CHAPTER 2

A Crash Course in Python

People are still crazy about Python after twenty-five years, which I find hard to believe.

—Michael Palin

All new employees at DataSciencester are required to go through new employee orientation, the most interesting part of which is a crash course in Python.

This is not a comprehensive Python tutorial but instead is intended to highlight the parts of the language that will be most important to us (some of which are often not the focus of Python tutorials).

The Basics

Getting Python

You can download Python from python.org. But if you don't already have Python, I recommend instead installing the Anaconda distribution, which already includes most of the libraries that you need to do data science.

As I write this, the latest version of Python is 3.4. At DataSciencester, however, we use old, reliable Python 2.7. Python 3 is not backward-compatible with Python 2, and many important libraries only work well with 2.7. The data science community is still firmly stuck on 2.7, which means we will be, too. Make sure to get that version.

If you don't get Anaconda, make sure to install pip, which is a Python package manager that allows you to easily install third-party packages (some of which we'll need). It's also worth getting IPython, which is a much nicer Python shell to work with.

(If you installed Anaconda then it should have come with pip and IPython.)

Just run:

```
pip install ipython
```

and then search the Internet for solutions to whatever cryptic error messages that causes.

The Zen of Python

Python has a somewhat Zen description of its design principles, which you can also find inside the Python interpreter itself by typing `import this`.

One of the most discussed of these is:

> There should be one—and preferably only one—obvious way to do it.

Code written in accordance with this "obvious" way (which may not be obvious at all to a newcomer) is often described as "Pythonic." Although this is not a book about Python, we will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Whitespace Formatting

Many languages use curly braces to delimit blocks of code. Python uses indentation:

```
for i in [1, 2, 3, 4, 5]:
    print i                     # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print j                 # first line in "for j" block
        print i + j             # last line in "for j" block
    print i                     # last line in "for i" block
print "done looping"
```

This makes Python code very readable, but it also means that you have to be very careful with your formatting. Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

and for making code easier to read:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

```
two_plus_three = 2 + \
                 3
```

One consequence of whitespace formatting is that it can be hard to copy and paste code into the Python shell. For example, if you tried to paste the code:

```
for i in [1, 2, 3, 4, 5]:

    # notice the blank line
    print i
```

into the ordinary Python shell, you would get a:

```
IndentationError: expected an indented block
```

because the interpreter thinks the blank line signals the end of the `for` loop's block.

IPython has a magic function `%paste`, which correctly pastes whatever is on your clipboard, whitespace and all. This alone is a good reason to use IPython.

Modules

Certain features of Python are not loaded by default. These include both features included as part of the language as well as third-party features that you download yourself. In order to use these features, you'll need to `import` the modules that contain them.

One approach is to simply import the module itself:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Here `re` is the module containing functions and constants for working with regular expressions. After this type of `import` you can only access those functions by prefixing them with `re.`.

If you already had a different `re` in your code you could use an alias:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

You might also do this if your module has an unwieldy name or if you're going to be typing it a lot. For example, when visualizing data with `matplotlib`, a standard convention is:

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

If you need a few specific values from a module, you can import them explicitly and use them without qualification:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

If you were a bad person, you could import the entire contents of a module into your namespace, which might inadvertently overwrite variables you've already defined:

```
match = 10
from re import *    # uh oh, re has a match function
print match         # "<function re.match>"
```

However, since you are not a bad person, you won't ever do this.

Arithmetic

Python 2.7 uses integer division by default, so that `5 / 2` equals `2`. Almost always this is not what we want, so we will always start our files with:

```
from __future__ import division
```

after which `5 / 2` equals `2.5`. Every code example in this book uses this new-style division. In the handful of cases where we need integer division, we can get it with a double slash: `5 // 2`.

Functions

A function is a rule for taking zero or more inputs and returning a corresponding output. In Python, we typically define functions using `def`:

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Python functions are *first-class*, which means that we can assign them to variables and pass them into functions just like any other arguments:

```
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double              # refers to the previously defined function
x = apply_to_one(my_double)     # equals 2
```

It is also easy to create short anonymous functions, or lambdas:

```
y = apply_to_one(lambda x: x + 4)      # equals 5
```

You can assign lambdas to variables, although most people will tell you that you should just use `def` instead:

```
another_double = lambda x: 2 * x       # don't do this
def another_double(x): return 2 * x    # do this instead
```

Function parameters can also be given default arguments, which only need to be specified when you want a value other than the default:

```
def my_print(message="my default message"):
    print message
```

```
my_print("hello")   # prints 'hello'
my_print()          # prints 'my default message'
```

It is sometimes useful to specify arguments by name:

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5)  # returns -5
subtract(b=5)   # same as previous
```

We will be creating many, many functions.

Strings

Strings can be delimited by single or double quotation marks (but the quotes have to match):

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python uses backslashes to encode special characters. For example:

```
tab_string = "\t"       # represents the tab character
len(tab_string)         # is 1
```

If you want backslashes as backslashes (which you might in Windows directory names or in regular expressions), you can create *raw* strings using `r""`:

```
not_tab_string = r"\t"  # represents the characters '\' and 't'
len(not_tab_string)     # is 2
```

You can create multiline strings using triple-[double-]-quotes:

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

Exceptions

When something goes wrong, Python raises an *exception*. Unhandled, these will cause your program to crash. You can handle them using `try` and `except`:

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

Although in many languages exceptions are considered bad, in Python there is no shame in using them to make your code cleaner, and we will occasionally do so.

Lists

Probably the most fundamental data structure in Python is the `list`. A list is simply an ordered collection. (It is similar to what in other languages might be called an array, but with some added functionality.)

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)     # equals 3
list_sum    = sum(integer_list)     # equals 6
```

