Beginning Python Visualization: Crafting Visual Transformation Scripts

ISBN-13 (pbk): 978-1-4302-1843-2

ISBN-13 (electronic): 978-1-4302-1844-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Frank Pohlmann, Michelle Lowman
Technical Reviewer: C. Titus Brown
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Ami Knox
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Dina Quan
Proofreader: Liz Welch
Indexer: Julie Grady
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Navigating the World of Data Visualization

A Case Study

As an engineer, I work with data all the time. I parse log files, analyze data, estimate values, and compare the results with theory. Things don't always add up. So I double-check my analysis, perform more calculations, or run simulations to better understand the results. I refer to previous work because the ideas are similar or sometimes because they're dissimilar. I look at the graphs and realize I'm missing some crucial information. So I add the missing data, but it's noisy and needs filtering. Eventually, I realize my implementation of the algorithm is poor or that there is a better algorithm with better results, and so back to square one. It's an iterative process: tweak, test, tweak again until I'm satisfied with the results.

Those are the tasks surrounding research and development (R&D) work. And to be honest, there's no systematic method. Most of the time, research is organized chaos. The emphasis, however, should be on organized, not chaos. Data should be analyzed and presented in a clear and coherent manner. Sources for graphs well understood and verified to be accurate. Algorithms tested and proven to be working as intended. The system should be flexible. Introducing new ideas and challenging previous methods should be easy and testing new ideas on current data fast and efficient.

In this book I will attempt to address all the topics associated with data processing and visualization: managing files and directories, reading files of varying formats, performing signal processing and numerical analysis in a high-level programming language similar to MATLAB and GNU-Octave, and teaching you Python, a rich and powerful programming language, along the way.

In a nutshell, *Beginning Python Visualization* deals with the processing, analysis, manipulation, and visualization of data using the Python programming language. The book covers the following:

- Fundamentals of the Python programming language required for data analysis and visualization
- Data files, format, and organization, as well as methods and guidelines for selecting file formats and storing and organizing data to enable fast, efficient data processing
- Readily available Python packages for numerical analysis, signal and image processing, graphing and plotting, and more

Gathering Data

We spend a considerable time recording and analyzing data. Data is stored in various formats depending on the tools used to collect it, the nature of the data (e.g., pictures vs. sampled analog data), the application that will later process the data, and personal preferences. Data files are of varying sizes; some are very large, others are smaller but in larger quantities. Data organization adds another level of complexity. Files can be stored in directories according to date, grouped together in one big directory or in a database, or adhere to a different scheme altogether. Typically, the number of data files or the amount of data per file is too large to allow skimming or browsing with an editor or viewer. Methods and tools are required to find the data and analyze it to produce meaningful results.

Case Study: GPS Data

You just got a USB GPS receiver for your birthday! You'd like to analyze GPS data and find out how often you exceed the speed limit and how much time you spend in traffic. You'd like to track data over a year, or even longer.

Some hardware background: most USB GPS receivers behave as serial ports (this is also true for Bluetooth GPS devices). What this means is that once a GPS is connected, and assuming it's installed properly, reading GPS data is as simple as opening the COM port associated with the GPS and reading the values. GPS values are typically clear text values: numbers and text. Of course, if you're planning on recording data from your car, it would make a lot of sense to hook it up to a laptop rather than a desktop.

We would like to record, analyze, and visualize the GPS data, in Python. First things first: recording GPS data.

■ **Note** If you wish to follow along with the remainder of the chapter by means of issuing the commands yourself and viewing the results, you might first want to refer to Chapter 2 and set up Python on your system. That being said, it's not necessary, and you can follow along to get an understanding of the book and its purpose. In fact, I encourage you to come back to this chapter and read it again after you've had more experience with Python.

Python is an interpreted programming language. What this means is each command is first read and then executed, in contrast to compiled programming languages, where the entire program is evaluated (compiled) and then executed. One of the important features of

interpreted programming languages is that it's easy to run them interactively. That is, perform a command, examine the results, perform more commands, and examine more results, and so on. The ability to run Python interactively is very useful, and it allows you to examine topics as you learn them.

It's also possible to run Python scripts, that is, noninteractively, and there are several ways to do that. You can run scripts from the interactive Python prompt by issuing the command `execfile('scriptname.py')`. Or you can enter `python scriptname.py` at the command-line interface of your operating system. If you're using IPython, you can issue the command `run scriptname.py` instead; and if you're running IDLE, the Python GUI, you can open the script and press F5 to execute it. The `.py` extension is a common convention that distinguishes Python scripts from other files.

Back to recording GPS data. To be able to access the serial port from Python, we'll be using the pySerial module. PySerial, as the name suggests, allows seamless access to serial ports. To use pySerial we must first read the module to memory, that is, import it using the `import` command. If all goes well, we'll be presented with the Python prompt again.

```
>>> import serial
```

■ **Note** To distinguish between interactive sessions and Python scripts, when code starts with `>>>`, it means that the code was run on Python interactively. In case the ellipsis symbol (`...`) appears, it means that this is a continuation of a previously interactively entered command. Lines of text following the symbols `...` or `>>>` is Python's response to the issued command. A code listing that does not start with `>>>` is a script written in an editor, and in order to execute it you will have to save it under `scriptname.py` (or some other name) and execute it as described previously.

