

&lt;class 'pandas.core.frame.DataFrame'&gt;
RangeIndex: 858 entries, 0 to 857
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
```

```
---  ------           --------------  -----
 0   category         858 non-null    object
 1   country          858 non-null    object
 2   date_of_birth    858 non-null    datetime64[ns, UTC]
 3   date_of_death    559 non-null    datetime64[ns, UTC]
 4   gender           858 non-null    object
 5   link             858 non-null    object
 6   name             858 non-null    object
 7   place_of_birth   831 non-null    object
 8   place_of_death   524 non-null    object
 9   text             858 non-null    object
 10  year             858 non-null    int64
 11  award_age        858 non-null    int64
 12  born_in          102 non-null    object
dtypes: datetime64[ns, UTC](2), int64(2), object(9)
memory usage: 87.3+ KB
```

Note that our dates of birth and death columns have the standard Pandas datatype of `object`. In order to make date comparisons, we'll need to convert those to the datetime type, `datetime64`. We can use Pandas' `to_datetime` method to achieve this conversion:

```
df.date_of_birth = pd.to_datetime(df.date_of_birth)
df.date_of_death = pd.to_datetime(df.date_of_death)
```

Running `df.info()` should now show two datetime columns:

```
df.info()

...
date_of_birth     858 non-null datetime64[ns, UTC] ❶
date_of_death     559 non-null datetime64[ns, UTC]
...
```

❶ UTC is the primary time standard by which the world regulates clocks and time. It's almost always desirable to work to this standard.

`to_datetime` usually works without needing extra arguments, but it's worth checking the converted columns to make sure. In the case of our Nobel Prize dataset, everything checks out.

Plotting with Pandas

Both Pandas `Series` and `DataFrames` have integrated plotting, which wraps the most common Matplotlib charts, a few of which we explored in the last chapter. This makes it easy to get quick visual feedback as you interact with your `DataFrame`. And if you want to visualize something a little more complicated, the Pandas containers will play nicely with vanilla Matplotlib. You can also adapt the plots produced by Pandas using standard Matplotlib customizations.

Let's look at an example of Pandas' integrated plotting, starting with a basic plot of gender disparity in Nobel Prize wins. Notoriously, the Nobel Prize has been distributed unequally among the sexes. Let's get a quick feel for that disparity by using a bar plot on the *gender* category. Example 4-2 produces Figure 4-1, showing the huge difference, with men receiving 811 of the 858 prizes in our dataset.

Example 4-2. Using Pandas' integrated plotting to see gender disparities

```
by_gender = df.groupby('gender')
by_gender.size().plot(kind='bar')
```

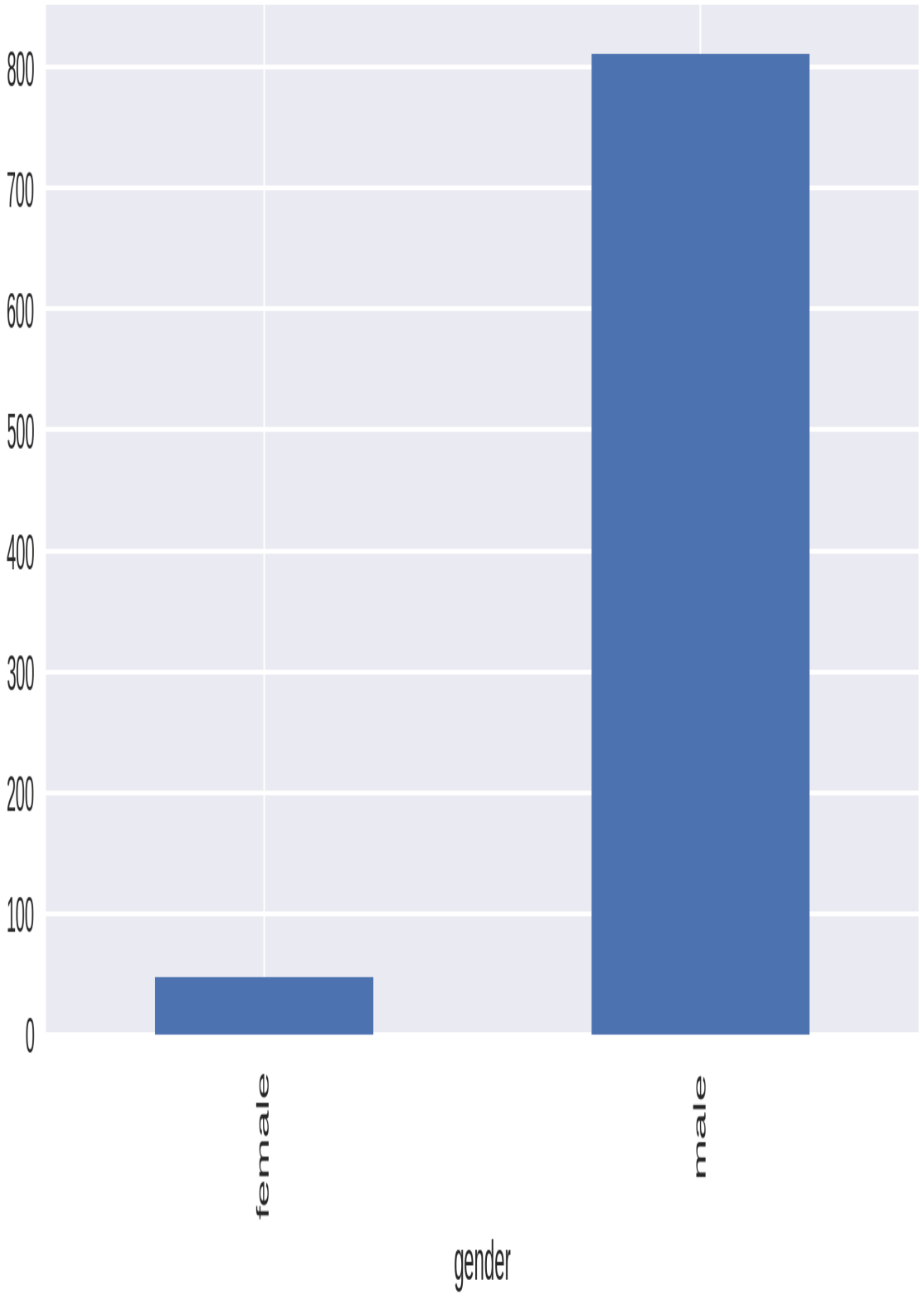

800
700
600
500
400
300
200
100
0
female
male
gender

Figure 4-1. Prize counts by gender

In Example 4-2, the `Series` produced by the gender group's `size` method has its own integrated `plot` method, which turns the raw numbers into a chart:

```
by_gender.size()
Out:
gender
female     47
male      811
dtype: int64
```

In addition to the default line plot, the Pandas `plot` method takes a `kind` argument to select among other possible plots. Among the more commonly used are:

- `bar` or `barh` (*h* for horizontal) for bar plots
- `hist` for a histogram
- `box` for a box plot
- `scatter` for scatter plots

You can find a full list of Pandas' integrated plots in the docs as well as some Pandas plotting functions that take `DataFrames` and `Series` as arguments.

Let's extend our investigation into gender disparities and start extending our plotting know-how.

Gender Disparities

Let's break down the gender numbers shown in Figure 4-1 by category of prize. Pandas' `groupby` method can take a list of columns to group by, with each group being accessed by multiple keys.

```
by_cat_gen = df.groupby(['category','gender'])

by_cat_gen.get_group(('Physics', 'female'))[['name', 'year']] ❶
Out:
                        name  year
269    Maria Goeppert-Mayer  1963
612  Marie Skłodowska-Curie  1903
```

❶ Gets a group using a `category` and `gender` key.

Using the `size` method to get the size of these groups returns a `Series` with a `MultiIndex` that labels the values by both category and gender:

```
by_cat_gen.size()
Out:
category                gender
Chemistry               female      4
                        male      167
Economics               female      1
                        male       74
...
Physiology or Medicine  female     11
                        male      191
dtype: int64
```

We can plot this multi-indexed `Series` directly, using `hbar` as the `kind` argument to produce a horizontal bar chart. This code produces Figure 4-2:

```
by_cat_gen.size().plot(kind='barh')
```

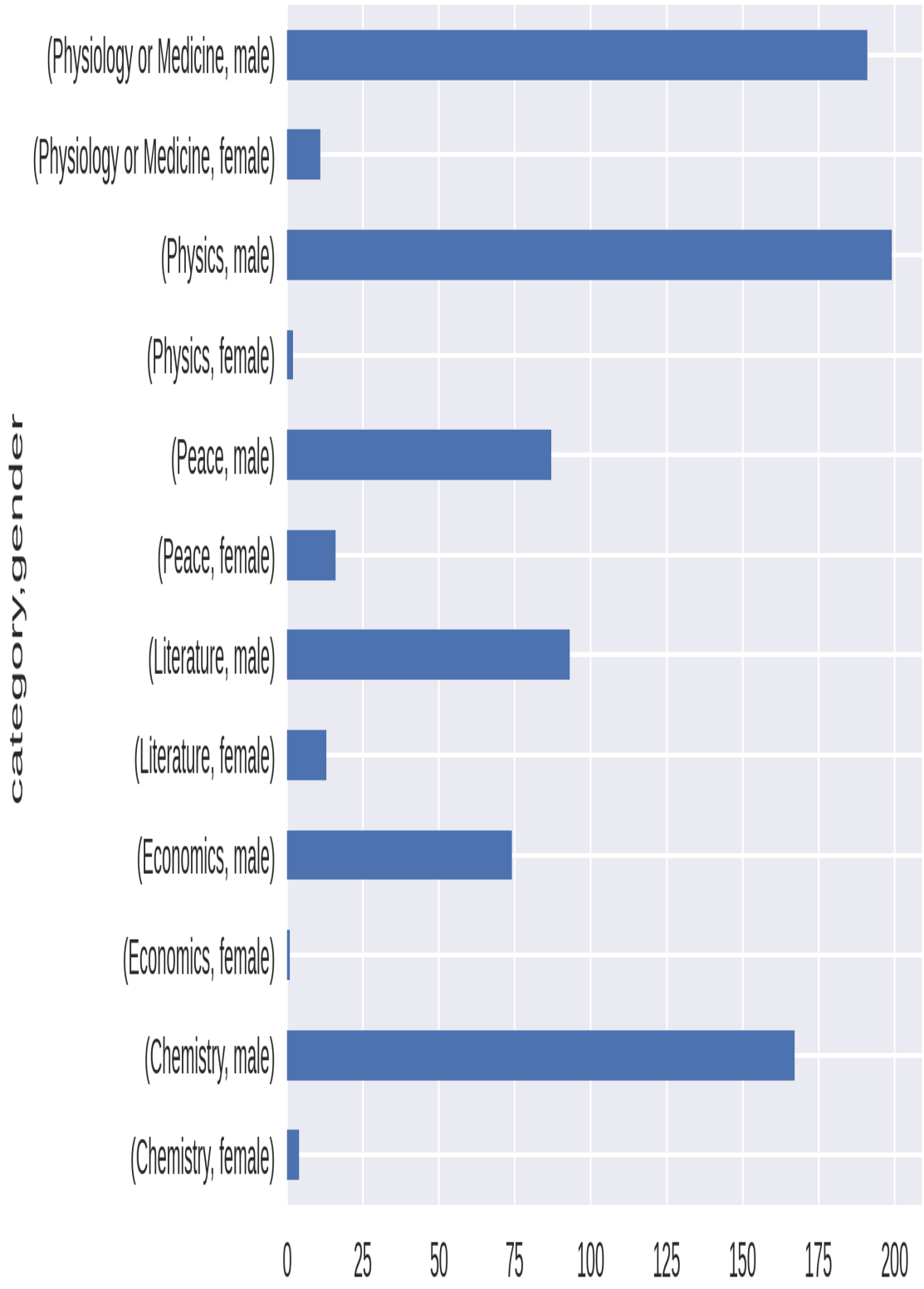

category,gender
(Physiology or Medicine, male)
(Physiology or Medicine, female)
(Physics, male)
(Physics, female)
(Peace, male)
(Peace, female)
(Literature, male)
(Literature, female)
(Economics, male)
(Economics, female)
(Chemistry, male)
(Chemistry, female)
0
25
50
75
100
125
150
175
200

Figure 4-2. Plotting multikey groups

Figure 4-2 is a little crude and makes comparing gender disparities harder than it should be. Let's go about refining our charts to make those disparities clearer.

Unstacking Groups

Figure 4-2 isn't the easiest chart to read, even were we to improve the sorting of the bars. Handily, Pandas `Series` have a cool `unstack` method that takes the multiple indices—in this case, gender and category—and uses them as columns and indices, respectively, to create a new `DataFrame`. Plotting this `DataFrame` gives a much more usable plot, as it compares prize wins by gender. The following code produces Figure 4-3:

```
by_cat_gen.size().unstack().plot(kind='barh')
```

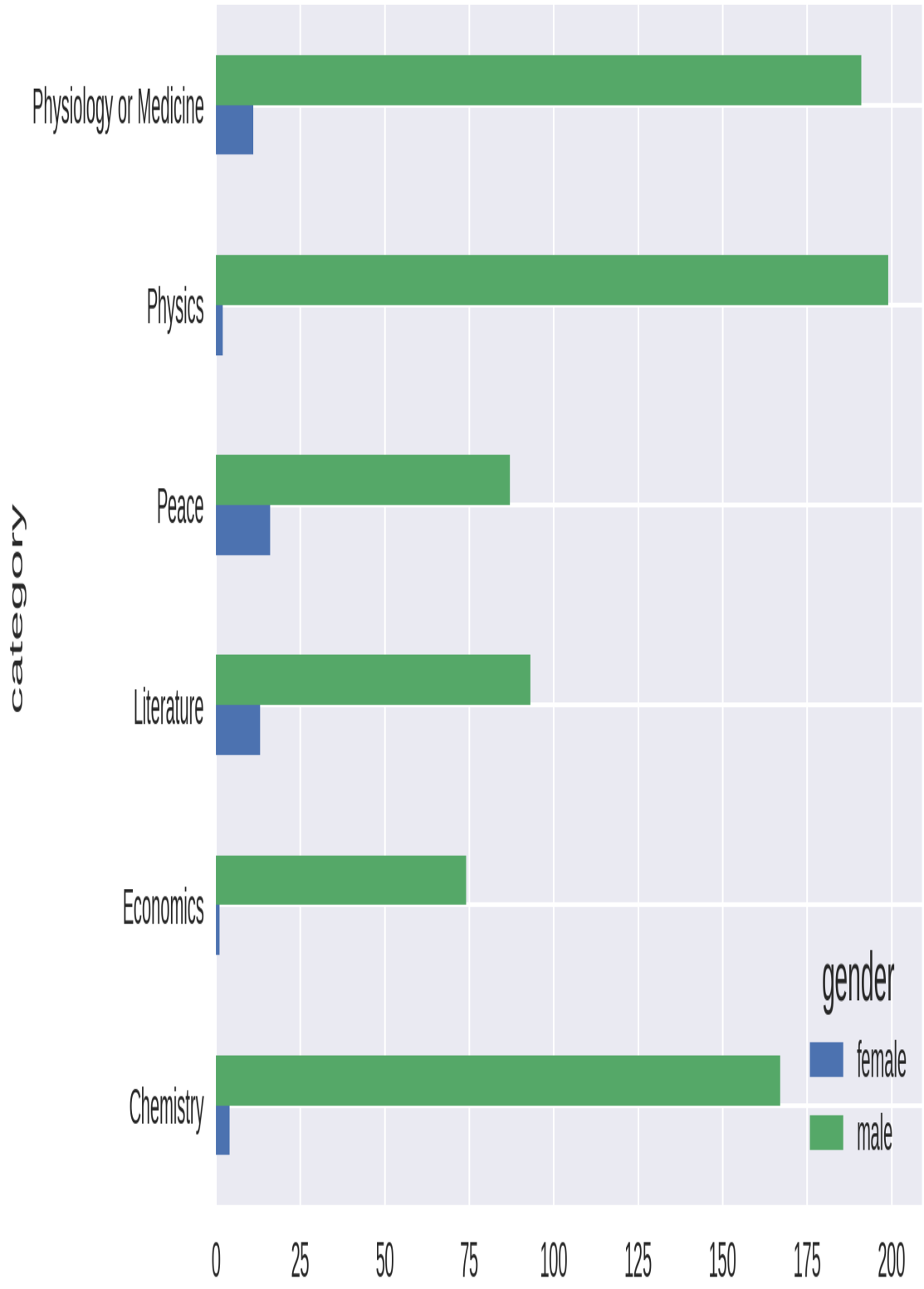

Physiology or Medicine
Physics
Peace
Literature
Economics
Chemistry
category
0
25
50
75
100
125
150
175
200
gender
female
male

Figure 4-3. Unstacked Series of group sizes

Let's improve Figure 4-3 by ordering the bar groups by number of female winners (low to high) and adding a total winners bar group for comparison. Example 4-3 produces the chart in Figure 4-4.

Example 4-3. Sorting and summing our gender groups