You can get or set the *n*th element of a list with square brackets:

```
x = range(10)   # is the list [0, 1, ..., 9]
zero = x[0]     # equals 0, lists are 0-indexed
one = x[1]      # equals 1
nine = x[-1]    # equals 9, 'Pythonic' for last element
eight = x[-2]   # equals 8, 'Pythonic' for next-to-last element
x[0] = -1       # now x is [-1, 1, 2, 3, ..., 9]
```

You can also use square brackets to "slice" lists:

```
first_three   = x[:3]               # [-1, 1, 2]
three_to_end = x[3:]                # [3, 4, ..., 9]
one_to_four = x[1:5]                # [1, 2, 3, 4]
last_three = x[-3:]                 # [7, 8, 9]
without_first_and_last = x[1:-1]    # [1, 2, ..., 8]
copy_of_x = x[:]                    # [-1, 1, 2, ..., 9]
```

Python has an `in` operator to check for list membership:

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

This check involves examining the elements of the list one at a time, which means that you probably shouldn't use it unless you know your list is pretty small (or unless you don't care how long the check takes).

It is easy to concatenate lists together:

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x is now [1,2,3,4,5,6]
```

If you don't want to modify x you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]       # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)      # x is now [1, 2, 3, 0]
```

```
y = x[-1]        # equals 0
z = len(x)       # equals 4
```

It is often convenient to *unpack* lists if you know how many elements they contain:

```
x, y = [1, 2]    # now x is 1, y is 2
```

although you will get a `ValueError` if you don't have the same numbers of elements on both sides.

It's common to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]    # now y == 2, didn't care about the first element
```

Tuples

Tuples are lists' immutable cousins. Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple. You specify a tuple by using parentheses (or nothing) instead of square brackets:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

Tuples are a convenient way to return multiple values from functions:

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)    # equals (5, 6)
s, p = sum_and_product(5, 10) # s is 15, p is 50
```

Tuples (and lists) can also be used for *multiple assignment*:

```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

Another fundamental data structure is a dictionary, which associates *values* with *keys* and allows you to quickly retrieve the value corresponding to a given key:

```
empty_dict = {}                         # Pythonic
empty_dict2 = dict()                    # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 }    # dictionary literal
```

You can look up the value for a key using square brackets:

```
joels_grade = grades["Joel"]            # equals 80
```

But you'll get a KeyError if you ask for a key that's not in the dictionary:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

You can check for the existence of a key using in:

```
joel_has_grade = "Joel" in grades     # True
kate_has_grade = "Kate" in grades     # False
```

Dictionaries have a get method that returns a default value (instead of raising an exception) when you look up a key that's not in the dictionary:

```
joels_grade = grades.get("Joel", 0)   # equals 80
kates_grade = grades.get("Kate", 0)   # equals 0
no_ones_grade = grades.get("No One") # default default is None
```

You assign key-value pairs using the same square brackets:

```
grades["Tim"] = 99                    # replaces the old value
grades["Kate"] = 100                  # adds a third entry
num_students = len(grades)            # equals 3
```

We will frequently use dictionaries as a simple way to represent structured data:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Besides looking for specific keys we can look at all of them:

```
tweet_keys   = tweet.keys()     # list of keys
tweet_values = tweet.values()   # list of values
tweet_items  = tweet.items()    # list of (key, value) tuples

"user" in tweet_keys            # True, but uses a slow list in
"user" in tweet                 # more Pythonic, uses faster dict in
"joelgrus" in tweet_values      # True
```

Dictionary keys must be immutable; in particular, you cannot use lists as keys. If you need a multipart key, you should use a tuple or figure out a way to turn the key into a string.

defaultdict

Imagine that you're trying to count the words in a document. An obvious approach is to create a dictionary in which the keys are words and the values are counts. As you check each word, you can increment its count if it's already in the dictionary and add it to the dictionary if it's not:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

You could also use the "forgiveness is better than permission" approach and just handle the exception from trying to look up a missing key:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

A third approach is to use `get`, which behaves gracefully for missing keys:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Every one of these is slightly unwieldy, which is why `defaultdict` is useful. A `defaultdict` is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zero-argument function you provided when you created it. In order to use `defaultdict`s, you have to import them from `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

They can also be useful with `list` or `dict` or even your own functions:

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0,1]}
```

These will be useful when we're using dictionaries to "collect" results by some key and don't want to have to check every time to see if the key exists yet.

Counter

A `Counter` turns a sequence of values into a `defaultdict(int)`-like object mapping keys to counts. We will primarily use it to create histograms:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
```

This gives us a very simple way to solve our `word_counts` problem:

```
word_counts = Counter(document)
```

A `Counter` instance has a `most_common` method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print word, count
```

Sets

Another data structure is `set`, which represents a collection of *distinct* elements:

```
s = set()
s.add(1)       # s is now { 1 }
s.add(2)       # s is now { 1, 2 }
s.add(2)       # s is still { 1, 2 }
x = len(s)     # equals 2
y = 2 in s     # equals True
z = 3 in s     # equals False
```

We'll use sets for two main reasons. The first is that `in` is a very fast operation on sets. If we have a large collection of items that we want to use for a membership test, a set is more appropriate than a list:

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list     # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # very fast to check
```

The second reason is to find the *distinct* items in a collection:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                # 6
item_set = set(item_list)                 # {1, 2, 3}
num_distinct_items = len(item_set)        # 3
distinct_item_list = list(item_set)       # [1, 2, 3]
```

We'll use `sets` much less frequently than `dicts` and `lists`.