Scanning Serial Ports

Next, we need to find the serial port parameters: the *baud rate* and the *port number*. The baud rate is a GPS parameter, so it's best to consult the GPS manual (not to worry if you can't find this information, I'll discuss later how to "guess" what it is). As for the port number, this is determined by your operating system. If you're not sure how to find the port number, or if the port number keeps changing when you plug and unplug your GPS, you can use the short program in Listing 1-1 to identify active serial ports.

Listing 1-1. *Scanning Serial Ports with* `scanport.py`

```
import serial

found = False
for i in range(64):
    try:
        ser = serial.Serial(i)
        ser.close()
        print "Found COM", i+1
        found = True
```

```
    except serial.serialutil.SerialException:
        pass

if not found:
    print "No ports found, make sure GPS is connected."
```

■ **Note** Short programs are typically referred to as *scripts*.

Run `scanport.py` and note the result:

```
>>> execfile('scanport.py')
Found COM 5
```

This is a rather quick introduction to Python! First, let's dissect `scanport.py` line by line. The first line, `import serial`, loads the pySerial module. We then assign to the Boolean variable `found` the value `False`; this variable will be used as an indication of whether a serial port was found or not. We proceed with the `for` loop: the loop goes over the values between `0` and `63` as implied by `range(64)` (most systems have less than 64 virtual COM ports). The function `range(N)` returns a list of values from `0` to `N-1`. Our approach to seeing what ports are available is rather simple: try and open the port, and if all goes well, that port is a candidate. If it was not possible to open the port, just ignore that port. And so this is exactly how it's coded!

This is a common motto in Python: It's Easier to Ask Forgiveness than Permission, or EAFP. The idea is this: Try and perform an operation. If all goes well, great. If not, handle it with the `except` clause, or more figuratively, ask forgiveness. This is eloquently coded with the `try/except` mechanism.

In our case, the function that's most likely to fail (raise an exception) is the one that tries to open a nonexistent port: `ser = serial.Serial(i)`. The function `Serial()` is part of the serial module (notice case sensitivity). To access functions within modules, you specify the module name, dot (`.`), and the function name. So to call the function `Serial()` within the module serial, write `serial.Serial()`. The function `Serial()` takes one parameter: the port number. Python, like C, starts counting at 0, so remember to subtract 1 from your virtual COM port when passing a parameter to the function. My GPS turned out to be connected to COM5, so a call to `serial.Serial(4)` will allow me access to the GPS. If the port is successfully opened, no exception is raised, and the opened port is associated with the variable `ser`.

The next line in the `try` block, `ser.close()`, tries to close the port. Closing the port renders it accessible to other applications, including your own. If you neglect to close the port, Python will close it for you once the variable associated with it, `ser`, is no longer in use. We also print out a message saying the port is a good candidate and set the `found` flag to `True`.

If the block of commands under `try` fails, the block of commands under `except` is executed assuming the `except` condition is met. In our case, if an exception occurred, and if the exception is of type `serial.serialutil.SerialException`, which means the port could not be opened, we want to simply disregard it. This is done using the `pass` statement, which does nothing.

Lastly, once the `for` loop is complete, and in case no port was found, a message indicating that is printed.

■ **Note** The indentation (tabs) in Python is important because it groups commands together. This is also true when using Python in an interactive mode. All lines with the same indentation are considered one block. Python's indentation is equivalent to C/C++ curly braces—{}.

Recording GPS Data

Let's start gathering data. Enter code in Listing 1-2 and record it in the file `record_gps.py`.

Listing 1-2. *record_gps.py*

```
import time, serial

# change these parameters to your GPS parameters
ser = serial.Serial(4)
ser.baudrate = 4800
fmt = "../data/GPS-%4d-%02d-%02d-%02d-%02d-%02d.csv"

filename = fmt % time.localtime()[0:6]
f = open(filename, 'wb')
while True:
    line = ser.readline()
    f.write(line)
    print line,
```

This time, we've imported another module: time. The time module provides access to date and time functions, and we'll use those to name our GPS data files. We also introduce an important notion here, comments! Comments in Python are denoted by the # sign and are similar to C++ double slash notation, //. Everything from that point onward is considered a remark. If the # sign is at the beginning of a line, then the entire line is a remark, usually describing the next line or block of code. The exception to the # sign indicating a remark is if it is quoted inside a string, as follows: "#".

Don't forget to change the port number to point at your serial port (minus 1) and set the proper baud rate. Determining the baud rate is not complex either—best to consult the manual. Mine turned out to be 4800, but if you're not sure, you can tweak this parameter. The script `record_gps.py` will print the output from the GPS on screen so you can change the baud rate value (in multiples of 2, for example 4800, 9600, and so on) until you see some meaningful results (i.e., text and numbers).