```
cat_gen_sz = by_cat_gen.size().unstack()
cat_gen_sz['total'] = cat_gen_sz.sum(axis=1) ❶
cat_gen_sz = cat_gen_sz.sort_values(by='female', ascending=True) ❷
cat_gen_sz[['female', 'total', 'male']].plot(kind='barh')
```

❶ Sums the male and female totals. The `axis` argument is `0` for index sum, `1` for columns.

❷ Sorts the rows using the `female` field, from low to high.

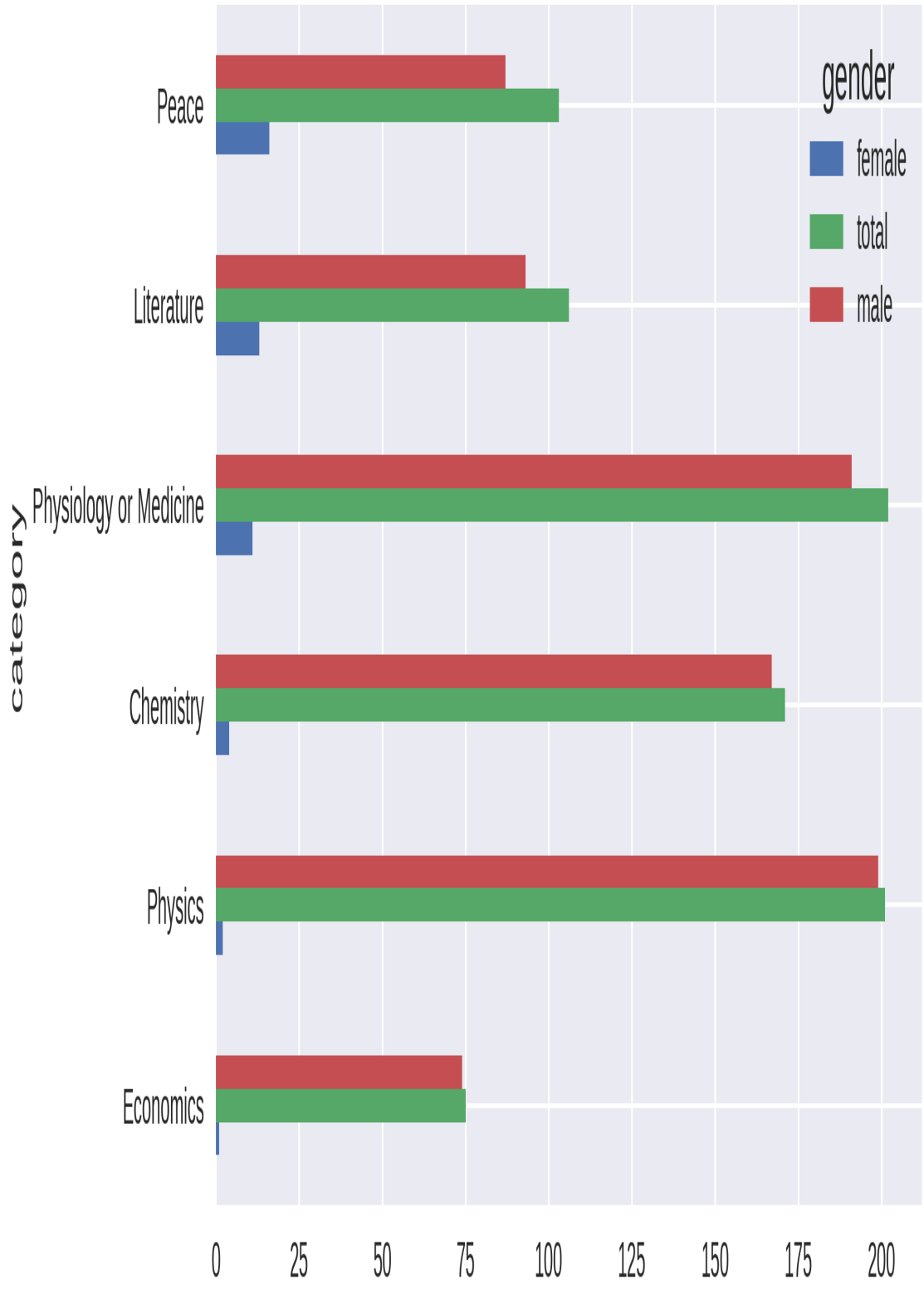

gender
female
total
male
category
Peace
Literature
Physiology or Medicine
Chemistry
Physics
Economics
0
25
50
75
100
125
150
175
200

Figure 4-4. Bars ordered by number of female winners

Ignoring Economics, a recent and contentious addition to the Nobel Prize categories, Figure 4-4 shows that the largest discrepancy in the number of male and female prize winners is in Physics, with only two female winners. Let's remind ourselves who they are:

```
df[(df.category == 'Physics') & (df.gender == 'female')]\
    [['name', 'country','year']]

Out:
                        name    country  year
269    Maria Goeppert-Mayer  United States  1963
612  Marie Skłodowska-Curie         Poland  1903
```

While most people will have heard of Marie Curie, who is actually one of the four illustrious winners of two Nobel Prizes, few have heard of Maria Goeppert-Mayer.[1] This ignorance is surprising, given the drive to encourage women into science. I would want my visualization to enable people to discover and learn a little about Maria Goeppert-Mayer.

Historical Trends

It would be interesting to see if there has been any increase in female prize allocation in recent years. One way to visualize this would be as grouped bars over time. Let's run up a quick plot, using `unstack` as in Figure 4-3 but using the year and gender columns.

```
by_year_gender = df.groupby(['year','gender'])
year_gen_sz = by_year_gender.size().unstack()
year_gen_sz.plot(kind='bar', figsize=(16,4))
```

Figure 4-5, the hard-to-read plot produced, is only functional. The trend of female prize distributions can be observed, but the plot has many problems. Let's use Matplotlib's and Pandas' eminent flexibility to fix them.

gender
female
male
0
2
4
6
8
10
12
14
year

Figure 4-5. Prizes by year and gender

The first thing we need to do is reduce the number of x-axis labels. By default, Matplotlib will label each bar or bar group of a bar plot, which in the case of our hundred years of prizes creates a mess of labels. What we need is the ability to thin out the number of axis labels as desired. There are various ways to do this in Matplotlib; I'll demonstrate the one I've found to be most reliable. It's the sort of thing you're going to want to reuse, so it makes sense to stick it in a dedicated function. Example 4-4 shows a function to reduce the number of ticks on our x-axis.

Example 4-4. Reducing the number of x-axis labels

```
def thin_xticks(ax, tick_gap=10, rotation=45):
    """ Thin x-ticks and adjust rotation """
    ticks = ax.xaxis.get_ticklocs() ❶
    ticklabels = [l.get_text()\
                  for l in ax.xaxis.get_ticklabels()] ❶
    ax.xaxis.set_ticks(ticks[::tick_gap]) ❷
    ax.xaxis.set_ticklabels(ticklabels[::tick_gap],\ ❷
                            rotation=rotation) ❸
    ax.figure.show()
```

❶ Gets the existing locations and labels of the x-ticks, currently one per bar.

❷ Sets the new tick locations and labels at an interval of `tick_gap` (default `10`).

❸ Rotates the labels for readability, by default on an upward diagonal.

As well as needing to reduce the number of ticks, the x-axis in Figure 4-5 has a discontinuous range, missing the years 1939–1945 of WWII, during which no Nobel Prizes were presented. We want to see such gaps, so we need to set the x-axis range manually to include all years from the start of the Nobel Prize to the current day.

The current unstacked group sizes use an automatic year index:

```
by_year_gender = df.groupby(['year', 'gender'])
by_year_gender.size().unstack()
Out:
gender  female  male
year
1901       NaN     6.0
1902       NaN     7.0
...
2014         2.0    11.0
[111 rows x 2 columns]
```

In order to see any gaps in the prize distribution, all we have to do is reindex this `Series` with one containing the full range of years:

```
new_index = pd.Index(np.arange(1901, 2015), name='year') ❶
by_year_gender = df.groupby(['year','gender'])
year_gen_sz = by_year_gender.size().unstack()
  .reindex(new_index) ❷
```

❶ Here we create a full-range index named `year`, covering all the Nobel Prize years.

❷ We replace our discontinuous index with the new continuous one.

Another problem with Figure 4-5 is the excessive number of bars. Although we do get male and female bars side by side, it looks messy and has aliasing artifacts too. It's better to have dedicated male and female plots but stacked so as to allow easy comparison. We can achieve this using the subplotting method we saw in "Axes and Subplots", using the Pandas data but customizing the plot using our Matplotlib know-how. Example 4-5 shows how to do this, producing the plot in Figure 4-6.

Example 4-5. Stacked gender prizes by year

```
new_index = pd.Index(np.arange(1901, 2015), name='year')
by_year_gender = df.groupby(['year','gender'])

year_gen_sz = by_year_gender.size().unstack().reindex(new_index)

fig, axes = plt.subplots(nrows=2, ncols=1, ❶
            sharex=True, sharey=True, figsize=(16, 8)) ❷
```

```
ax_f = axes[0]
ax_m = axes[1]

fig.suptitle('Nobel Prize-winners by gender', fontsize=16)

ax_f.bar(year_gen_sz.index, year_gen_sz.female) ❸
ax_f.set_ylabel('Female winners')

ax_m.bar(year_gen_sz.index, year_gen_sz.male)
ax_m.set_ylabel('Male winners')

ax_m.set_xlabel('Year')
```

❶ Creates two axes, on a two (row) by one (column) grid.

❷ We'll share the x- and y-axes, which will make comparisons between the two plots sensible.

❸ We provide the axis's bar chart (`bar`) method with the continuous year index and the unstacked gender columns.

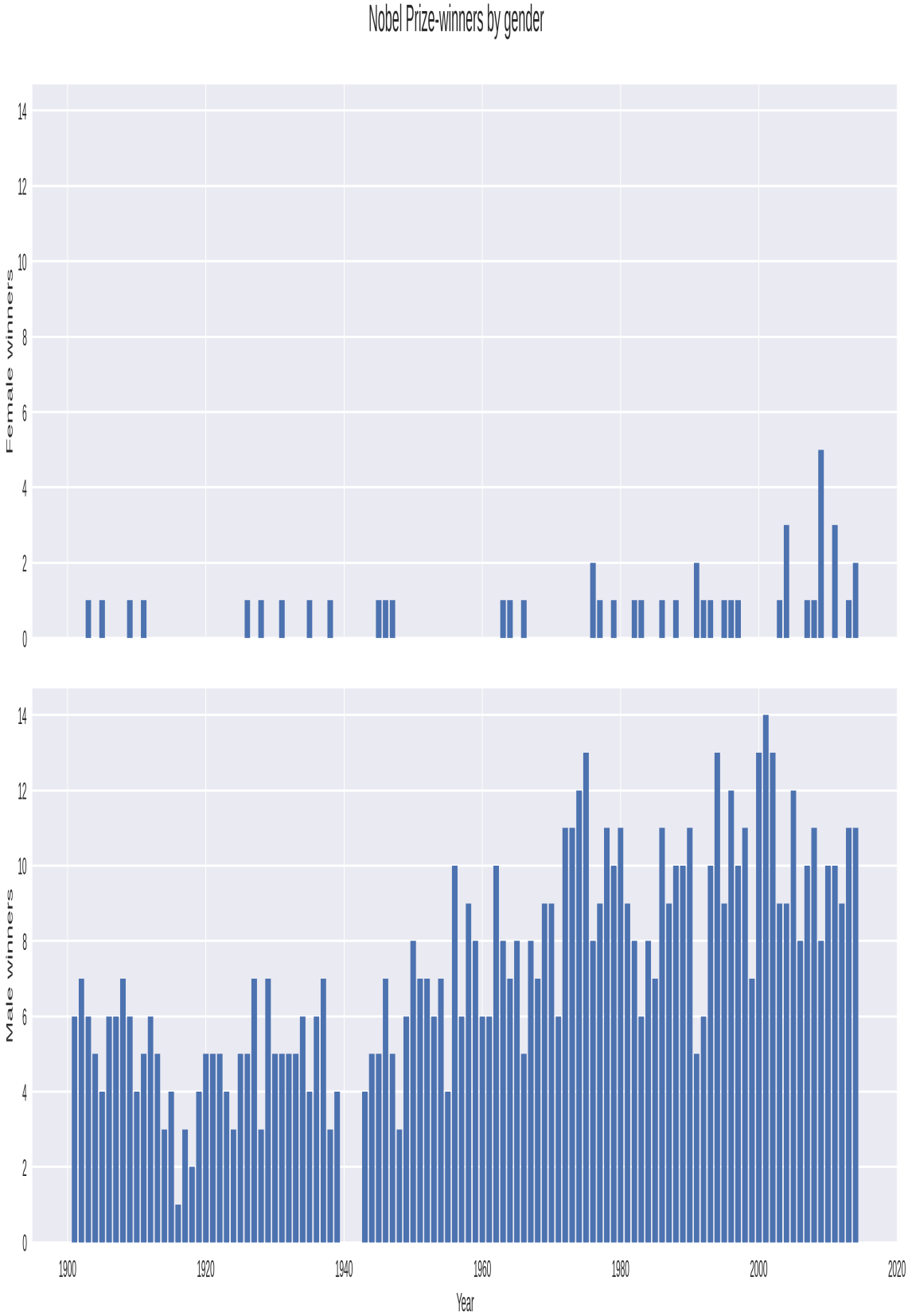

Nobel Prize-winners by gender
Female winners
Male winners
0
2
4
6
8
10
12
14
1900
1920
1940
1960
1980
2000
2020
Year

Figure 4-6. Prizes by year and gender, on two stacked axes

So the take-home from our investigation into gender distributions is that there is a huge discrepancy but, as shown by Figure 4-6, a slight improvement in recent years. Moreover, with Economics being an outlier, the difference is greatest in the sciences. Given the fairly small number of female prize winners, there's not a lot more to be seen here.

Let's now take a look at national trends in prize wins and see if there are any interesting nuggets for visualization.

National Trends

The obvious starting point in looking at national trends is to plot the absolute number of prize winners. This is easily done in one line of Pandas, broken up here for ease of reading:

```
df.groupby('country').size().order(ascending=False)\
        .plot(kind='bar', figsize=(12,4))
```

This produces Figure 4-7, showing the United States with the lion's share of prizes.

The absolute number of prizes will be bound to favor countries with large populations. Let's look at a fairer comparison, visualizing prizes per capita.

300
200
100
0
United States
United Kingdom
Germany
France
Sweden
Switzerland
Japan
Russia
Netherlands
Austria
Italy
Denmark
Norway
Canada
Israel
Belgium
Australia
South Africa
Ireland
Spain
Poland
Argentina
China
India
Egypt
Hungary
Liberia
Portugal
Czech Republic
Chile
Pakistan
East Timor
Mexico
Greece
Guatemala
Costa Rica
Turkey
Bangladesh
Taiwan
Azerbaijan
Venezuela
Vietnam
Yemen
Colombia
Palestinian Territory
Saint Lucia
Cyprus
Iceland
Nigeria
Myanmar (Burma)
Korea, South
Kenya
Finland
Ghana
Iran
Yugoslavia
country

Figure 4-7. Absolute prize wins by country

Prize Winners per Capita

The absolute number of prize winners is bound to favor larger countries, which raises the question, how do the numbers stack up if we account for population sizes? In order to test prize haul per capita, we need to divide the absolute prize numbers by population size. In [Link to Come], we downloaded some country data from the Web and stored it to MongoDB. Let's retrieve it now and use it to produce a plot of prizes relative to population size.

First let's get the national group sizes, with country names as index labels:

```
nat_group = df.groupby('country')
ngsz = nat_group.size()
ngsz.index
Out:
Index([u'Argentina', u'Australia', u'Austria', u'Azerbaijan',...
```

Now let's load our country data into a `DataFrame` and remind ourselves of the data it contains:

```
df_countries = pd.read_json('data/winning_country_data.json',\
                            orient='index')

df_countries.loc['Japan'] # coutries indexed by name

Out:
gini                   38.1
name                  Japan
alpha3Code              JPN
area                377930.0
latlng         [36.0, 138.0]
capital                Tokyo
population         127080000
Name: Japan, dtype: object
```

Our country dataset is already indexed to its `name` column. If we add to it the `ngsz` national group-size `Series`, which also has a country name

index, the two will combine on the shared indices, giving our country data a new `nobel_wins` column. We can then use this new column to create a `nobel_wins_per_capita` by dividing it by population size:

```
df_countries = df_countries.set_index('name')
df_countries['nobel_wins'] = ngsz
df_countries['nobel_wins_per_capita'] =\
    df_countries.nobel_wins / df_countries.population
```

We now need only sort the `df_countries DataFrame` by its new `nobel_wins_per_cap` column and plot the Nobel Prize wins per capita, producing Figure 4-8.

```
df.countries.sort_values(by='nobel_wins_per_capita',\
    ascending=False).nobel_per_capita.plot(kind='bar',\
    figsize=(12, 4))
```

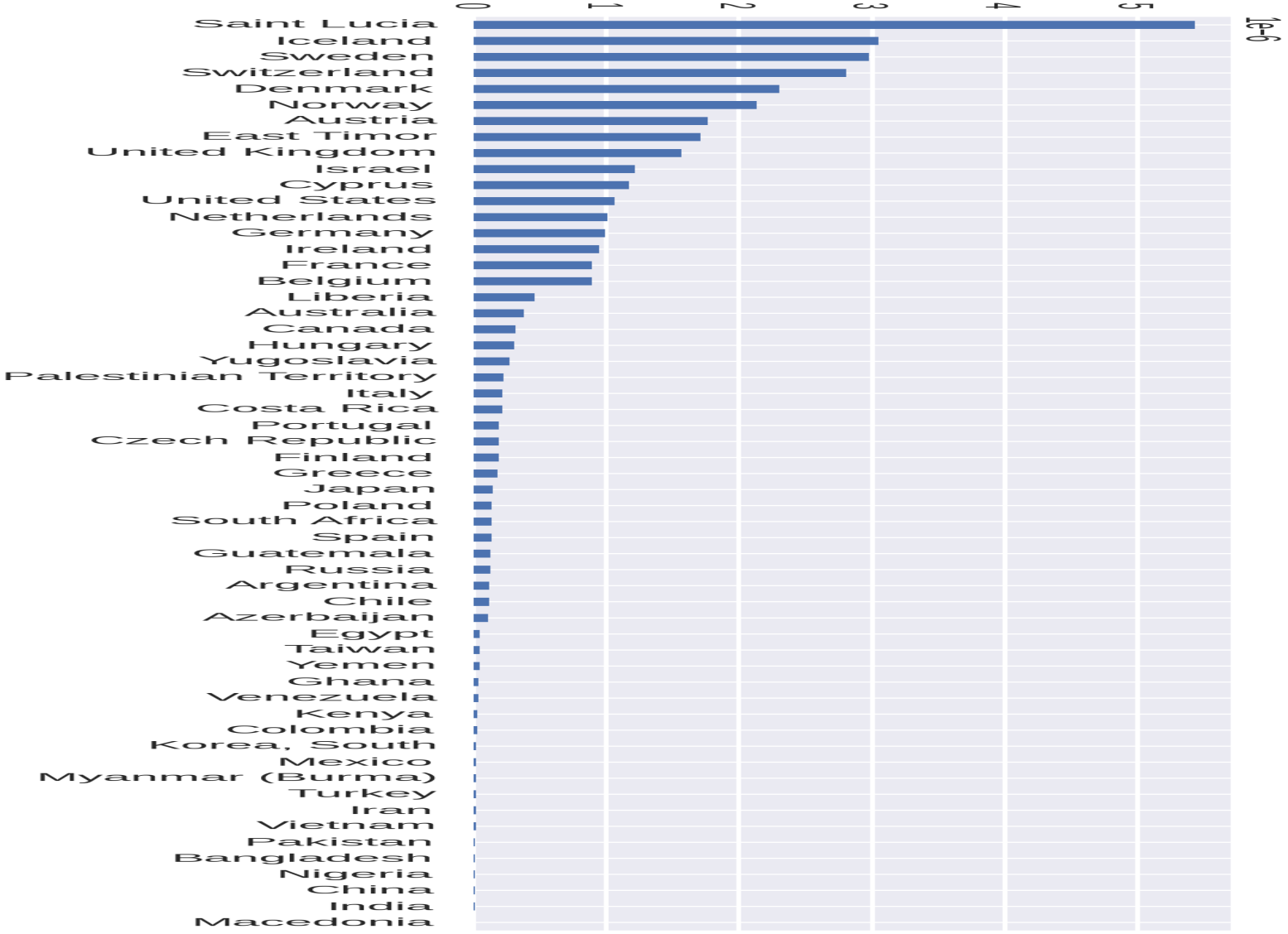

0
1
2
3
4
5
1e-6
Saint Lucia
Iceland
Sweden
Switzerland
Denmark
Norway
Austria
East Timor
United Kingdom
Israel
Cyprus
United States
Netherlands
Germany
Ireland
France
Belgium
Liberia
Australia
Canada
Hungary
Yugoslavia
Palestinian Territory
Italy
Costa Rica
Portugal
Czech Republic
Finland
Greece
Japan
Poland
South Africa
Spain
Guatemala
Russia
Argentina
Chile
Azerbaijan
Egypt
Taiwan
Yemen
Ghana
Venezuela
Kenya
Colombia
Korea, South
Mexico
Myanmar (Burma)
Turkey
Iran
Vietnam
Pakistan
Bangladesh
Nigeria
China
India
Macedonia

Figure 4-8. National prize numbers per capita

This shows the Caribbean Island of Saint Lucia taking top place. Home to the Nobel Prize–winning poet Derek Walcott, its small population of 175,000 gives it a high Nobel Prizes per capita.

Let's see how things stack up with the larger countries by filtering the results for countries that have won more than two Nobel Prizes:

```
df_countries[df_countries.nobel_wins > 2]\
        .sort_values(by='nobel_wins_per_capita',
ascending=False)\
        .nobel_wins_per_capita.plot(kind='bar')
```

The results in Figure 4-9 show the Scandinavian countries and Switzerland punching above their weight.

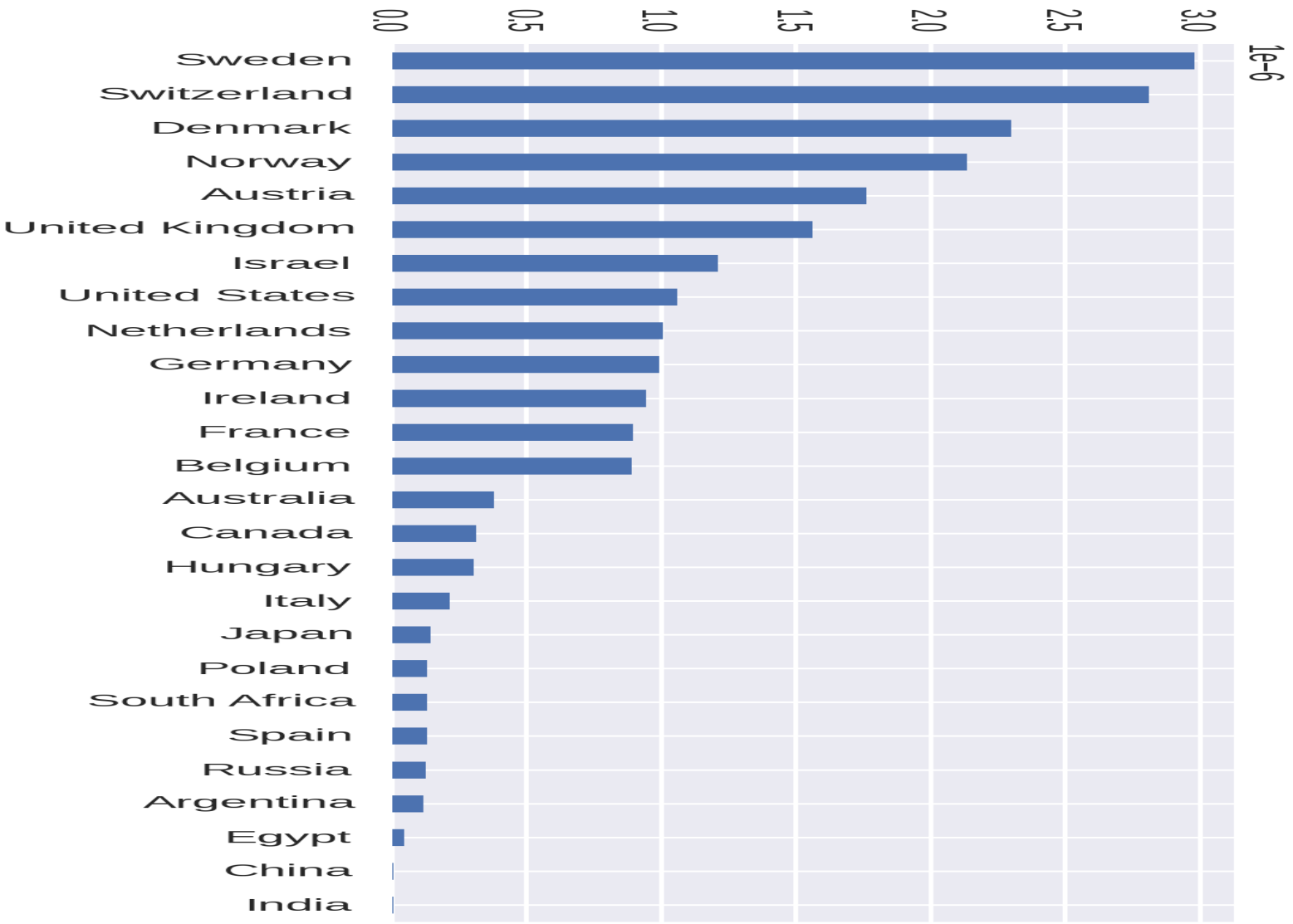

0.0
0.5
1.0
1.5
2.0
2.5
3.0
1e-6
Sweden
Switzerland
Denmark
Norway
Austria
United Kingdom
Israel
United States
Netherlands
Germany
Ireland
France
Belgium
Australia
Canada
Hungary
Italy
Japan
Poland
South Africa
Spain
Russia
Argentina
Egypt
China
India

Figure 4-9. National prize numbers per capita, filtered for three or more wins

Changing the metric for national prize counts from absolute to per capita makes a big difference. Let's now refine our search a little and focus on the prize categories, looking for interesting nuggets there.

Prizes by Category

Let's drill down a bit into the absolute prize data and look at wins by category. This will require grouping by country and category columns, getting the size of those groups, unstacking the resulting `Series` and then plotting the columns of the resulting `DataFrame`. First we get our categories with country group sizes:

```
nat_cat_sz = df.groupby(['country', 'category']).size()
.unstack()
nat_cat_sz
Out:
category     Chemistry  Economics  Literature  Peace  \...
country
Argentina            1        NaN         NaN      2
Australia          NaN          1           1    NaN
Austria              3          1           1      2
Azerbaijan         NaN        NaN         NaN    NaN
Bangladesh         NaN        NaN         NaN      1
```

We then use the `nat_cat_sz` `DataFrame` to produce subplots for the six Nobel Prize categories:

```
COL_NUM = 2
ROW_NUM = 3

fig, axes = plt.subplots(ROW_NUM, COL_NUM, figsize=(12,12))

for i, (label, col) in enumerate(nat_cat_sz.iteritems()): ❶
    ax = axes[int(i/COL_NUM), i%COL_NUM]
    col = col.order(ascending=False)[:10] ❷
    col.plot(kind='barh', ax=ax)
    ax.set_title(label)

plt.tight_layout() ❸
```

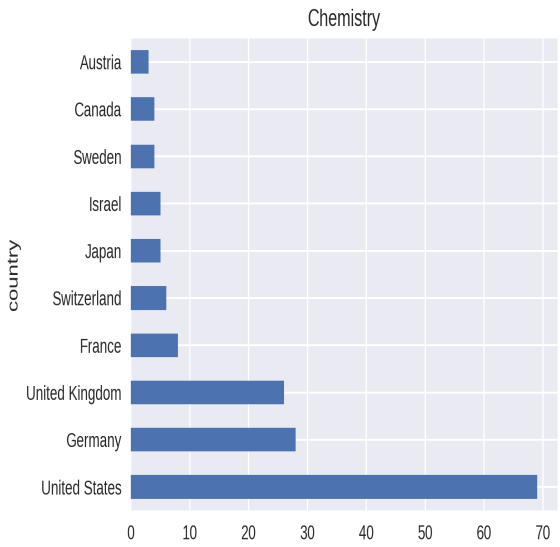
❶ `iteritems` returns an iterator for the `DataFrames` columns in form of (column_label, column) tuples.

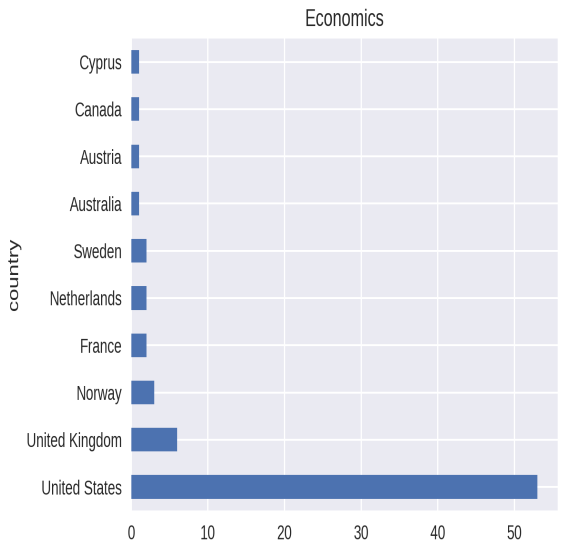
❷ `order` orders the column's `Series` by first making a copy. It is the equivalent of `sort(inplace=False)`.

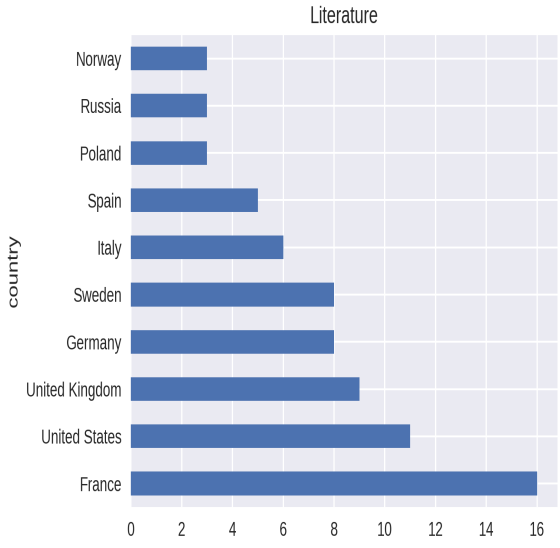
❸ `tight_layout` should prevent label overlaps among the subplots. If you have any problems with `tight_layout`, see the end of "Titles and Axes Labels".

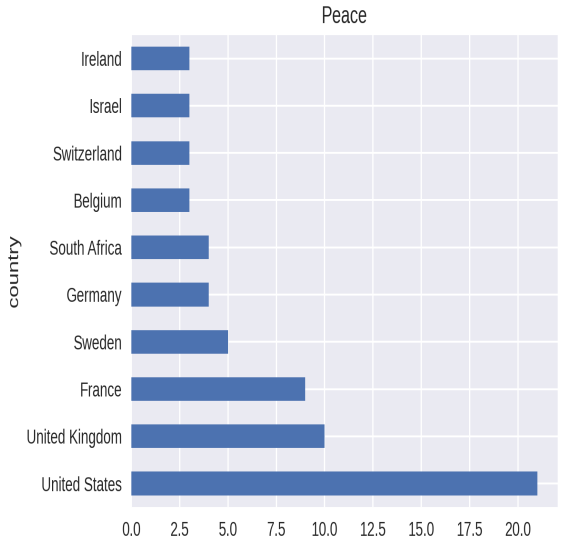
This produces the plots in Figure 4-10.

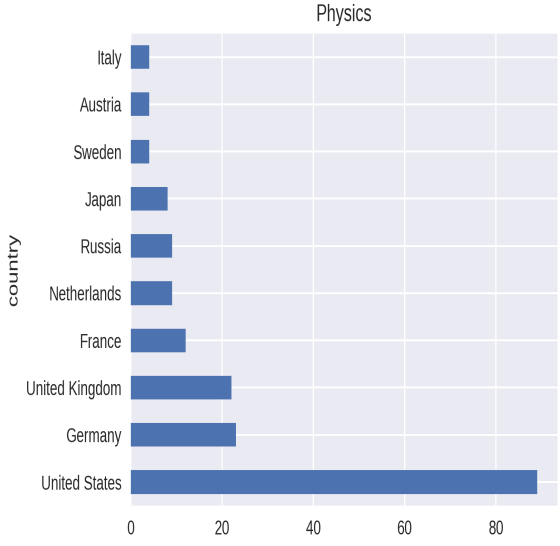
A couple of interesting nuggets from Figure 4-10 are the United States' overwhelming dominance of the Economics prize, reflecting a post-WWII economic consensus, and France's leadership of the Literature prize.


Chemistry
country
Austria
Canada
Sweden
Israel
Japan
Switzerland
France
United Kingdom
Germany
United States
0
10
20
30
40
50
60
70


Economics
country
Cyprus
Canada
Austria
Australia
Sweden
Netherlands
France
Norway
United Kingdom
United States
0
10
20
30
40
50


Literature
country
Norway
Russia
Poland
Spain
Italy
Sweden
Germany
United Kingdom
United States
France
0
2
4
6
8
10
12
14
16


Peace
country
Ireland
Israel
Switzerland
Belgium
South Africa
Germany
Sweden
France
United Kingdom
United States
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Physics
country
Italy
Austria
Sweden
Japan
Russia
Netherlands
France
United Kingdom
Germany
United States
0
20
40
60
80

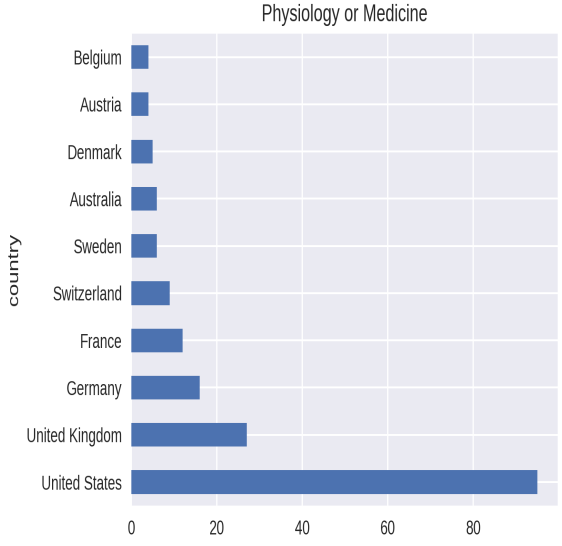

Physiology or Medicine
country
Belgium
Austria
Denmark
Australia
Sweden
Switzerland
France
Germany
United Kingdom
United States
0
20
40
60
80

Figure 4-10. Prizes by country and category

Historical Trends in Prize Distribution

Now that we know the aggregate prize stats by country, are there any interesting historical trends to the prize distribution? Let's explore this with some line plots.

First, let's increase the default font size to 20 points to make the plot labels more legible:

```
plt.rcParams['font.size'] = 20
```

We're going to be looking at prize distribution by year and country, so we'll need a new unstacked `DataFrame` based on these two columns. As previously, we add a `new_index` to give continuous years:

```
new_index = pd.Index(np.arange(1901, 2015), name='year')

by_year_nat_sz = df.groupby(['year', 'country'])\
    .size().unstack().reindex(new_index)
```

The trend we're interested in is the cumulative sum of Nobel Prizes by country over its history. We can further explore trends in individual categories, but for now we'll look at the total for all. Pandas has a handy `cumsum` method for just this. Let's take the United States column and plot it:

```
by_year_nat_sz['United States'].cumsum().plot()
```

This produces the chart in Figure 4-11.

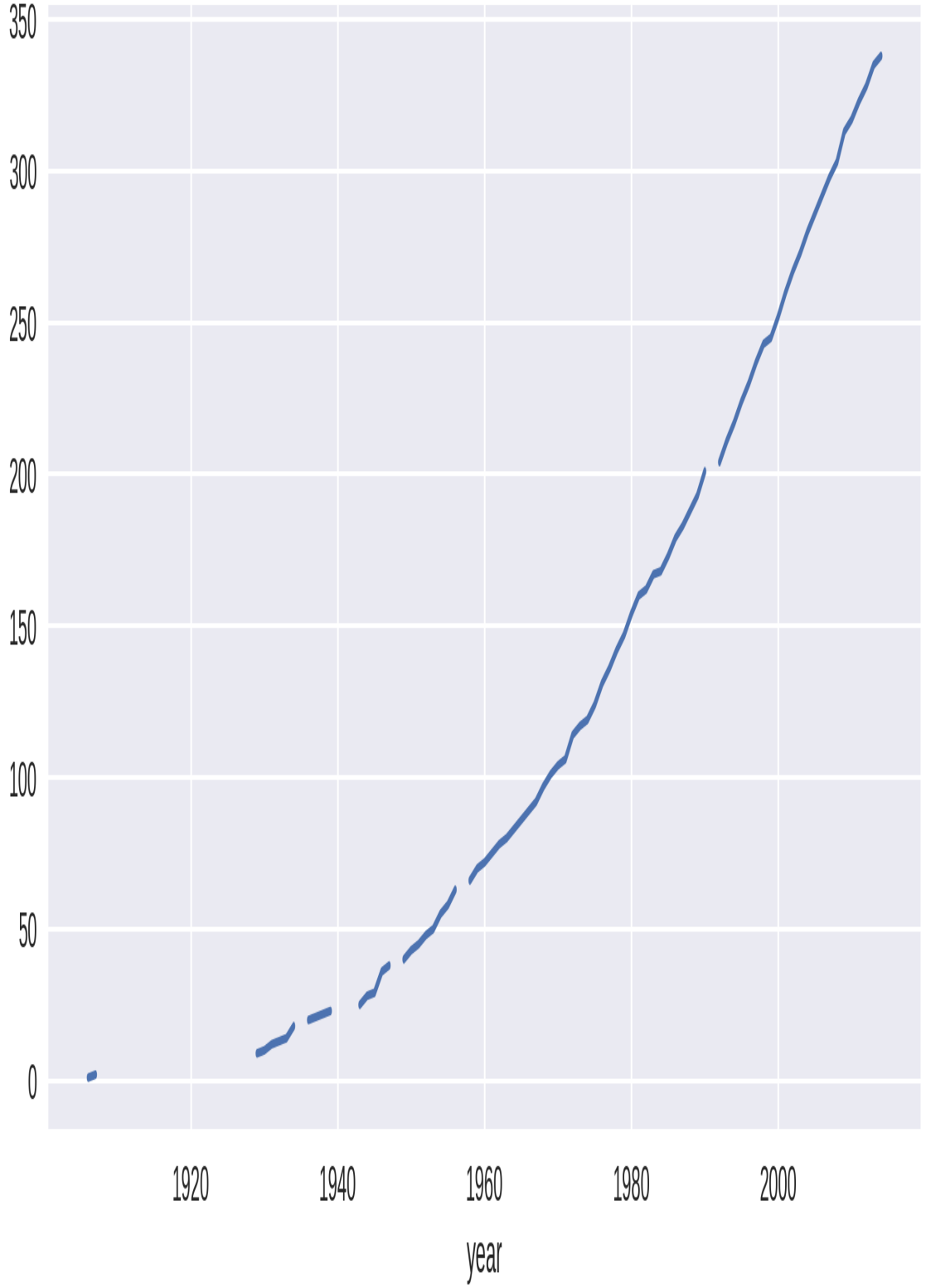

350
300
250
200
150
100
50
0
1920
1940
1960
1980
2000
year

Figure 4-11. Cumulative sum of US prize winners over time

The gaps in the line plot are where the fields are `NaN`, years when the US won no prizes. The `cumsum` algorithm returns `NaN` here. Let's fill those in with a zero to remove the gaps:

```
by_year_nat_sz['United States'].fillna(0)
    .cumsum().plot()
```

This produces the cleaner chart shown in Figure 4-12.

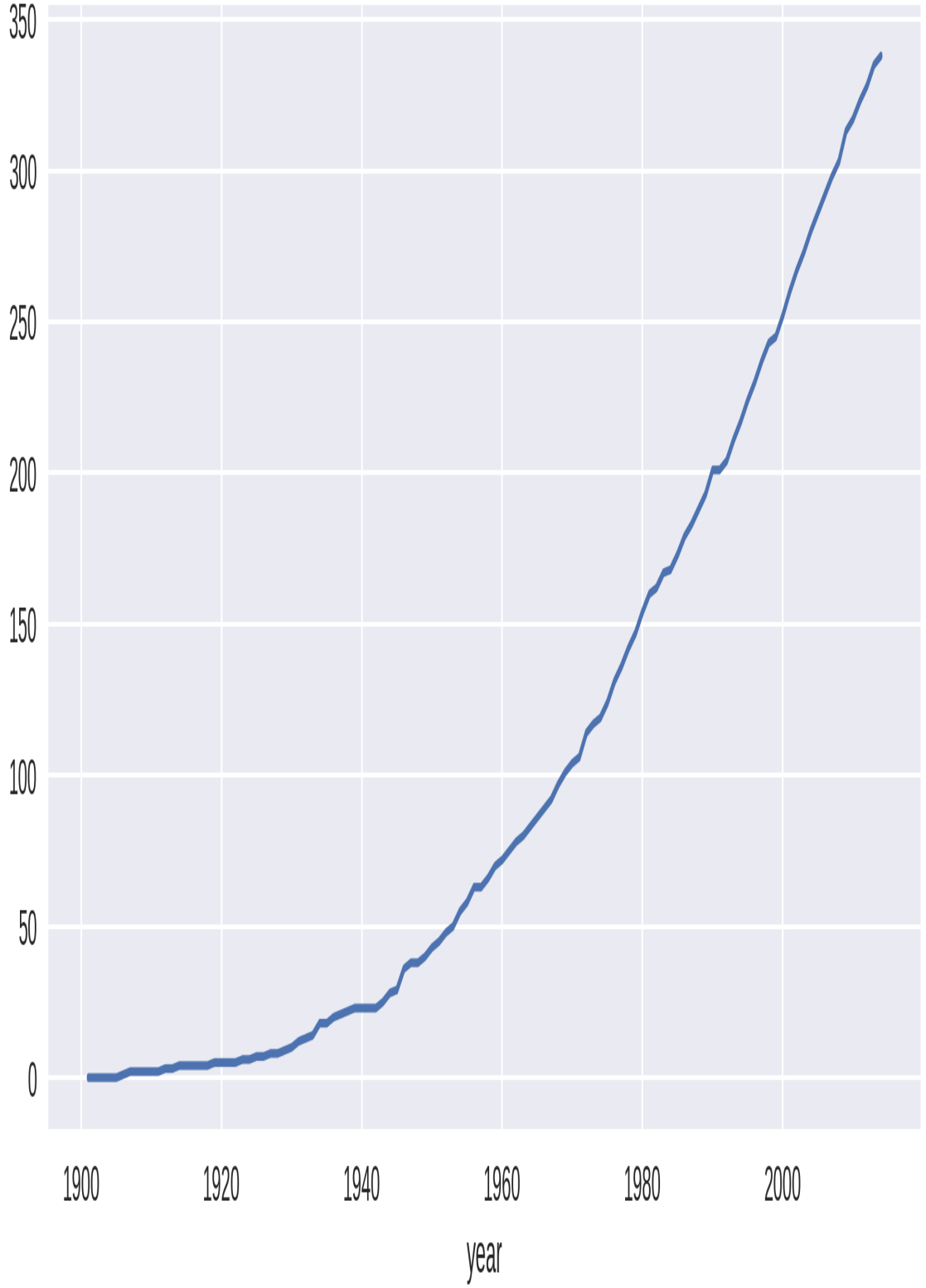

350
300
250
200
150
100
50
0
1900
1920
1940
1960
1980
2000
year

Figure 4-12. Cumulative sum of US prize winners over time

Let's compare the US prize rate with that of the rest of the world:

```
by_year_nat_sz = df.groupby(['year', 'country'])
    .size().unstack().fillna(0)

not_US = by_year_nat_sz.columns.tolist() ❶
not_US.remove('United States')

by_year_nat_sz['Not US'] = by_year_nat_sz[not_US].sum(axis=1) ❷
ax = by_year_nat_sz[['United States', 'Not US']]\
    .cumsum().plot()
```

❶ Gets the list of country column names and removes United States.

❷ Uses our list of non-US country names to create a `'Not_US'` column, the sum of all the prizes for countries in the `not_US` list.

This code produces the chart shown in Figure 4-13.

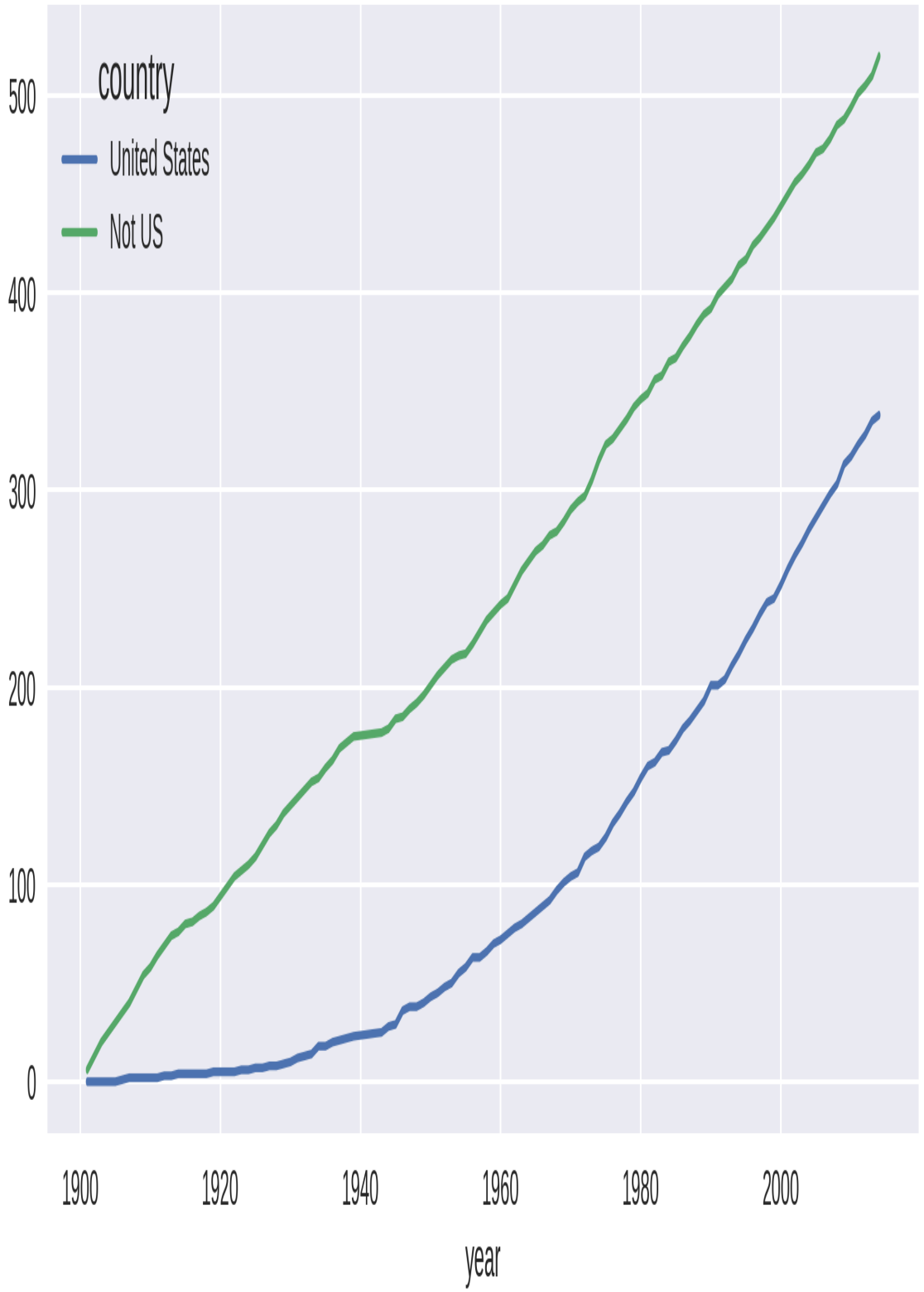

country
United States
Not US
500
400
300
200
100
0
1900
1920
1940
1960
1980
2000
year

Figure 4-13. United States versus rest of world prize hauls

Where the `'Not_US'` haul shows a steady increase over the years of the prize, the US shows a rapid increase around the end of World War II. Let's investigate that further, looking at regional differences. We'll focus on the two or three largest winners for North America, Europe, and Asia:

```
by_year_nat_sz = df.groupby(['year', 'country'])\
    .size().unstack().reindex(new_index).fillna(0)

regions = [ ❶
    {'label':'N. America',
      'countries':['United States', 'Canada']},
    {'label':'Europe',
     'countries':['United Kingdom', 'Germany', 'France']},
    {'label':'Asia',
     'countries':['Japan', 'Russia', 'India']}
]

for region in regions: ❷
    by_year_nat_sz[region['label']] =\
        by_year_nat_sz[region['countries']].sum(axis=1)

by_year_nat_sz[[r['label'] for r in regions]].cumsum()\
  .plot() ❸
```

❶ Our continental country list created by selecting the biggest two or three winners in the three continents compared.

❷ Creates a new column with a region label for each `dict` in the `regions` list, summing its `countries` members.

❸ Plots the cumulative sum of all the new region columns.

This gives us the plot in Figure 4-14. The rate of Asia's prize haul has increased slightly over the years, but the main point of note is North America's huge increase in prizes around the mid-1940s, overtaking a declining Europe in total prizes around the mid-1980s.

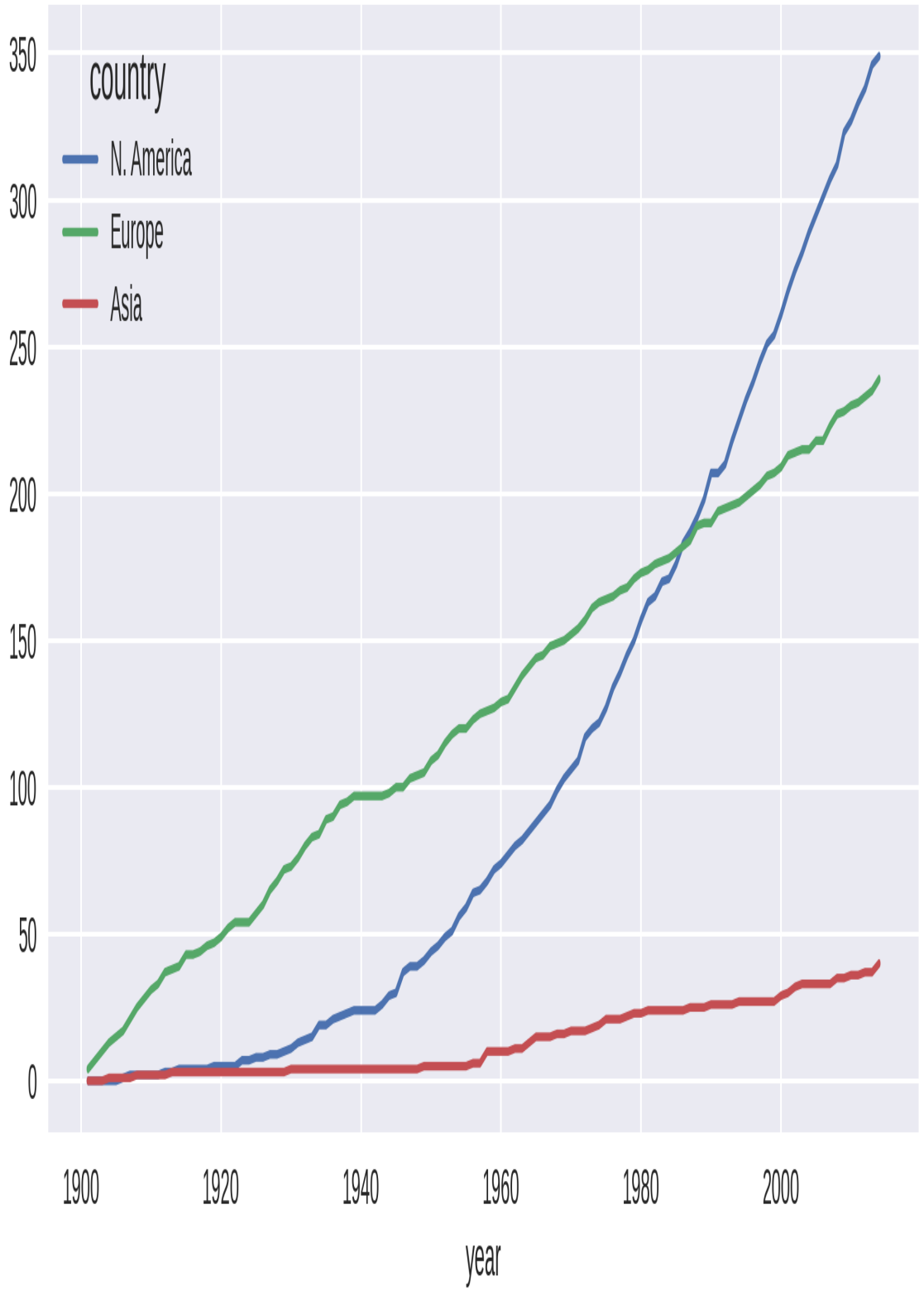

country
N. America
Europe
Asia
350
300
250
200
150
100
50
0
1900
1920
1940
1960
1980
2000
year

Figure 4-14. Historical prize trends by region

Let's improve the resolution of the previous national plots by summarizing the prize rates for the 16 biggest winners, excluding the outlying United States:

```
COL_NUM = 4
ROW_NUM = 4

by_nat_sz = df.groupby('country').size()
by_nat_sz.sort_values(ascending=False,\
    inplace=True)  ❶

fig, axes = plt.subplots(COL_NUM, ROW_NUM,\ ❷
    sharex=True, sharey=True, ❷
    figsize=(12,12))

for i, nat in enumerate(by_nat.index[1:17]): ❸
    ax = axes[i/COL_NUM, i%ROW_NUM]
    by_year_nat_sz[nat].cumsum().plot(ax=ax) ❹
    ax.set_title(nat)
```

❶ Sorts our country groups from highest to lowest win hauls.

❷ Gets a 4×4 grid of axes with shared x- and y-axes for normalized comparison.

❸ Enumerates over the sorted index from second row (1), excluding the US (0).

❹ Selects the `nat` country name column and plots its cumulative sum of prizes on the grid axis `ax`.

This produces Figure 4-15, which shows some nations like Japan, Australia, and Israel on the rise historically, while others flatten off.

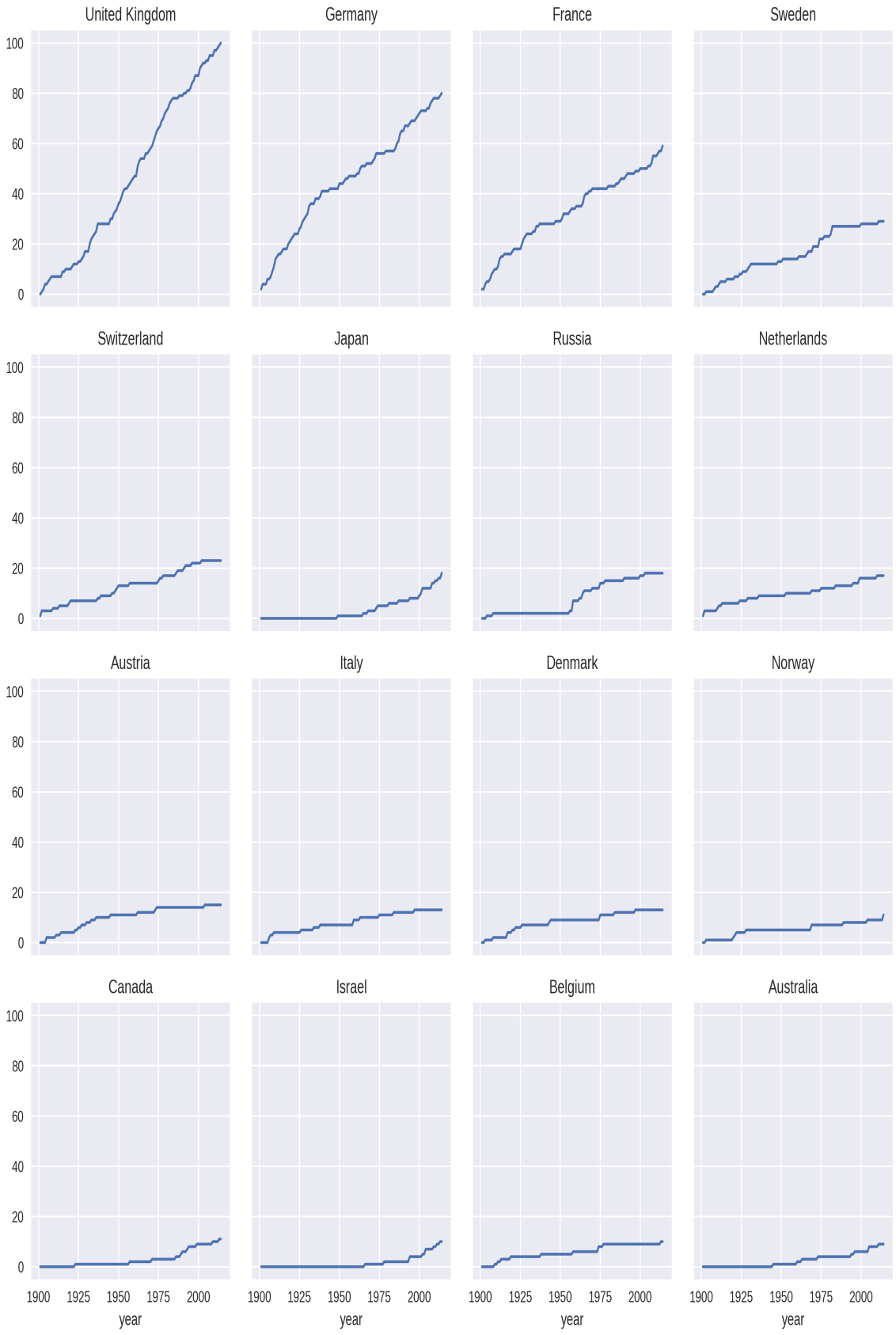

United Kingdom
Germany
France
Sweden
Switzerland
Japan
Russia
Netherlands
Austria
Italy
Denmark
Norway
Canada
Israel
Belgium
Australia
0
20
40
60
80
100
1900
1925
1950
1975
2000
year

Figure 4-15. Prize rates for the 16 largest national winners after the US

Another good way to summarize national prize rates over time is by using a heatmap and dividing the totals by decade. The Seaborn library provides a good heatmap. Let's import it and use its `set` method to increase the font size of its labels by scaling them:

```
import seaborn as sns

sns.set(font_scale = 1.3)
```

The division of data into chunks is also known as *binning*, as it creates *bins* of data. Pandas has a handy `cut` method for just this job, taking a column of continuous values—in our case, Nobel Prize years—and returning ranges of a specified size. You can supply the `DataFrame`'s `groupby` method with the result of `cut` and it will group by the range of indexed values. The following code produces Figure 4-16.