Control Flow

As in most programming languages, you can perform an action conditionally using `if`:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

You can also write a *ternary* if-then-else on one line, which we will do occasionally:

```
parity = "even" if x % 2 == 0 else "odd"
```

Python has a `while` loop:

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

although more often we'll use `for` and `in`:

```
for x in range(10):
    print x, "is less than 10"
```

If you need more-complex logic, you can use `continue` and `break`:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print x
```

This will print `0`, `1`, `2`, and `4`.

Truthiness

Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False
```

Python uses the value `None` to indicate a nonexistent value. It is similar to other languages' `null`:

```
x = None
print x == None    # prints True, but is not Pythonic
print x is None    # prints True, and is Pythonic
```

Python lets you use any value where it expects a Boolean. The following are all "Falsy":

- `False`
- `None`
- `[]` (an empty `list`)
- `{}` (an empty `dict`)
- `""`
- `set()`
- `0`
- `0.0`

Pretty much anything else gets treated as `True`. This allows you to easily use `if` statements to test for empty lists or empty strings or empty dictionaries or so on. It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

A simpler way of doing the same is:

```
first_char = s and s[0]
```

since `and` returns its second value when the first is "truthy," the first value when it's not. Similarly, if `x` is either a number or possibly `None`:

```
safe_x = x or 0
```

is definitely a number.

Python has an `all` function, which takes a list and returns `True` precisely when every element is truthy, and an `any` function, which returns `True` when at least one element is truthy:

```
all([True, 1, { 3 }])   # True
all([True, 1, {}])      # False, {} is falsy
any([True, 1, {}])      # True, True is truthy
all([])                 # True, no falsy elements in the list
any([])                 # False, no truthy elements in the list
```

The Not-So-Basics

Here we'll look at some more-advanced Python features that we'll find useful for working with data.

Sorting

Every Python list has a `sort` method that sorts it in place. If you don't want to mess up your list, you can use the `sorted` function, which returns a new list:

```
x = [4,1,2,3]
y = sorted(x)     # is [1,2,3,4], x is unchanged
x.sort()          # now x is [1,2,3,4]
```

By default, `sort` (and `sorted`) sort a list from smallest to largest based on naively comparing the elements to one another.

If you want elements sorted from largest to smallest, you can specify a `reverse=True` parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with `key`:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

List Comprehensions

Frequently, you'll want to transform a list into another list, by choosing only certain elements, or by transforming elements, or both. The Pythonic way of doing this is *list comprehensions*:

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

You can similarly turn lists into dictionaries or sets:

```
square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set  = { x * x for x in [1, -1] }       # { 1 }
```

If you don't need the value from the list, it's conventional to use an underscore as the variable:

```
zeroes = [0 for _ in even_numbers]      # has the same length as even_numbers
```

A list comprehension can include multiple `for`s:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

and later `for`s can use the results of earlier ones:

```python
increasing_pairs = [(x, y)                       # only pairs with x < y,
                    for x in range(10)           # range(lo, hi) equals
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi - 1]
```

We will use list comprehensions a lot.

Generators and Iterators

A problem with lists is that they can easily grow very big. `range(1000000)` creates an actual list of 1 million elements. If you only need to deal with them one at a time, this can be a huge source of inefficiency (or of running out of memory). If you potentially only need the first few values, then calculating them all is a waste.

A *generator* is something that you can iterate over (for us, usually using `for`) but whose values are produced only as needed (*lazily*).

One way to create generators is with functions and the `yield` operator:

```python
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

The following loop will consume the `yield`ed values one at a time until none are left:

```python
for i in lazy_range(10):
    do_something_with(i)
```

(Python actually comes with a `lazy_range` function called `xrange`, and in Python 3, `range` itself is lazy.) This means you could even create an infinite sequence:

```python
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

although you probably shouldn't iterate over it without using some kind of `break` logic.

The flip side of laziness is that you can only iterate through a generator once. If you need to iterate through something multiple times, you'll need to either recreate the generator each time or use a list.

A second way to create generators is by using `for` comprehensions wrapped in parentheses:

CHAPTER 3

Visualizing Data

I believe that visualization is one of the most powerful means of achieving personal goals.

—Harvey Mackay

A fundamental part of the data scientist's toolkit is data visualization. Although it is very easy to create visualizations, it's much harder to produce *good* ones.

There are two primary uses for data visualization:

- To *explore* data
- To *communicate* data

In this chapter, we will concentrate on building the skills that you'll need to start exploring your own data and to produce the visualizations we'll be using throughout the rest of the book. Like most of our chapter topics, data visualization is a rich field of study that deserves its own book. Nonetheless, we'll try to give you a sense of what makes for a good visualization and what doesn't.

matplotlib

A wide variety of tools exists for visualizing data. We will be using the `matplotlib` library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the Web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

In particular, we will be using the `matplotlib.pyplot` module. In its simplest use, `pyplot` maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it (with `savefig()`) or display it (with `show()`).