Running `record_gps.py` (I'll get to how it works soon) yields GPS data:

```
>>> execfile('record_gps.py')
$GPRMC,140053.00,A,4454.1740,N,09325.0143,W,000.0,128.7,300508,001.1,E,A*2E
$GPGGA,140053.00,4454.1740,N,09325.0143,W,1,09,01.1,00289.8,M,-030.7,M,,*5E
$GPGSA,A,3,21,15,18,24,26,29,06,22,,03,,,02.0,01.1,01.7*04
$GPGSV,3,1,12,21,75,306,40,15,59,075,46,18,57,269,49,24,56,115,46*79
$GPGSV,3,2,12,26,48,059,43,29,27,188,48,06,25,308,41,22,18,257,33*7D
$GPGSV,3,3,12,08,14,060,,03,11,320,32,09,06,144,,16,04,311,*7C
```

```
$GPRMC,140054.00,A,4454.1740,N,09325.0143,W,000.0,128.7,300508,001.1,E,A*29
$GPGGA,140054.00,4454.1740,N,09325.0143,W,1,09,01.1,00289.8,M,-030.7,M,,*59
$GPGSA,A,3,21,15,18,24,26,29,06,22,,03,,,02.0,01.1,01.7*04
```

Data is being recorded to file as it is displayed. When you wish to stop viewing and recording GPS data, press Ctrl+C. If you're running in an interactive Python, once you issue Ctrl+C, be sure to close the serial port, or you won't be able to rerun the script `record_gps.py`. To close the port, issue the following command:

```
>>> ser.close()
```

It's also a good idea to close the file:

```
>>> f.close()
```

Let's turn back so I can explain how `record_gps.py` works. The heart of the script lies in the following lines of code:

```
while True:
    line = ser.readline()
    f.write(line)
    print line,
```

This is a straightforward implementation. The first line, `while True:`, instructs that the following block should be run indefinitely, that is, in an infinite loop. That's why you need to press Ctrl+C to stop recording. The next three lines are then executed continuously. What we do is read a line of text from the serial port, store it to file, and print it to screen. Reading GPS data is carried out by the command `line = ser.readline()`. Writing that data to a file for later processing is done by `f.write(line)`. Printing the data to screen so the user has some visual feedback is done with `print line,`. The reason for the comma following `line` is to suppress an extra line break.

Data Organization

Let's turn to selecting file format, file naming conventions, and data location. Now there isn't a good solution that fits all, but the methodologies and ideas are simple. The method I'll use here is based on file names. I'll show you how to name data files in a way that lends itself easily to automatic processing later on.

File Format

A file format is a set of rules describing the contents of a file. For the GPS problem, we'll choose the Comma Separated Values (CSV) file format. CSV files are text files with values separated by commas. For example:

```
$GPGSV,3,2,12,06,43,096,37,07,41,291,38,16,39,052,32,27,34,291,34*76
$GPGSV,3,3,12,19,26,152,35,08,06,280,,10,00,337,,00,00,000,*74
$GPRMC,233547.32,A,4455.6446,N,09329.3400,W,030.1,272.5,040608,001.1,E,A*2E
$GPGGA,233547.32,4455.6446,N,09329.3400,W,1,06,02.8,00299.0,M,-030.7,M,,*5A
```

CSV is a popular format recognized by most spreadsheets and database applications and, of course, text editors, seeing as they're really just text files. As it turns out, the data the GPS outputs is already comma separated, so all that's required is to save this information to a file, as is.

File Naming Conventions

We turn to selecting proper file names for our data files. File names should be unique so that files won't be accidentally overwritten. File names should be descriptive, that is, tell us something about the contents. Lastly, we'd like the file name extension to tell us how to view the file. The latter is typically achieved by selecting a proper extension, in our case, `.csv`. Here are the naming conventions I chose for this example:

- File names holding GPS data will start with the text "GPS-".
- Next is the date and time in ISO format with the separating colons omitted and a hyphen between the date and time: YYYY-mm-dd-HH-MM-SS, where YYYY stands for year, mm for month, dd for day, HH for hours, MM for minutes, and SS for seconds. In case a value is one digit and two digits are required, values will be padded with zeros, for example, the month of May will be denoted by 05, not 5. For additional information regarding the ISO format, refer to ISO 8601, "Data elements and interchange formats—Information interchange—Representation of dates and times" (`http://www.iso.org`).
- All files will have a `.csv` extension.

Following these conventions, a file name might look like this:

```
GPS-2008-05-30-09-10-52.csv
```

Data Location

This is where we store data files:

- All data files are stored in directory `data`. All scripts are stored in directory `src`. Both directories are under the same parent directory `Ch1`. So a relative path from `src` to `data` is `../data`.
- It's a good idea to also add a `Readme.txt` file. Readme files are clear text files describing the contents of a directory, in as much detail as deemed reasonable: the data source, data acquisition system, person in charge of data gathering, reason for gathering the data, and so on. Here's an example:

```
Data recorded from a USB GPS receiver, connected to a Lenovo laptop T60.
Data was gathered via the serial port stored to clear text files (CSV).
Measurements were taken to estimate speed and time spent in traffic.
Gathered by Shai Vaingast.
Date: throughout 2008, see file timestamps.
```

Data Analysis

Once data is organized and accessible in files, the next step is to extract information. Information can be a value, a graph, or a report pertaining to the problem at hand.