```
bins = np.arange(df.year.min(), df.year.max(), 10) ❶

by_year_nat_binned = df.groupby('country',\
    [pd.cut(df.year, bins, precision=0)])\ ❷
    .size().unstack().fillna(0)

plt.figure(figsize=(8, 8))

sns.heatmap(\
  by_year_nat_binned[by_year_nat_binned.sum(axis=1) > 2],\ ❸
  cmap='rocket_r') ❹
```

❶ Gets our bin ranges for the decades from 1901 (1901, 1911, 1921…).

❷ Cuts our Nobel Prize years into decades using the `bins` ranges with `precision` set to `0`, to give integer years.

❸ Before heatmapping, we filter for those countries with over two Nobel Prizes.

❹ We use the continuous *rocket_r* heatmap to highlight the differences. Check out all the Pandas color palettes here.

Figure 4-16 captures some interesting trends, such as Russia's brief flourishing in the 1950s, which petered out around the 1980s.

Now that we've investigated the Nobel Prize nations, let's turn our attention to the individual winners. Are there any interesting things we can discover about them using the data at hand?

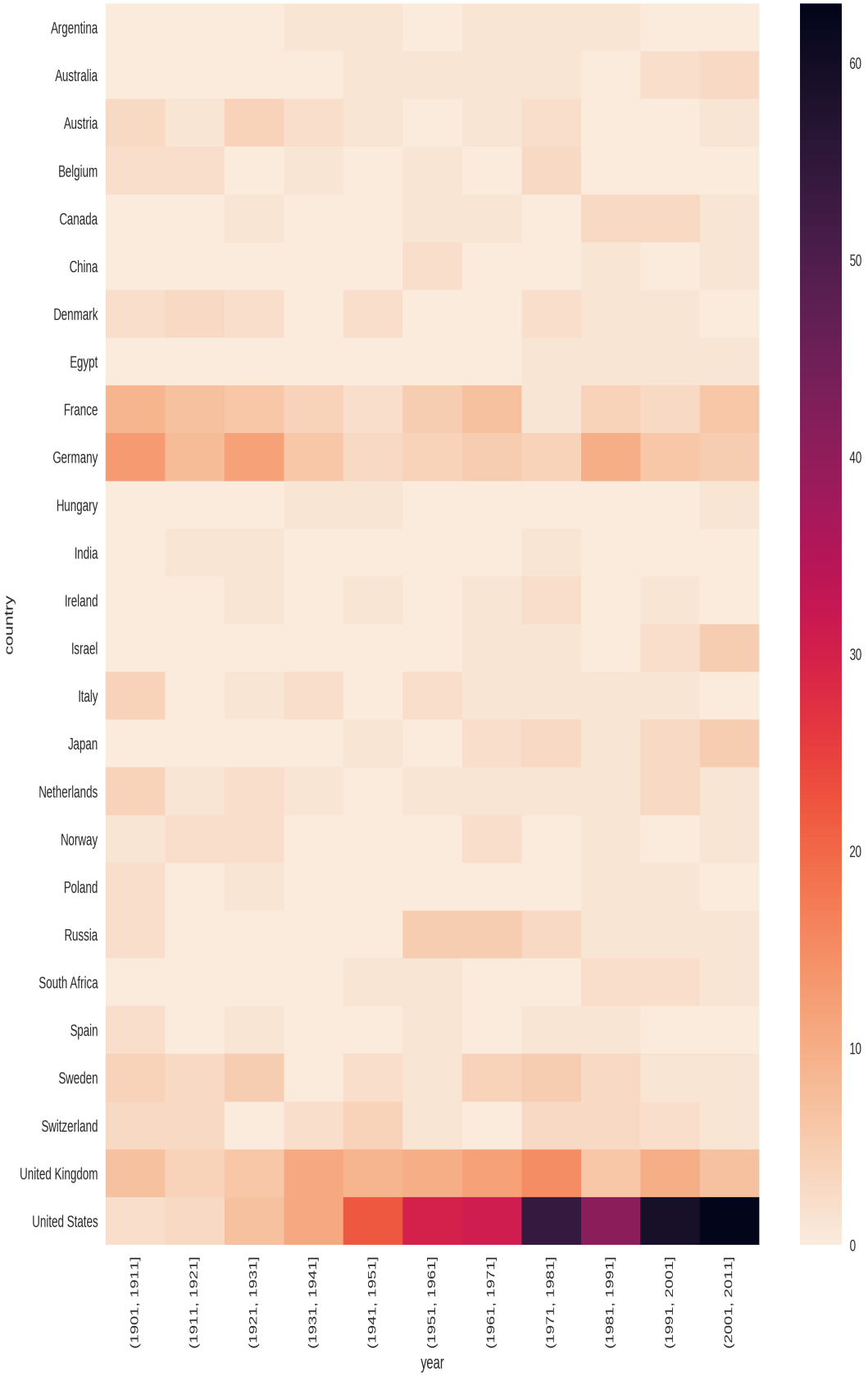

Argentina
Australia
Austria
Belgium
Canada
China
Denmark
Egypt
France
Germany
Hungary
India
Ireland
Israel
Italy
Japan
Netherlands
Norway
Poland
Russia
South Africa
Spain
Sweden
Switzerland
United Kingdom
United States
country
(1901, 1911]
(1911, 1921]
(1921, 1931]
(1931, 1941]
(1941, 1951]
(1951, 1961]
(1961, 1971]
(1971, 1981]
(1981, 1991]
(1991, 2001]
(2001, 2011]
year
60
50
40
30
20
10
0

Figure 4-16. Nations' Nobel Prize hauls by decade

Age and Life Expectancy of Winners

We have the date of birth for all our winners and the date of death for 559 of them. Combined with the year in which they won their prizes, we have a fair amount of individual data to mine. Let's investigate the age distribution of winners and try to glean some idea of the winners' longevity.

Age at Time of Award

In [Link to Come] we added an `'award_age'` column to our Nobel Prize dataset by subtracting the winners' ages from their prize years. A quick and easy win is to use Pandas' histogram plot to assess this distribution:

```
df['award_age'].hist(bins=20)
```

Here we require that the age data be divided into 20 bins. This produces Figure 4-17, showing that the early 60s is a sweet spot for the prize and if you haven't achieved it by 100, it probably isn't going to happen. Note the outlier around 20, which is the recently awarded 17-year-old recipient of the Peace Prize, Malala Yousafzai.

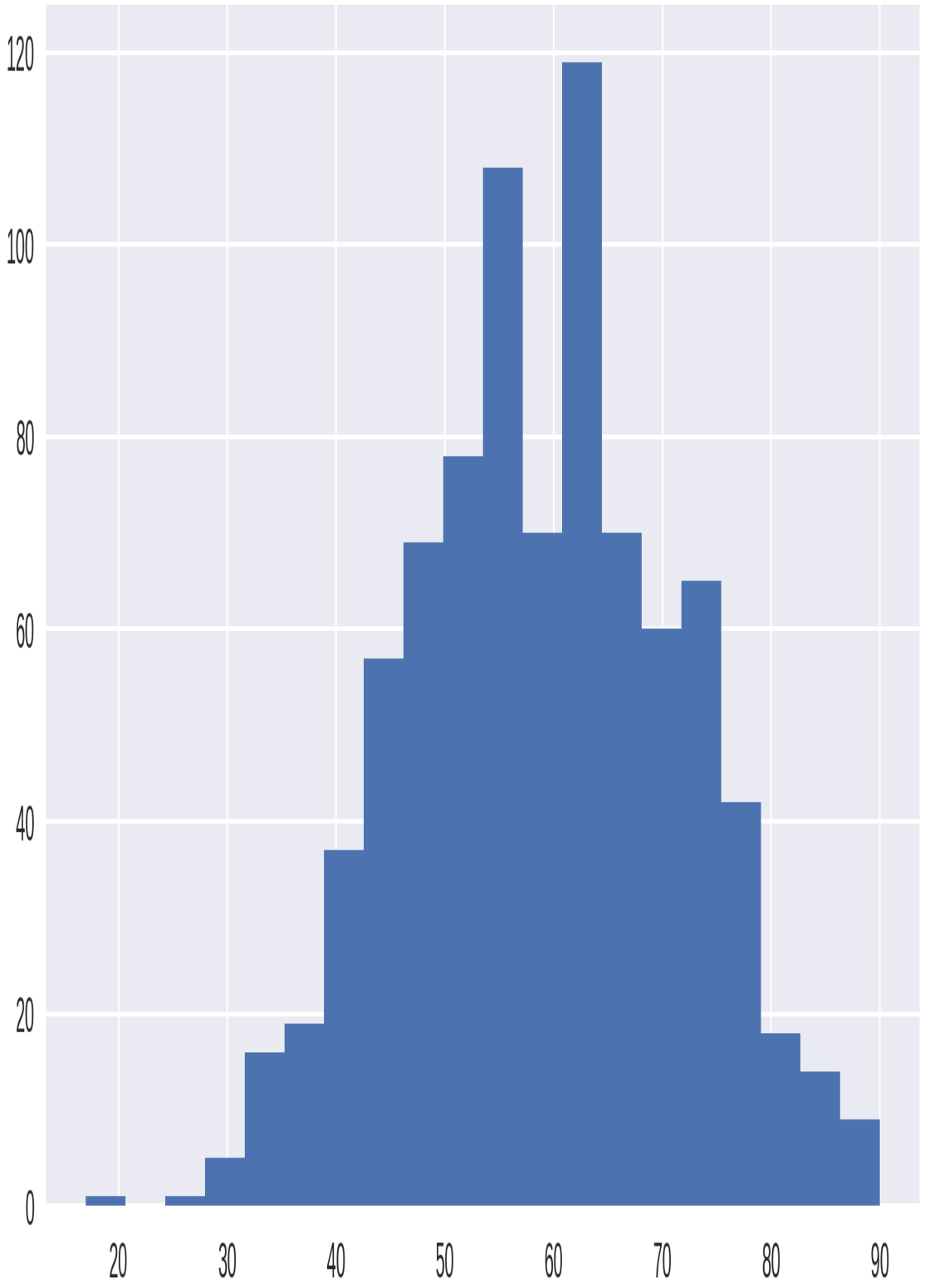

120
100
80
60
40
20
0
20
30
40
50
60
70
80
90

Figure 4-17. Distribution of ages at time of award

We can use Seaborn's `displot` to get a better feel for the distribution, adding a kernel density estimate (KDE)[2] to the histogram. The following one-liner produces Figure 4-18, showing that our sweet spot is around 60 years of age:

```
sns.displot(df['award_age'], kde=True, height=4, aspect=2)
```

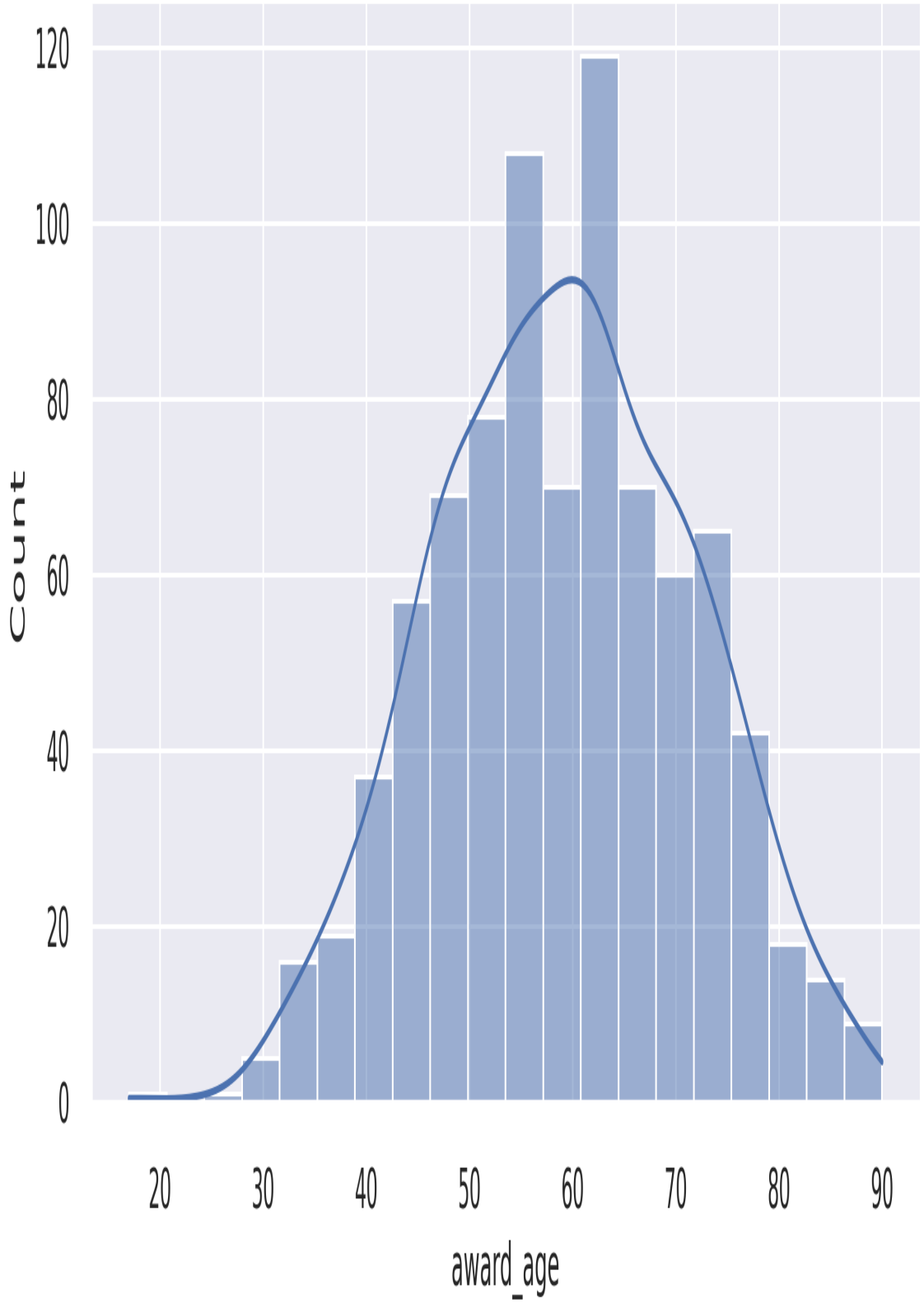

120
100
80
60
40
20
0
Count
20
30
40
50
60
70
80
90
award_age

Figure 4-18. Distribution of ages at time of award with KDE superimposed

A box plot is a good way of visualizing continuous data, showing the quartiles, the first and third marking the edges of the box and the second quartile (or median average) marking the line in the box. Generally, as in Figure 4-19, the horizontal end lines (known as the whisker ends) indicate the max and min of the data. Let's use a Seaborn box plot and divide the prizes by gender:

```
sns.boxplot(df, x='gender', y='award_age')
```

This produces Figure 4-19, which shows that the distributions by gender are similar, with women having a slightly lower average age. Note that with far fewer female prize winners, their statistics are subject to a good deal more uncertainty.

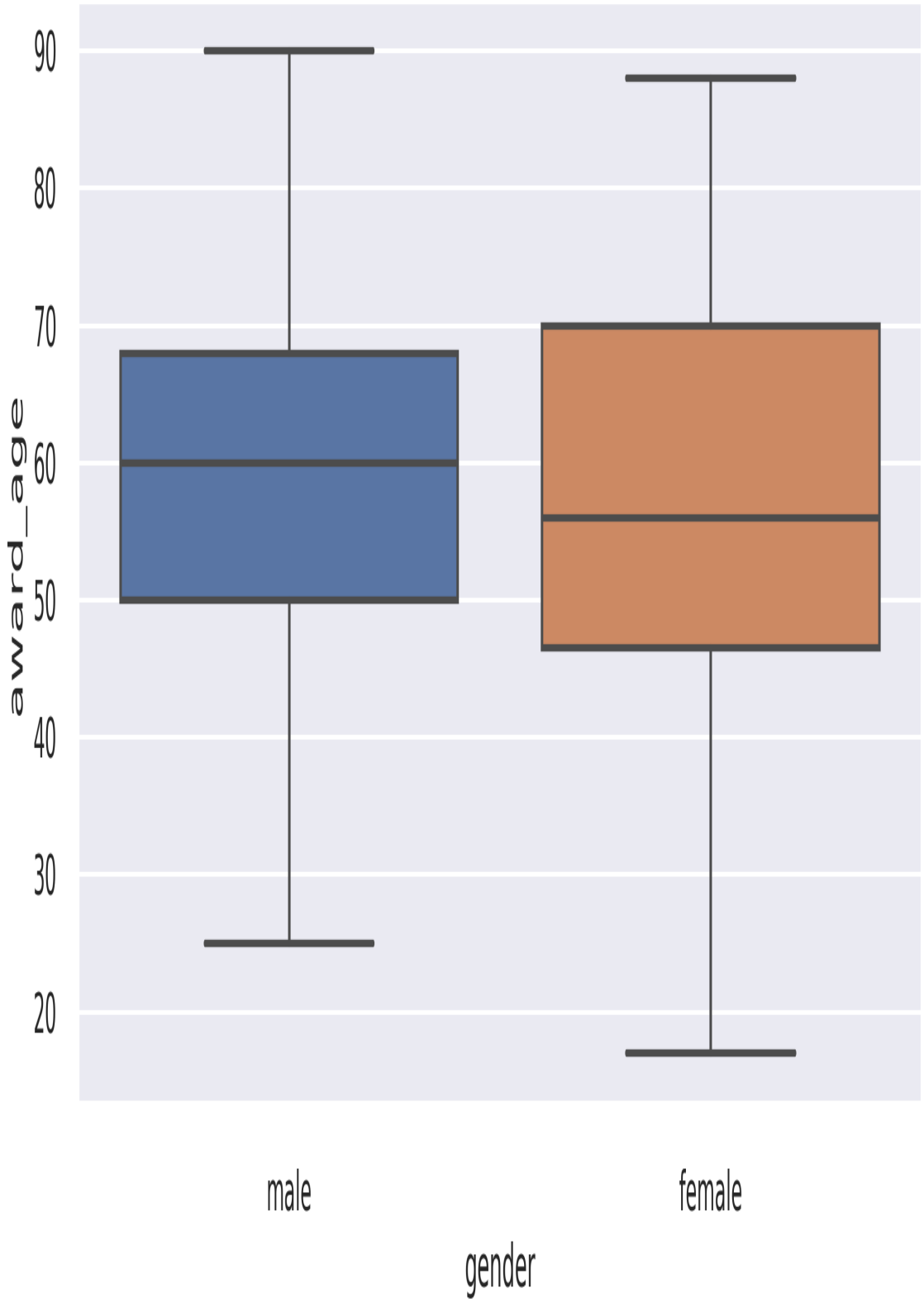

90
80
70
60
50
40
30
20
award_age
male
female
gender

Figure 4-19. Ages of prize winners by gender

Seaborn's rather nice violinplot combines the conventional box plot with a kernel density estimation to give a more refined view of the breakdown by age and gender. The following code produces Figure 4-20.

```
sns.violinplot(data=df, x='gender', y='award_age')
```

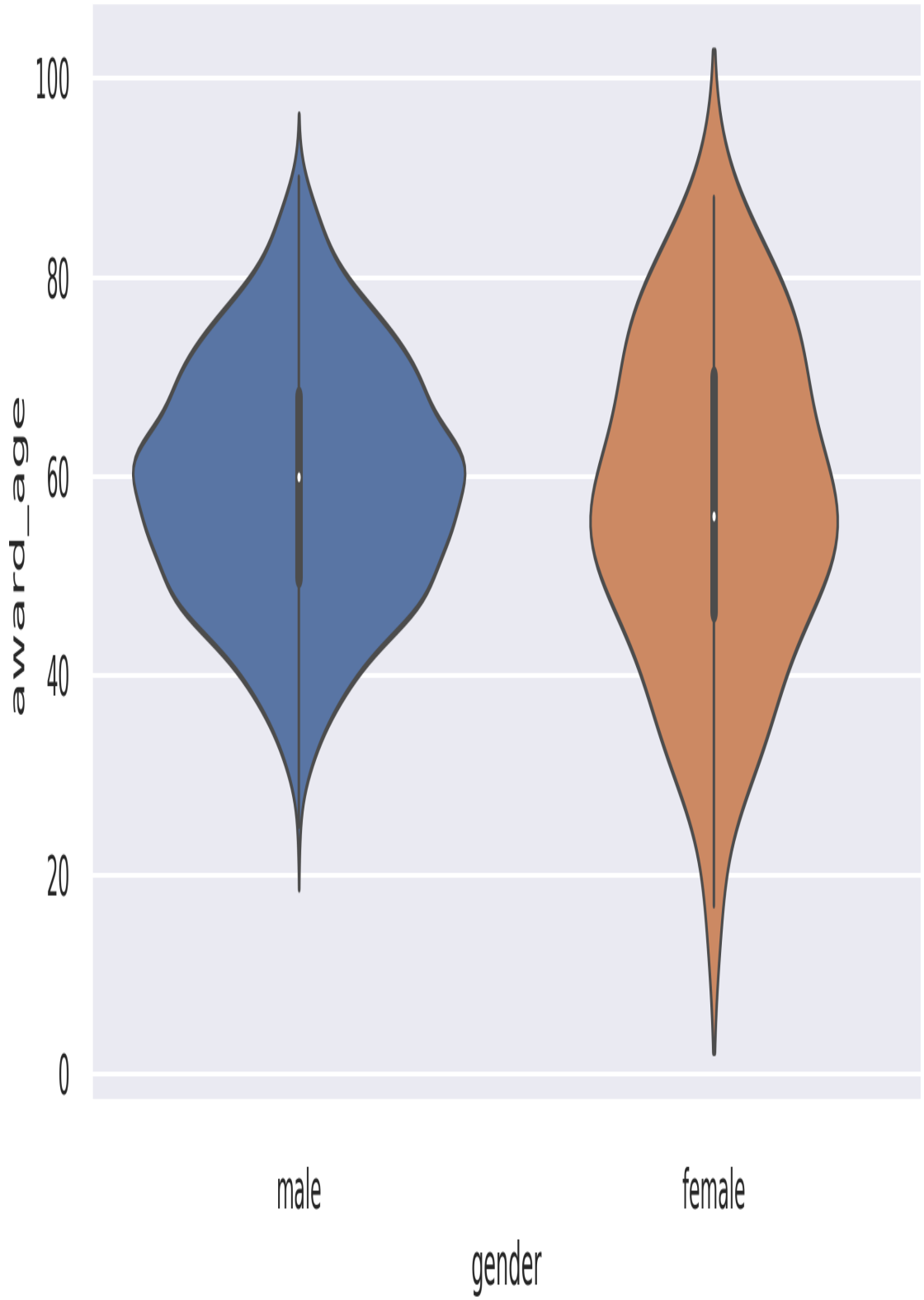

100
80
60
40
20
0
award_age
male
female
gender

Figure 4-20. Violinplots of prize-age distribution by gender

Life Expectancy of Winners

Now let's look at the longevity of Nobel Prize winners, by subtracting the available dates of death from their respective dates of birth. We'll store this data in a new `'age_at_death'` column:

```
df['age_at_death'] = (df.date_of_death - df.date_of_birth)\
                     .dt.days/365 ❶
```

❶ `datetime64` data can be added and subtracted in a sensible fashion, producing a Pandas `timedelta` column. We can use its `dt` method to get the interval in days, dividing this by 365 to get the age at death as a float.

We make a copy of the `'age_at_death'` column,[3] removing all empty `NaN` rows. This can then be used to make the histogram and KDE shown in Figure 4-21.

```
age_at_death = df[df.age_at_death.notnull()].age_at_death ❶

sns.displot(age_at_death, bins=40, kde=True, aspect=2, height=4)
```

❶ Removes all `NaN`s to clean the data and reduce plotting errors (e.g., `distplot` fails with `NaN`s).

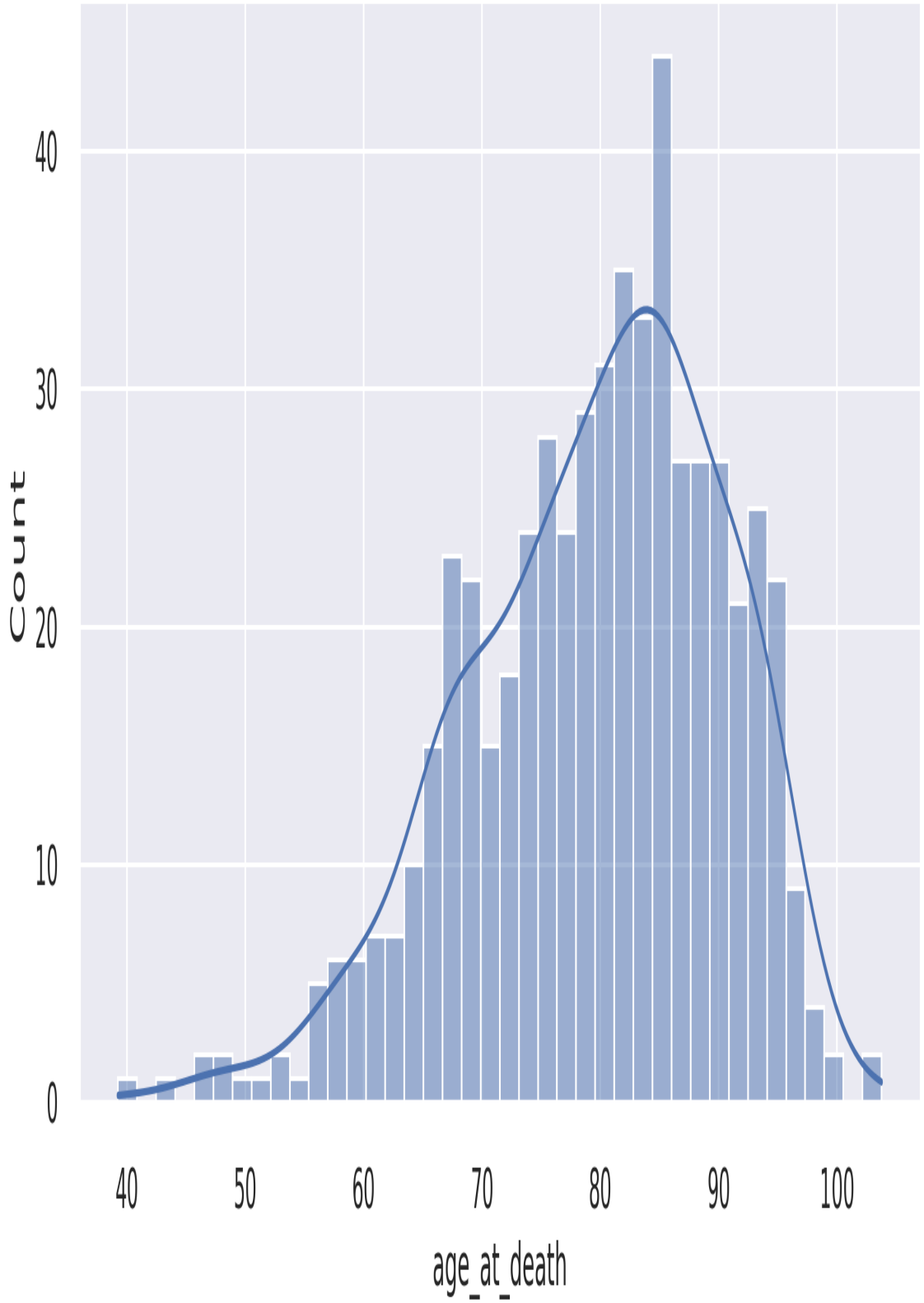

Count
0
10
20
30
40
40
50
60
70
80
90
100
age_at_death

Figure 4-21. Life expectancy of the Nobel Prize winners

Figure 4-21 shows the Nobel Prize winners to be a remarkably long-lived bunch, with an average age in the early 80s. This is all the more impressive given that the large majority of winners are men, who have considerably lower average life expectancies in the general population than women. One contributary factor to this longevity is the selection bias we saw earlier. Nobel Prize winners aren't generally honored until they're in their late 50s and 60s, which removes the subpopulation who died before having the chance to be acknowledged, pushing up the longevity figures.

Figure 4-21 shows some centenarians among the prize winners. Let's find them:

```
df[df.age_at_death > 100][['name', 'category', 'year']]
Out:
                      name                 category  year
101          Ronald Coase                Economics  1991
328   Rita Levi-Montalcini  Physiology or Medicine 1986
```

Now let's superimpose a couple of KDEs to show differences in mortality for male and female recipients:

```
df_temp = df_temp[df.age_at_death.notnull()] ❶
sns.kdeplot(df_temp[df_temp.gender == 'male']
    .age_at_death, shade=True, label='male')
sns.kdeplot(df_temp[df_temp.gender == 'female']
    .age_at_death, shade=True, label='female')

plt.legend()
```

❶ Creates a `DataFrame` with only valid `'age_at_death'` fields.

This produces Figure 4-22, which, allowing for the small number of female winners and flatter distribution, shows the male and female averages to be close. Female Nobel Prize winners seem to live relatively shorter lives than their counterparts in the general population.

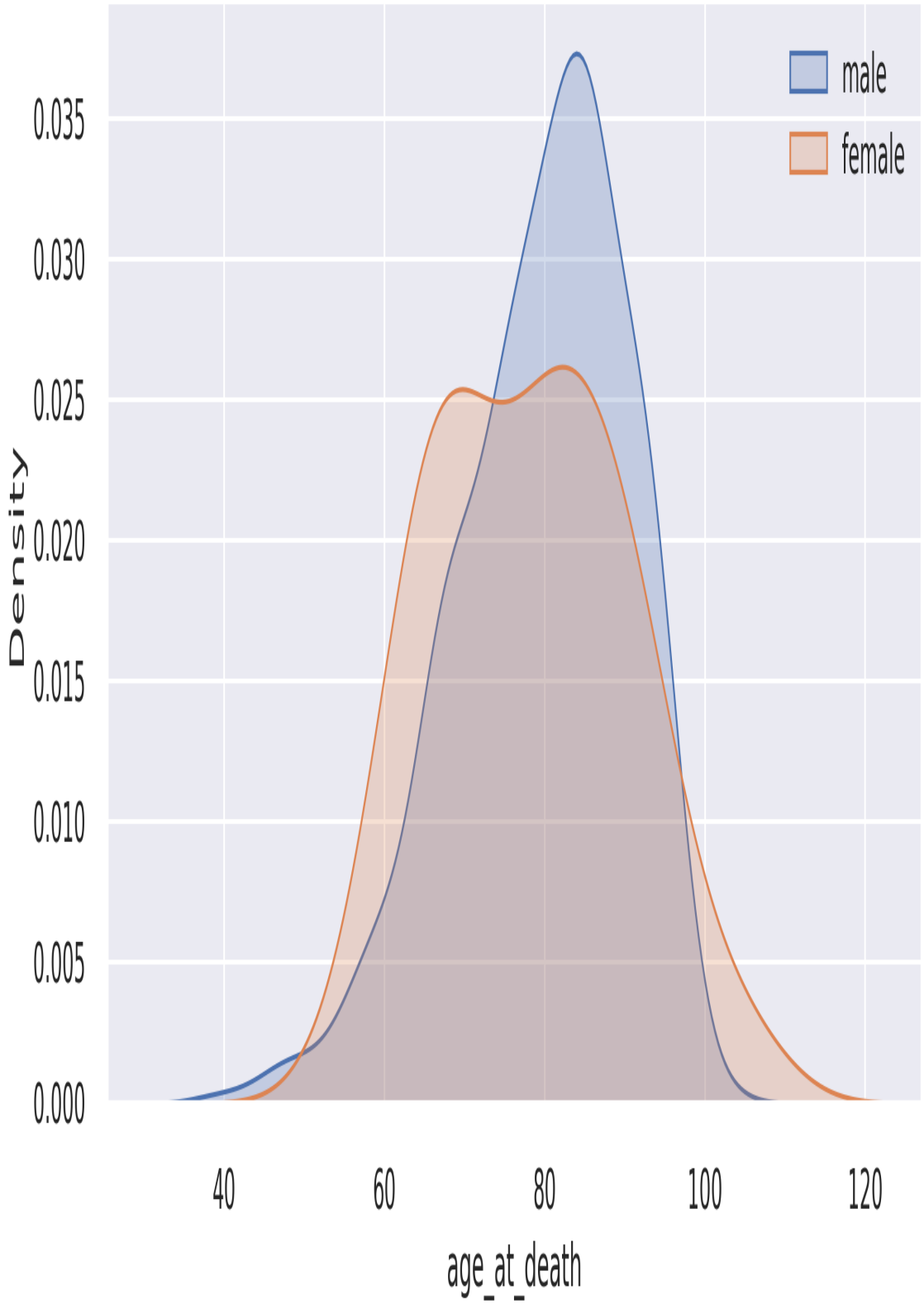

male
female
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Density
40
60
80
100
120
age_at_death

Figure 4-22. Nobel Prize winner life expectancies by gender

A violinplot provides another perspective, shown in Figure 4-23.