For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```

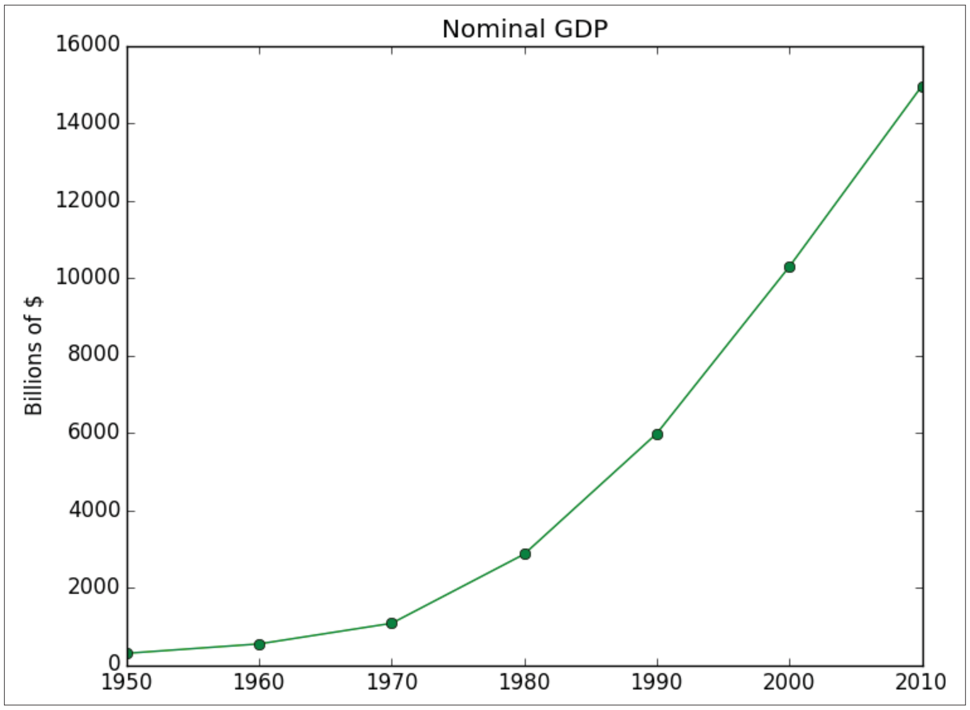

Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with (for example) axis labels, line styles, and point markers. Rather than attempt a comprehensive treatment of these options, we'll just use (and call attention to) some of them in our examples.

Although we won't be using much of this functionality, `matplotlib` is capable of producing complicated plots within plots, sophisticated formatting, and interactive visualizations. Check out its documentation if you want to go deeper than we do in this book.

Bar Charts

A bar chart is a good choice when you want to show how some quantity varies among some *discrete* set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# bars are by default width 0.8, so we'll add 0.1 to the left coordinates
# so that each bar is centered
xs = [i + 0.1 for i, _ in enumerate(movies)]

# plot bars with left x-coordinates [xs], heights [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")

# label x-axis with movie names at bar centers
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

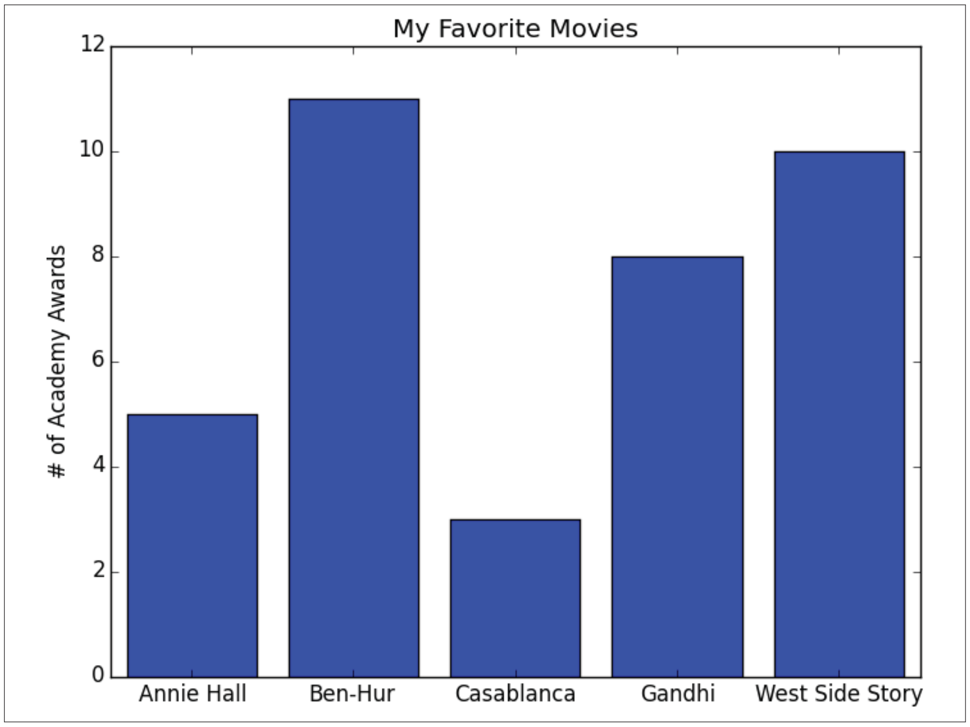

Figure 3-2. A simple bar chart

A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are *distributed*, as in Figure 3-3:

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # shift each bar to the left by 4
        histogram.values(),                # give each bar its correct height
        8)                                 # give each bar a width of 8

plt.axis([-5, 105, 0, 5])                  # x-axis from -5 to 105,
                                           # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])    # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```