The idea is to use Python's scripting abilities and the wide range of readily available packages to write a fully automated application to process, analyze, and visualize data. Scripts are small pieces of code that are written relatively quickly in a high-level programming language. The key word here is productivity, the ability to change and test algorithms and extract results fast. Scripts might not be highly efficient in terms of processing speed, but written properly, they should not slow down running times. For example, a script might generate graphs or search the hard drive for data files, analyze log files, and extract the maximum and minimum temperatures, or in our case, analyze GPS data.

Back to our GPS case study. The following is the algorithm we'll follow:

1. Compile a list of all the data files.
2. For each file
   a. Read the data.
   b. Process the data.
   c. Plot the data.

Walking Directories

To compile a list of all the files having GPS data, we'll use the function `os.walk()` provided with the module os, which is part of the Python Standard Library. To use os, we issue `import os`.

```
>>> import os
>>> for root, dirs, files in os.walk('../data'):
...    print root, dirs, files
...
../data [] ['GPS-2008-05-30-09-00-50.csv', 'GPS-2008-05-30-09-10-52.csv',
'Readme.txt']
```

■ **Note** To be able to change directories within the Python interpreter, first issue `import os`. Then, to change to a directory, issue `os.chdir(directory_path)`. To list directory contents, you can use `os.listdir(directory_path)`. Some interpreters like IPython let you use, among other enhancements, shell-like commands such as `cd` and `ls`, which add considerably to usability.

The function `os.walk()` iterates through the directory `data` and its subdirectories recursively, looking for files and folders, storing the results in variables `root`, `dirs`, and `files`. The second line prints out the root directory for our search, in our case `../data` (notice the relative path), then the subdirectories, and lastly the files themselves, in a list. I've only recorded two data files, but as time progresses, more data is added to this folder, and the number can

increase substantially. Since we have no subdirectories in folder `data`, the output corresponding to `dirs` should be an empty list, which is denoted by `[]`.

`os.walk()` is a bit of an overkill here. In our case, directory `data` doesn't have any subdirectories, and we could have just as easily listed the contents of the directory using the `os.listdir()` function call, as follows:

```
>>> os.listdir('../data')
['GPS-2008-30-05-09-00-50.csv', 'GPS-2008-30-05-09-10-52.csv', 'Readme.txt']
```

However, `os.walk()` is very useful. It's not uncommon to have files grouped together in directories and within those directories subdirectories holding more files. For example, you might want to group files in accordance with the GPS that recorded the data. Or if another driver is recording GPS data, you might want to put that data in a separate subdirectory within your `data` directory. In those cases, `os.walk()` is exactly what's needed.

Now that we have a list of all the files in directory `data`, we turn to process only those with the `.csv` extension. This is done using the `endswith()` function, which checks whether a string ends with "csv". Files that do not end with "csv" are skipped using the `continue` statement: `continue` instructs the `for` loop to skip current execution and proceed to the next element. Files that do end with "csv" are read and processed. We also introduce a function to create a full file name path from the directory and the file name, `os.path.join()`, as shown in Listing 1-3.

Listing 1-3. *Processing Only CSV Files*

```
for filename in files:
    # create full file name including path
    cur_file = os.path.join(root, filename)
    if filename.endswith('csv'):
        y = read_csv_file(cur_file)
    else:
        continue

    # only files with the .csv extension from here on
```

Reading CSV Files

Our next step is to read the files. Again, we turn to Python's built-in modules, this time the csv module. Although the CSV file format is quite popular, there's no clear definition, and each spreadsheet and database employs its own "dialect." The files we'll be processing adhere to the most basic CSV file dialect, so we'll use the default behavior of Python's csv module. Since we'll be reading several CSV files, it stands to reason to define a function to perform this task. Listing 1-4 shows this function.

Listing 1-4. *A Function to Read CSV Files*

```
def read_csv_file(filename):
    """Reads a CSV file and returns it as a list of rows."""
```

```
    data = []
    for row in csv.reader(open(filename)):
        data.append(row)
    return data
```

The first line defines a function named `read_csv_file()`. CSV file support is introduced with the csv module, so we have to `import csv` before calling the function. The function takes one variable, `filename`, and returns an array of rows holding data in the file. What I mean by this is that every line read is processed and becomes a list, with every comma-separated value as one element in that list. The function returns an array of such lists. For example:

```
>>> import csv
>>> x = read_csv_file('../data/GPS-2008-06-04-09-03-45.csv')
>>> len(x)
3683
>>> x[10]
['$GPGSV', '3', '3', '12', '29', '10', '040', '', '16', '01', '302', '', '26', '01',
'037', '', '00', '00', '000', '*72']
>>> x[1676]
['$GPGSV', '3', '1', '12', '21', '86', '258', '43', '18', '66', '286', '20', '15',
'50', '059', '45', '24', '44', '126', '43*72']
```

`len(x)` lets us know the size of the array of lists. It's also a crude way for us to ensure that data was actually read into the array.