```
sns.violinplot(data=df, x='gender', y='age_at_death',\
               aspect=2, height=4)
```

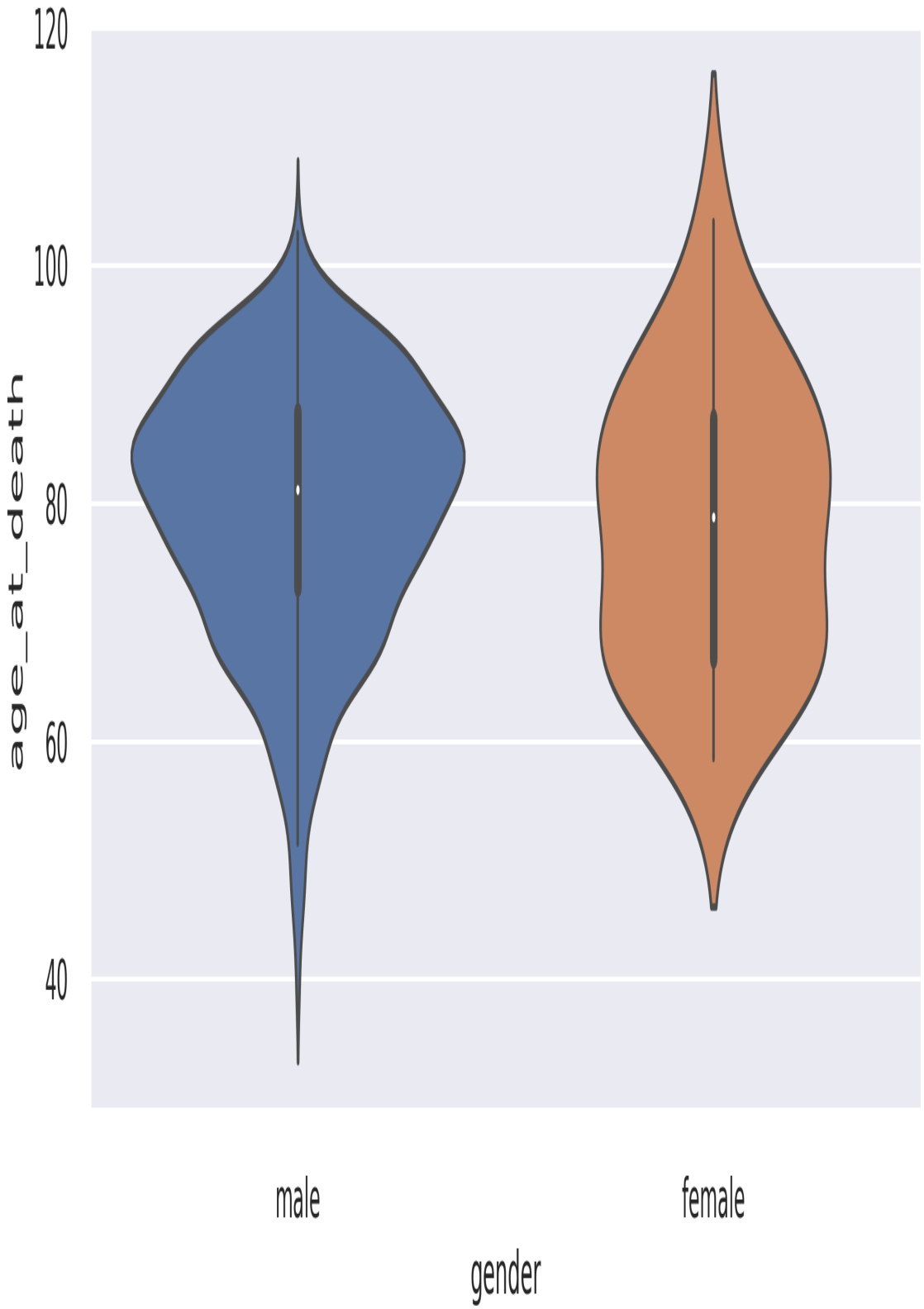

120
100
80
60
40
age_at_death
male
female
gender

Figure 4-23. Winner life expectancies by gender

Increasing Life Expectancies over Time

Let's do a little historical demographic analysis by seeing if there's a correlation between the date of birth of our Nobel Prize winners and their life expectancy. We'll use one of Seaborn's `lmplots` to provide a scatter plot and line-fitting with confidence intervals (see "Seaborn").

```
df_temp = df[df.age_at_death.notnull()] ❶
data = pd.DataFrame( ❷
    {'age at death':df_temp.age_at_death,
     'date of birth':df_temp.date_of_birth.dt.year})
sns.lmplot(data=data, x='date of birth', y='age at death',
  height=6, aspect=1.5)
```

❶ Creates a temporary `DataFrame`, removing all the rows with no `'age_at_death'` field.

❷ Creates a new `DataFrame` with only the two columns of interest from the refined `df_temp`. We grab only the year from the `date_of_birth`, using its `dt` accessor.

This produces Figure 4-24, showing an increase in life expectancy of a decade or so over the prize's duration.


100
90
80
70
60
50
40
age at death
1820
1840
1860
1880
1900
1920
1940
date of birth

Figure 4-24. Correlating date of birth with age at death

The Nobel Diaspora

While cleaning our Nobel Prize dataset in [Link to Come], we found duplicate entries recording the winner's place of birth and country at time of winning. We preserved these, giving us 104 winners whose country at time of winning was different from their country of birth. Is there a story to tell here?

A good way to visualize the movement patterns from the winners' country of birth to their adopted country is by using a heatmap to show all `born_in`/`country` pairs. The following code produces the heatmap in Figure 4-25:

```
by_bornin_nat = df[df.born_in.notnull()].groupby(\ ❶
    ['born_in', 'country']).size().unstack()
by_bornin_nat.index.name = 'Born in' ❷
by_bornin_nat.columns.name = 'Moved to'
plt.figure(figsize=(12, 12))

ax = sns.heatmap(by_bornin_nat, vmin=0, vmax=8, cmap="crest",\ ❸
                 linewidth=0.5)
ax.set_title('The Nobel Diaspora')
```

❶ Selects all rows with a `'born_in'` field, and forms groups on this and the country column.

❷ We rename the row index and column names to make them more descriptive.

❸ Seaborn's `heatmap` attempts to set the correct bounds for the data, but in this case, we must manually adjust the limits (`vmin` and `vmax`) to see all the cells.

Figure 4-25 shows some interesting patterns, which tell a tale of persecution and sanctuary. First, the United States is the overwhelming recipient of relocated Nobel winners, followed by the United Kingdom. Note that the biggest contingents for both (except cross-border traffic from Canada) are from Germany. Italy, Hungary, and Austria are the next largest groups. Examining the individuals in these groups shows that the majority were displaced as a result of the rise of antisemitic fascist regimes in the run-up to World War II and the increasing persecution of Jewish minorities.

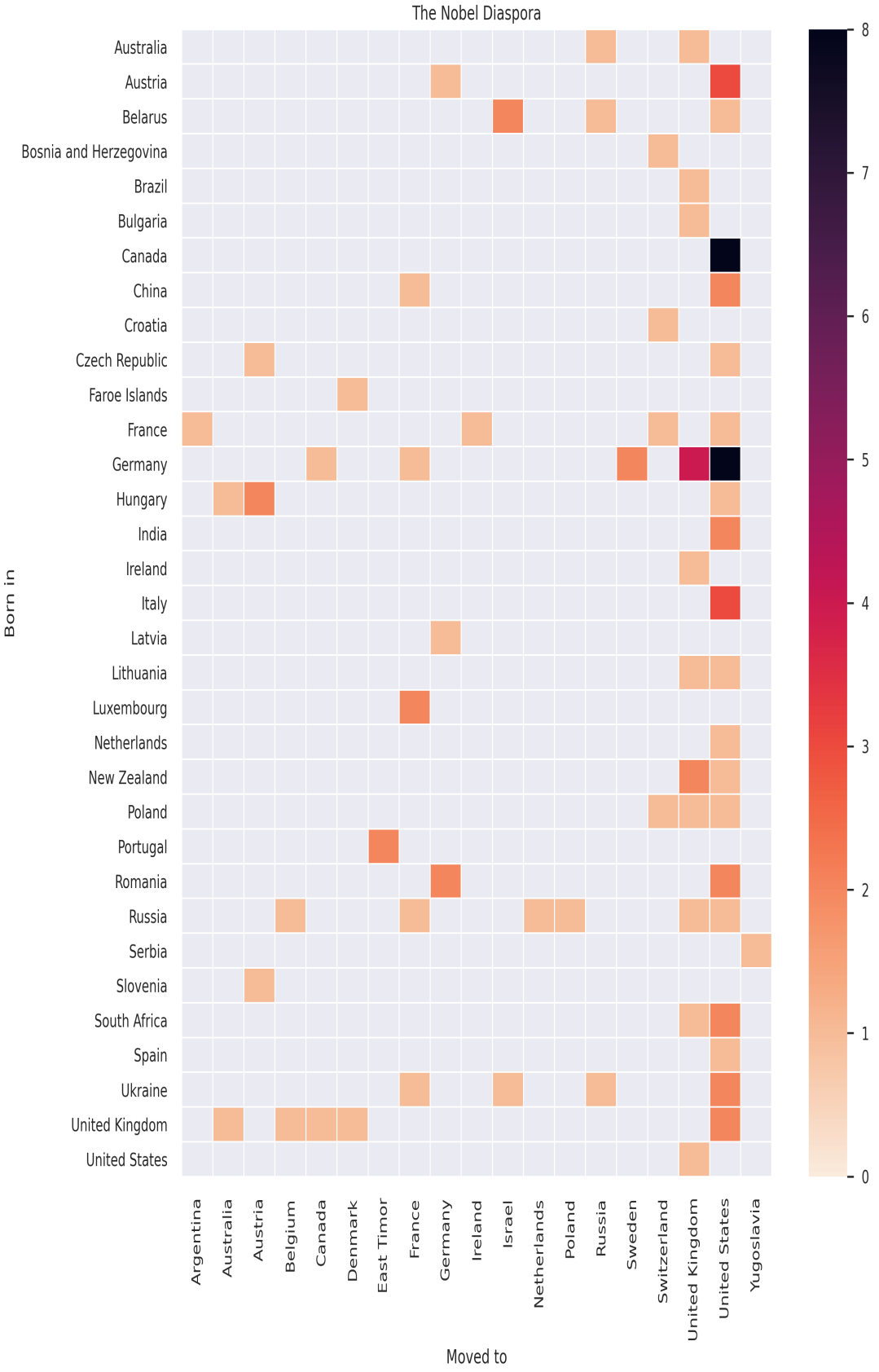

The Nobel Diaspora
Born in
Australia
Austria
Belarus
Bosnia and Herzegovina
Brazil
Bulgaria
Canada
China
Croatia
Czech Republic
Faroe Islands
France
Germany
Hungary
India
Ireland
Italy
Latvia
Lithuania
Luxembourg
Netherlands
New Zealand
Poland
Portugal
Romania
Russia
Serbia
Slovenia
South Africa
Spain
Ukraine
United Kingdom
United States
Argentina
Australia
Austria
Belgium
Canada
Denmark
East Timor
France
Germany
Ireland
Israel
Netherlands
Poland
Russia
Sweden
Switzerland
United Kingdom
United States
Yugoslavia
Moved to
0
1
2
3
4
5
6
7
8

Figure 4-25. The Nobel Prize diaspora

To take an example, all four of the Nobel winners who moved from Germany to the United Kingdom were German research scientists with Jewish ancestry who moved in response to the Nazis' rise to power:

```
df[(df.born_in == 'Germany') & (df.country == 'United Kingdom')]
    [['name', 'date_of_birth', 'category']]
```

```
Out:
                    name date_of_birth                category
119  Ernst Boris Chain    1906-06-19  Physiology or Medicine
484   Hans Adolf Krebs    1900-08-25  Physiology or Medicine
486           Max Born    1882-12-11                 Physics
503       Bernard Katz    1911-03-26  Physiology or Medicine
```

Ernst Chain pioneered the industrial production of penicillin. Hans Krebs discovered the Krebs cycle, one of the most important discoveries in biochemistry, which regulates the energy production of cells. Max Born was one of the pioneers of quantum mechanics, and Bernard Katz uncovered the fundamental properties of synaptic junctions in neurons.

There are many such illustrious names among the winning emigrants. One interesting discovery is the number of prize winners who were part of the famous Kindertransport, an organized rescue effort that took place nine months before the outbreak of WWII and saw 10,000 Jewish children from Germany, Austria, Czechoslovakia, and Poland transported to the United Kingdom. Of these children, four went on to win a Nobel Prize.

Summary

In this chapter, we explored our Nobel Prize dataset, probing the key fields of gender, category, country, and year (of prize) looking for interesting trends and stories we can tell or enable visually. We used a fair number of Matplotlib (by way of Pandas) and Seaborn's plots, from basic bar charts to more complicated statistical summaries like violinplots and heatmaps. Mastery of these tools and the others in the Python chart armory will allow

you to quickly get the feel of your datasets, which is a prerequisite to building a visualization around them. We found more than enough stories in the data to suggest a web visualization. In the next chapter we will imagine and design just such a Nobel Prize winner visualization, cherry-picking the nuggets gained in this chapter.

---

1 Anecdotally, no one I have asked in person or in talk audiences has known the name of the *other* female Nobel Prize winner for Physics.

2 See Wikipedia for details. Essentially the data is smoothed and a probability density function derived.

3 We are ignoring leap years and other subtle, complicating factors in deriving years from days.

About the Author

Kyran Dale is a jobbing programmer, ex-research scientist, recreational hacker, independent researcher, occasional entrepreneur, cross-country runner and improving jazz pianist. During 15 odd years as a research scientist he hacked a lot of code, learned a lot of libraries and settled on some favorite tools. These days he finds Python, JavaScript, and a little C++ goes a long way to solving most problems out there. He specializes in fast-prototyping and feasibility studies, with an algorithmic bent but is happy to just build cool things.