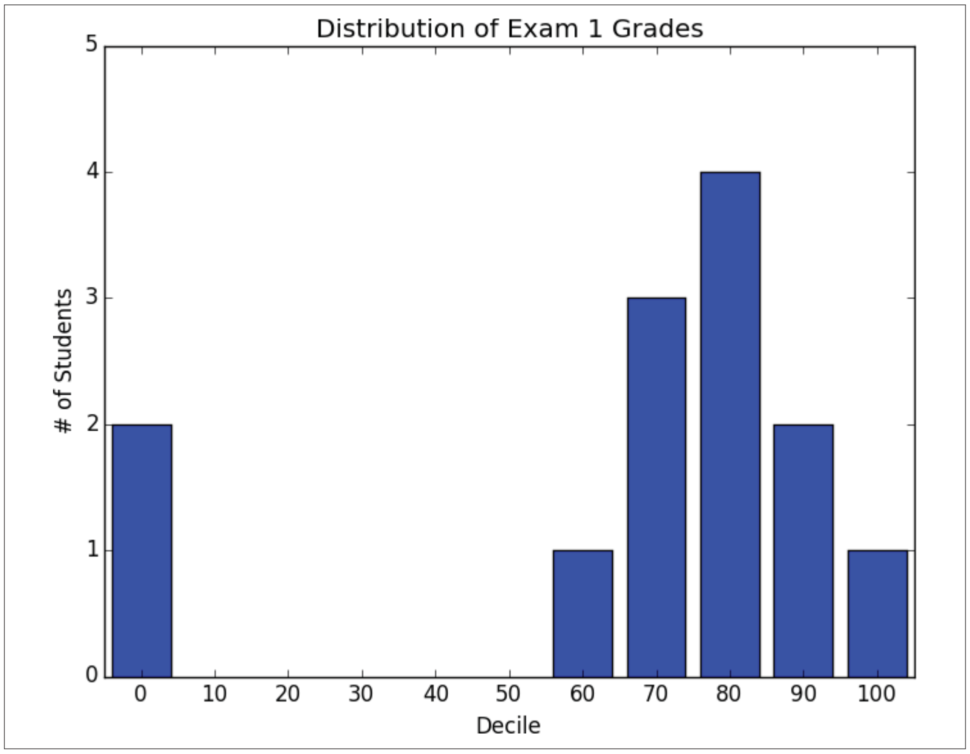

Figure 3-3. Using a bar chart for a histogram

The third argument to `plt.bar` specifies the bar width. Here we chose a width of 8 (which leaves a small gap between bars, since our buckets have width 10). And we shifted the bar left by 4, so that (for example) the "80" bar has its left and right sides at 76 and 84, and (hence) its center at 80.

The call to `plt.axis` indicates that we want the x-axis to range from -5 to 105 (so that the "0" and "100" bars are fully shown), and that the y-axis should range from 0 to 5. And the call to `plt.xticks` puts x-axis labels at 0, 10, 20, …, 100.

Be judicious when using `plt.axis()`. When creating bar charts it is considered especially bad form for your y-axis not to start at 0, since this is an easy way to mislead people (Figure 3-4):

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("# of times I heard someone say 'data science'")

# if you don't do this, matplotlib will label the x-axis 0, 1
# and then add a +2.013e3 off in the corner (bad matplotlib!)
plt.ticklabel_format(useOffset=False)

# misleading y-axis only shows the part above 500
plt.axis([2012.5,2014.5,499,506])
plt.title("Look at the 'Huge' Increase!")
plt.show()
```

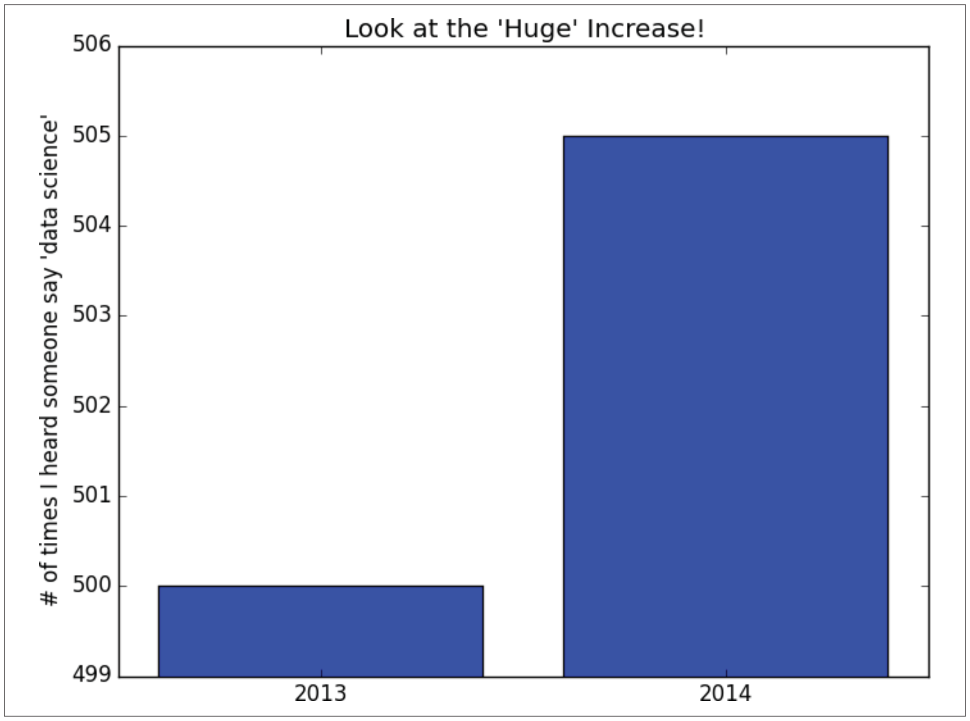

Figure 3-4. A chart with a misleading y-axis

In Figure 3-5, we use more-sensible axes, and it looks far less impressive:

```
plt.axis([2012.5,2014.5,0,550])
plt.title("Not So Huge Anymore")
plt.show()
```

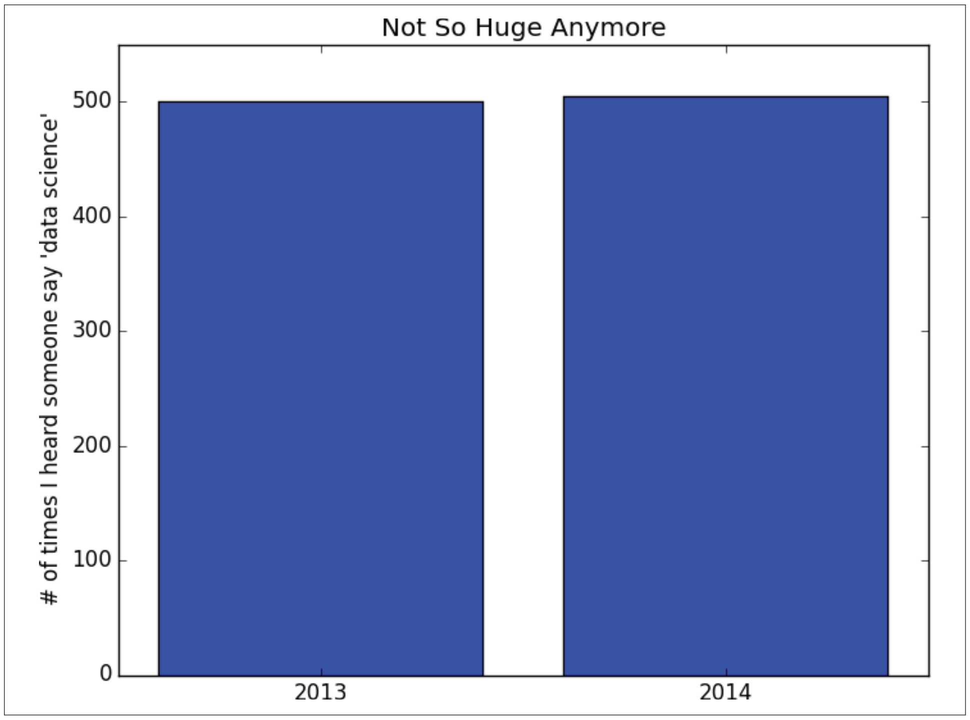

Figure 3-5. The same chart with a nonmisleading y-axis

Line Charts

As we saw already, we can make line charts using `plt.plot()`. These are a good choice for showing *trends*, as illustrated in Figure 3-6:

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# we can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line

# because we've assigned labels to each series
# we can get a legend for free
# loc=9 means "top center"
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff")
plt.show()
```

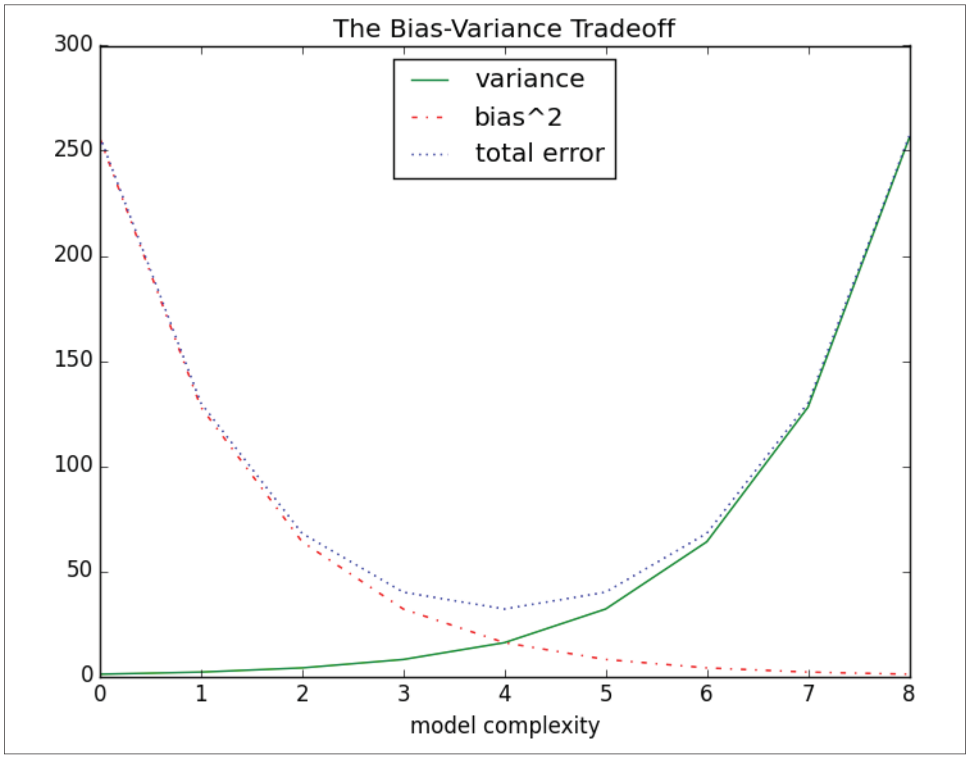

Figure 3-6. Several line charts with a legend

Scatterplots

A scatterplot is the right choice for visualizing the relationship between two paired sets of data. For example, Figure 3-7 illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# label each point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # put the label with its point
        xytext=(5, -5),                  # but slightly offset
        textcoords='offset points')

plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

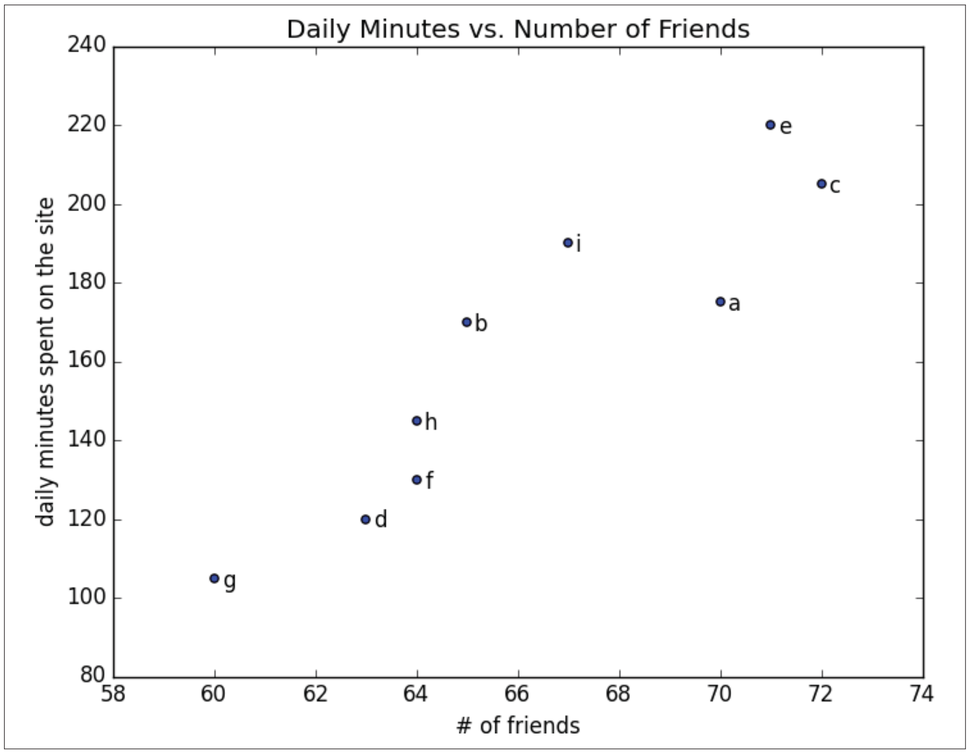

Figure 3-7. A scatterplot of friends and time on the site

If you're scattering comparable variables, you might get a misleading picture if you let `matplotlib` choose the scale, as in Figure 3-8:

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```

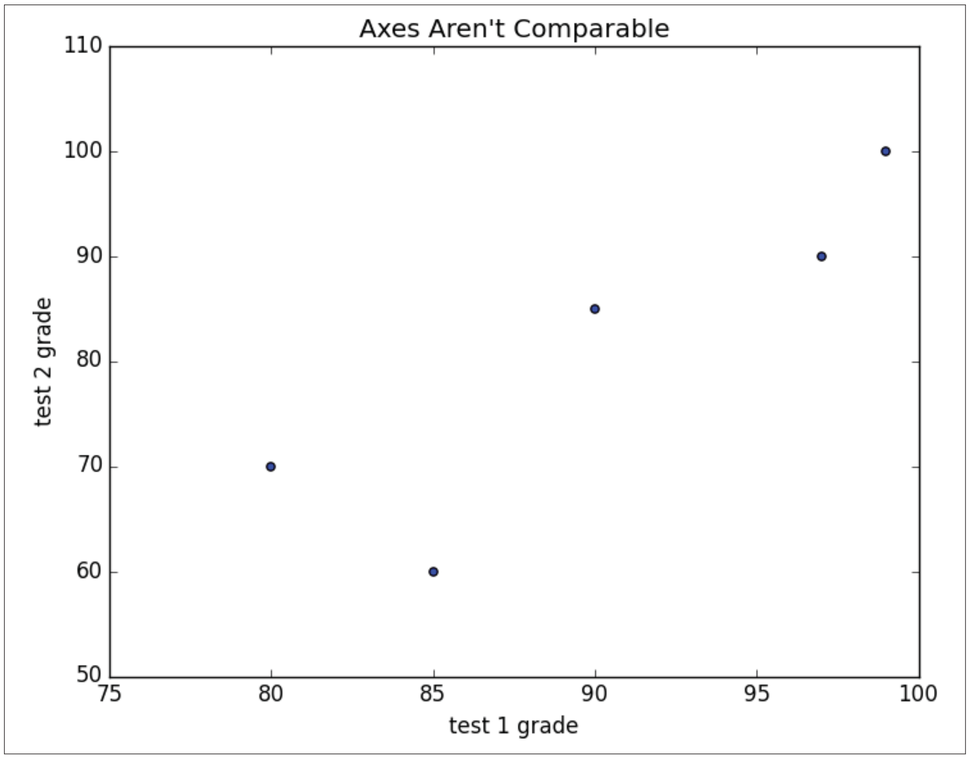

Figure 3-8. A scatterplot with uncomparable axes

If we include a call to `plt.axis("equal")`, the plot (Figure 3-9) more accurately shows that most of the variation occurs on test 2.

That's enough to get you started doing visualization. We'll learn much more about visualization throughout the book.