The second line in the function is called a *docstring*, and it is characterized by three quotes (`"""`) surrounding the text in the following manner: `"""dosctring"""`. In this case, a docstring is used to document the function, that is, what it does. Issuing the command `help(funcname)` yields its docstring:

```
>>> help(read_csv_file)
Help on function read_csv_file in module __main__:

read_csv_file(filename)
    Reads a CSV file and returns it as a list of rows.
```

You should use `help()` extensively. `help()` can be invoked with functions as well as modules. For example, the following invokes help on module csv:

```
>>> help(csv)
Help on module csv:

NAME
    csv - CSV parsing and writing.

FILE
    /usr/lib/python2.5/csv.py

MODULE DOCS
    http://www.python.org/doc/current/lib/module-csv.html
```

```
DESCRIPTION
    This module provides classes that assist in the reading and writing
    of Comma Separated Value (CSV) files, and implements the interface
    described by PEP 305. Although many CSV files are simple to parse,
    the format is not formally defined by a stable specification and
    is subtle enough that parsing lines of a CSV file with something
    like line.split(",") is bound to fail.  The module supports three
    basic APIs: reading, writing, and registration of dialects.
```

Next in our dissection is the line `data = []` which declares a variable named `data` and initializes it as an empty list. `data` will be used to store the values from the CSV file.

The csv module helps us read CSV files by automating a lot of the tasks associated with reading CSV files. I will discuss CSV files and the csv module in Chapters 4 and 5, so here I'll only provide an overview.

These are the operations to perform in order to read CSV files using the csv module:

1. Open the file for reading.
2. Create a `csv.reader` object. The `csv.reader` object has functions that help us read CSV files.
3. Using the `csv.reader` object, read the data from the file, a row at a time.
4. Append every row to variable `data`.
5. Close the file.

Let's try this, a step at a time:

```
>>> f = open('../data/GPS-2008-06-04-09-03-45.csv')
>>> cr = csv.reader(f)
>>> for row in cr:
...     print row
...
 ['$GPGSA', 'A', '3', '21', '18', '15', '24', '', '22', '', '', '', '', '', '',
'03.5', '02.2', '02.7*09']
['$GPGSV', '3', '1', '12', '21', '86', '267', '39', '18', '66', '286', '44', '15',
'51', '060', '43', '24', '45', '125', '30*7A']
['$GPGSV', '3', '2', '12', '06', '28', '300', '33', '22', '27', '265', '31', '03',
'18', '312', '27', '29', '15', '185', '31*7C']
['$GPGSV', '3', '3', '12', '09', '15', '138', '31', '16', '00', '301', '', '19',
'00', '332', '', '00', '00', '000', '*70']
['$GPRMC', '140706.24', 'A', '4455.6241', 'N', '09328.0519', 'W', '011.4', '152.7',
'040608', '001.2', 'E', 'A*25']
['$GPGGA', '140706.24', '4455.6241', 'N', '09328.0519', 'W', '1', '04',
'03.0','00295.1', 'M', '-030.7', 'M', '', '*51']
['$GPGSA', 'A', '3', '21', '18', '15', '24', '', '', '', '', '', '', '', '', '08.9',
'03.0', '08.4*04']
>>> f.close()
```

First we open the data file and assign it to variable `f`. The opened file can now be referred to by the variable `f`. Next, we create a `csv.reader` object, `cr`. We associate the `csv.reader`

object, `cr`, with the file `f`. We then iterate through every row of the `csv.reader` object and print that row. Lastly, we close the file by calling `f.close()`. It is considered good practice to close the file once you're done with it, but if you neglect to do so, Python will close the file automatically once the variable `f` is no longer in use.

One of the things that you can do in Python is cascade functions. This means you can call functions on results of other functions. This process can be repeated several times. Cascading (usually) adds clarity and produces more elegant scripts. In our case, since variable `f` isn't really important to us, we discard it after we attach it to a `csv.reader` object; so instead of the preceding code, we can write the following:

```
>>> cr = csv.reader(open('data/CB401-2005-06-21-013504.csv'))
>>> for row in cr:
...     print row
```

The same holds true for variable `cr`, so if we're feeling particularly brave, we can use this script:

```
>>> for row in csv.reader(open('data/CB401-2005-06-21-013504.csv')):
...     print row
```

While the script might be shorter, there's no performance gain. It is therefore suggested that you cascade functions only if it adds clarity; there's a good chance you'll be editing this code later on, and it's important to be able to understand what's going on. In fact, not cascading functions might be useful at times because you might need access to intermediate variables (such as `f` and `cr` in our case).

The `csv.reader` object converts each row we read into a row of fields, in the form of a list. That row is then appended to a list of rows, `data`. This is also the value returned by the function.

■ **Note** By now you've seen the dot symbol (`.`) used several times, and it might be a bit confusing, so an explanation is in order. The dot symbol is used to access function members of modules as well as function members of objects (classes). You've seen it in member functions of modules, such as `csv.reader()`, but also for objects, such as `f.read()`. In the latter, it means that the file object has a member function `read()` and that function is called to operate on variable `f`. To access these functions, we use the dot operator. We'll touch on this again in Chapter 3. Lastly, we use the ellipsis symbol (`...`) to denote line continuation when interactively entering commands in Python.

Analyzing GPS Data

Let's take a closer look at the GPS data.

- Each row seems to start with a text header stamp, beginning with the characters `$GP`.
- There are several header stamps, for example, `$GPGSA` and `$GPRMC`.
- Following the header are additional values, most of which are numeric.

Not being GPS savvy, I looked up the GPS format on the Internet. It turns out the format is known as NMEA 0183. NMEA stands for the National Marine Electronics Association; see `http://www.nmea.org` for more information. The NMEA 0183 data format is described at `http://www.gpsinformation.org/dale/nmea.htm`. There are a lot of header stamps in the format, and some might hold useful information for our task.

As mentioned earlier, several `$GP` header stamps appear in our data files, but which ones exactly are of relevance is a different question. First, it would be nice to know which header stamps from the NMEA standard are even present in our data files. One option would be to open the files, look for the headers, and jot down every new header once we see it. Another, of course, would be to use Python to do that for us.

Python is a very high-level programming language. As such, it has built-in support for *dictionaries* (also known as associative arrays in Perl), which are data structures that have a one-to-one relationship between a key and a value, very much like real dictionaries. Traditional dictionaries, however, often have several values for a key, that is, several interpretations (values) for one word (key). You can easily implement this in Python's using the dictionary object as well by assigning a list value to a key. That way you can have several entries per one key, because the key is associated with a list that can hold several values. In reality, it's still a one-to-one relationship. But enough about that for now, I'll cover dictionaries in more detail in future chapters. What we want to do here is use a dictionary object to hold the number of times a header is encountered. Our key will be the GPS header stamp, and our value will be a number, indicating occurrence. We'll increment the value whenever a key is encountered, as shown in Listing 1-5.

Listing 1-5. *Function* `list_gps_commands()`

```
def list_gps_commands(data):
    """Counts the number of times a GPS command is observed.

Returns a dictionary object."""

    gps_cmds = dict()
    for row in data:
        try:
            gps_cmds[row[0]] += 1
        except KeyError:
            gps_cmds[row[0]] = 1

    return gps_cmds
```

Some notes about this function. First, the docstring spans multiple lines, which is one of the key benefits of docstrings. Docstrings will display all the spaces and line breaks as shown in the function itself. Next we initialize a variable, `gps_cmds`, to be our dictionary. We then process every list in the GPS data: we only care about the first element of every row, as that's the value that holds the GPS header stamps. We then increment the value associated with the key: `gps_cmds[row[0]] += 1`. We use the `+=` operation to increment the value by 1, similar to how it's done in C (Python, however, does not use the `++` operator). If the key does not exist, which will happen whenever we encounter a new header stamp, an exception will be raised. We

catch the exception with our `except KeyError` statement. In case of an exception, we set the dictionary value associated with the key to `1`.

The function `list_gps_commands()` can be written even more compactly using the dictionary method `get()`; see Chapter 3 for details.

Let's analyze some GPS data:

```
>>> x = read_csv_file('../data/GPS-2008-05-30-09-00-50.csv')
>>> list_gps_commands(x)
{'$GPGSA': 282, '$GPGSV': 846, '$GPGGA': 282, '$GPRMC': 283}
```

Turns out there are four distinct GPS headers being generated by my GPS. Of those, only two interest me: `$GPGSV`, which holds the number of satellites in view (Hey! It's really important!), and `$GPRMC`, which holds location and velocity information.

So what we'd like to do is code a function that takes the GPS data and, whenever the header field is `$GPGSV` or `$GPRMC`, extracts the information and stores it in numerical arrays that will be easier to manipulate later on. Numerical arrays are introduced with the NumPy module, so we have to issue `import numpy`. Since we'll be using a lot of the functionality of NumPy, SciPy, and matplotlib, an easier approach would be to issue `import pylab`, which imports all these modules, as follows:

```
>>> from pylab import *
```

■ **Note** The name PyLab comes from Python and MATLAB. PyLab provides MATLAB-like functionality in Python.

Extracting GPS Data

In the case of a `$GPGSV` header, the number of satellites is the fourth entry. In case of a `$GPRMC` header, we have a bit more interesting information. The second field is the timestamp, the fourth field is the latitude, the sixth field is the longitude, and the eighth field is the velocity. Again, turn to the NMEA 0183 format for more details. Table 1-1 summarizes the fields and their values in a `$GPRMC` line.