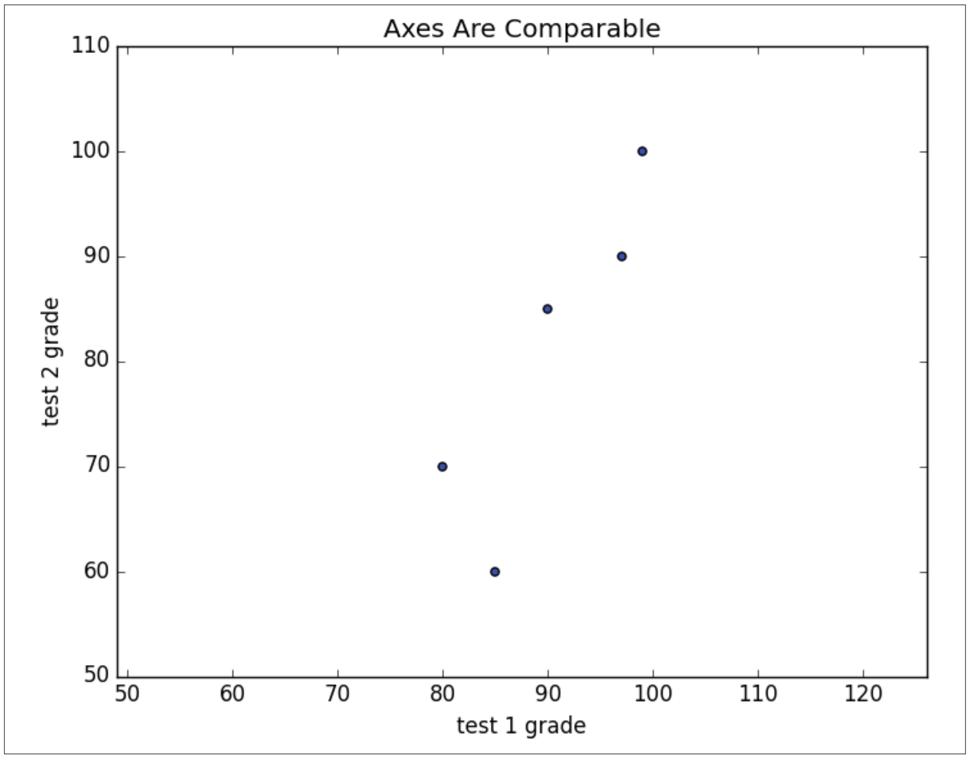

Figure 3-9. The same scatterplot with equal axes

For Further Exploration

- seaborn is built on top of `matplotlib` and allows you to easily produce prettier (and more complex) visualizations.
- D3.js is a JavaScript library for producing sophisticated interactive visualizations for the web. Although it is not in Python, it is both trendy and widely used, and it is well worth your while to be familiar with it.
- Bokeh is a newer library that brings D3-style visualizations into Python.
- ggplot is a Python port of the popular R library `ggplot2`, which is widely used for creating "publication quality" charts and graphics. It's probably most interesting if you're already an avid `ggplot2` user, and possibly a little opaque if you're not.

CHAPTER 9
Getting Data

To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.

—F. Scott Fitzgerald

In order to be a data scientist you need data. In fact, as a data scientist you will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data. In a pinch, you can always type the data in yourself (or if you have minions, make them do it), but usually this is not a good use of your time. In this chapter, we'll look at different ways of getting data into Python and into the right formats.

stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print count
```

You could then use these to count how many lines of a file contain numbers. In Windows, you'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

whereas in a Unix system you'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

The | is the pipe character, which means "use the output of the left command as the input of the right command." You can build pretty elaborate data-processing pipelines this way.

> If you are using Windows, you can probably leave out the `python` part of this command:
>
> ```
> type SomeFile.txt | egrep.py "[0-9]" | line_count.py
> ```
>
> If you are on a Unix system, doing so might require a little more work.

Similarly, here's a script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)   # non-zero exit code indicates error

counter = Counter(word.lower()                      # lowercase words
                  for line in sys.stdin             #
                  for word in line.strip().split()  # split on spaces
                  if word)                          # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
```

```
        sys.stdout.write(word)
        sys.stdout.write("\n")
```

after which you could do something like:

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

If you are a seasoned Unix programmer, you are probably familiar with a wide variety of command-line tools (for example, `egrep`) that are built into your operating system and that are probably preferable to building your own from scratch. Still, it's good to know you can if you need to.

Reading Files

You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only
file_for_reading = open('reading_file.txt', 'r')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)
# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:                # look at each line in the file
        if re.match("^#",line):      # use a regex to see if it starts with '#'
            starts_with_hash += 1    # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip()` it before doing anything with it.

For example, imagine you have a file full of email addresses, one per line, and that you need to generate a histogram of the domains. The rules for correctly extracting domains are somewhat subtle (e.g., the Public Suffix List), but a good first approximation is to just take the parts of the email addresses that come after the `@`. (Which gives the wrong answer for email addresses like `joel@mail.datasciencester.com`.)

```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Delimited Files

The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line. These files are very often either *comma-separated* or *tab-separated*. Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably do). For this reason, it's pretty much always a mistake to try to parse them yourself. Instead, you should use Python's `csv` module (or the `pandas` library). For technical reasons that you should feel free to blame on Microsoft, you should always work with `csv` files in *binary* mode by including a *b* after the *r* or *w* (see Stack Overflow).

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014   AAPL    90.91
6/20/2014   MSFT    41.68
6/20/2014   FB  64.5
6/19/2014   AAPL    91.86
6/19/2014   MSFT    41.51
6/19/2014   FB  64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

you can either skip the header row (with an initial call to `reader.next()`) or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

`csv.writer` will do the right thing if your fields themselves have commas in them. Your own hand-rolled writer probably won't. For example, if you attempt:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many commas in it!
        f.write("\n")                    # row might have newlines as well!
```

You will end up with a `csv` file that looks like:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

and that no one will ever be able to make sense of.

Scraping the Web

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

HTML and the Parsing Thereof

Pages on the Web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

In a perfect world, where all web pages are marked up semantically for our benefit, we would be able to extract data using rules like “find the `<p>` element whose `id` is `subject` and return the text it contains.” In the actual world, HTML is not generally well-formed, let alone annotated. This means we’ll need help making sense of it.

To get data out of HTML, we will use the BeautifulSoup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.3.2 (`pip install beautifulsoup4`), which is what we’ll be using. We’ll also be using the requests library