Table 1-1. *$GPRMC Information (Excerpt)*

Field Name	Index	Format
Header	0	$GPRMC (fixed)
Timestamp	1	hhmmss.ss
Latitude	3	DDMM.MMM
Longitude	5	DDDMM.MMM
Velocity	7	VVV.V

Some caveats regarding the information in `$GPRMC`. We first turn to the timestamp of an arbitrary line:

```
>>> x[12]
['$GPRMC', '140055.00', 'A', '4454.1740', 'N', '09325.0143', 'W', '000.0', '128.7',
'300508', '001.1', 'E', 'A*28']
```

In this output, the timestamp appears as `'140055.00'`. This follows the format hhmmss.ss where hh are two digits representing the hour (it will always consist of two digits—if the hour is one digit, say 7 in the morning, a 0 will be added before it), mm are two digits representing the minute (again, always two digits), and ss.ss are five characters (four digits plus the dot) representing seconds and fractions of seconds. (There's also a North/South field as well as an East/West field. Here, for simplicity, we assume northern hemisphere, but you can easily change these values by reading the entire `$GPRMC` structure.)

■ **Note** In the ISO time format, we've used HHMMSS to denote hours minutes and seconds. Here we follow the convention in NMEA, which uses hhmmss.ss for hours, minutes, and seconds and sets DD and MM to angular degrees and minutes.

The timestamp string is a bit hard to work with, especially when plotting data. The first reason is that it's a string, not a number. But even if you translated it to a number, the system does not lend itself nicely to plotting because there are 60 seconds in a minute, not a 100. So what we want to do is "linearize" the timestamp. To achieve this, we translate the timestamp as seconds elapsed since midnight, as follows: T = hh * 3600 + mm * 60 + ss.ss.

The second issue we have is that hh, mm, and ss.ss are strings, not numbers. Multiplying a string in Python does something completely different from what we want here. So we have to first convert the strings to numerical values, in our case, `float`, because of the decimal point in the string representing the seconds. This all folds nicely into the following:

```
>>> row = x[12]
['$GPRMC', '140055.00', 'A', '4454.1740', 'N', '09325.0143', 'W', '000.0', '128.7',
'300508', '001.1', 'E', 'A*28']
>>> float(row[1][0:2])*3600+float(row[1][2:4])*60+float(row[1][4:6])
50445.0
```

The operator `[]` denotes the index, so `row[1]` is the second field of `row` (counting starts at zero) which is a string. The first two characters of a string are denoted by `[0:2]`; this is known as *string slicing*. So to access the first two characters of the first field, we write `row[1][0:2]`. Upcoming chapters will include more about strings and methods of slicing them.

Next we tackle latitude and longitude. We face the same issue as with the timestamp, only here we deal with degrees. Latitude follows the format DDMM.MMM where DD stands for degrees and MM.MMM stands for minutes. We decide to use degrees this time. To translate the latitude into decimal degrees, we need to divide the minutes by 60:

```
>>> row = x[12]
['$GPRMC', '140055.00', 'A', '4454.1740', 'N', '09325.0143', 'W', '000.0', '128.7',
'300508', '001.1', 'E', 'A*28']
>>> float(row[3][0:2])+float(row[3][2:])/60.0
44.902900000000002
```

For latitude information we require the fourth field, hence `row[3]`. This example also introduces another notation, `[2:]`, which means the slice of the string from the third character until the end. Also notice that the code uses `60.0` and not `60`. When dividing by 60, it's implied that you want an integer division; dividing by 60.0 means you want a floating-point division, which is to say you care about the information past the decimal point. However, seeing as we already specified that we want the information as a floating-point number as indicated by the `float()` conversion, the result will be a floating point regardless. Still, it's good practice to let Python know what kind of division you really want.

Here are some examples to further illustrate the point:

```
>>> 100/60
1
>>> 100/60.0
1.6666666666666667
>>> float(100)/60
1.6666666666666667
```

Longitude information is similar to latitude with a minor difference: longitude degrees are three characters instead of two (up to 180 degrees, not just up to 90 degrees) so the indices to the strings are different.

Listing 1-6 presents the entire function to process GPS data.

Listing 1-6. *Function process_gps_data()*

```
from pylab import *

# constant definitions
NMI = 1852.0

def process_gps_data(data):
    """Processes GPS data, NMEA 0183 format.

Returns a tuple of arrays: latitude, longitude, velocity [km/h],
time [sec] and number of satellites.
See also: http://www.gpsinformation.org/dale/nmea.htm."""

    latitude    = []
    longitude   = []
    velocity    = []
    t_seconds   = []
    num_sats    = []

    for row in data:
        if row[0] == '$GPGSV':
            num_sats.append(float(row[3]))
        elif row[0] == '$GPRMC':
            t_seconds.append(float(row[1][0:2])*3600 + \
                float(row[1][2:4])*60+float(row[1][4:6]))
            latitude.append(float(row[3][0:2]) + \
```

```
            float(row[3][2:])/60.0)
        longitude.append((float(row[5][0:3]) + \
            float(row[5][3:])/60.0))
        velocity.append(float(row[7])*NMI/1000.0)

    return (array(latitude), array(longitude), \
        array(velocity), array(t_seconds), array(num_sats))
```

Some notes about the `process_gps_data()` function:

- `NMI` is defined as `1852.0`, which is one nautical mile in meters and also one minute on the equator. The reason the constant `NMI` is not defined in the function is that we'd like to use it outside the function as well.
- We initialize the return values `latitude`, `longitude`, `velocity`, `t_seconds`, and `num_sats` by setting them to an empty list: `[]`. Initializing the lists creates them and allows us to use the `append()` method, which adds values to the lists.
- The `if` and `elif` statements are self-explanatory: `if` is a conditional clause, and `elif` is equivalent to saying "else, if." That is, if the first condition didn't succeed, but the next condition succeeds, execute the following block.
- The symbol `\` that appears on the several calculations and on the `return` line indicates that the operation continues on the next line.
- Lastly, the return value is a tuple of arrays. A *tuple* is an immutable sequence, meaning you cannot change it. So tuple means an unchangeable sequence of items (as opposed to a list, which is a mutable sequence). The reason we return a tuple and not a two-dimensional array, for example, is that we might have different lengths of lists to return: the length of the number of satellites list may be different from the length of the longitude list, since they originated from different header stamps.

Here's how you call `process_gps_data()`:

```
>>> y = read_csv_file('../data\\GPS-2008-05-30-09-00-50.csv')
>>> (lat, long, v, t, sats) = process_gps_data(y)
```

The second line introduces sequence unpacking, which allows multiple assignments. Armed with all these functions, we're ready to plot some data!

Data Visualization

Our next step is to visualize the data. We'll be relying on the matplotlib package heavily. We've already imported matplotlib with the command `from pylab import *`, so there's no additional importing needed at the moment. It's time to read the data and plot the course.

Our first problem is that the information is given in latitude and longitude. Latitude and longitude are spherical coordinates, that is, those are points on a sphere, the earth. But we want a map-like plot, which uses Cartesian coordinates, that is, x and y. So first we have to transform the spherical coordinates to Cartesian. We'll use the quick-and-dirty method shown in Listing 1-7 to do this, one that's actually quite accurate as long as the distances traveled are small relative to the radius of the earth.

Listing 1-7. *"Quick-and-Dirty" Spherical to Cartesian Transformation*

```
x = longitude*NMI*60.0*cos(latitude)
y = latitude*NMI*60.0
```

To justify this to yourself, consider the following reasoning: As you go up to the North Pole, the circumference at the location you're at gets smaller and smaller, until at the North Pole it's zero. So at latitude 0º, the equator, each degree (longitude) means more distance traveled than at latitude 45º. That's why x is a function of the longitude value itself but also of the latitude: the greater the latitude, the smaller a longitude change is in terms of distance. On the other hand, y, which is north to south, is not dependent on longitude.

The next thing to understand is that the earth is a sphere, and whenever we plot an x-y map, we're only really plotting a projection of that sphere on a plane of our choosing, hence we denote it by (px,py), where p stands for "projection." We'll take the southeastern-most point as the start of the GPS data projection: (px,py) = (0,0). This translates into the code shown in Listing 1-8.

Listing 1-8. *Projecting the Traveled Course to Cartesian Coordinates*

```
py = (lat-min(latitude))*NMI*60.0
px = (long-min(longitude))*NMI*60.0*cos(D2R*latitude)
```

Some things to note:

- Variables py and px are arrays of floating-point values. We now operate on entire arrays seamlessly. This is part of the NumPy package.
- D2R is a constant equal to π/180, converting degrees to radians.
- To set the y-axis at the minimum latitude and the x-axis at the minimum longitude, we subtract the minimum latitude and minimum longitude values from latitude and longitude values, respectively.

GPS Location Plot

Now the moment we've been waiting for, plotting GPS data. To be able to follow along and plot data, be sure to define the functions read_csv_file() and process_gps_data() as previously detailed and set the file name variable to point to your GPS data file. I've suppressed matplotlib responses so that the code is cleaner to follow.

```
>>> filename = 'GPS-2008-05-30-09-00-50.csv'
>>> y = read_csv_file('../data/'+filename)
>>> (lat, long, v, t, sats) = process_gps_data(y)
>>> px = (long-min(long))*NMI*60.0*cos(D2R*lat)
>>> py = (lat-min(lat))*NMI*60.0
>>> figure()
>>> gca().axes.invert_xaxis()
>>> plot(px, py, 'b', label='Cruising', linewidth=3)
>>> title(filename[:-4])
>>> legend(loc='upper left')
>>> xlabel('east-west (meters)')
```

```
>>> ylabel('south-north (meters)')
>>> grid()
>>> axis('equal')
>>> show()
```

Figure 1-1 shows the result, which is rather pleasing.

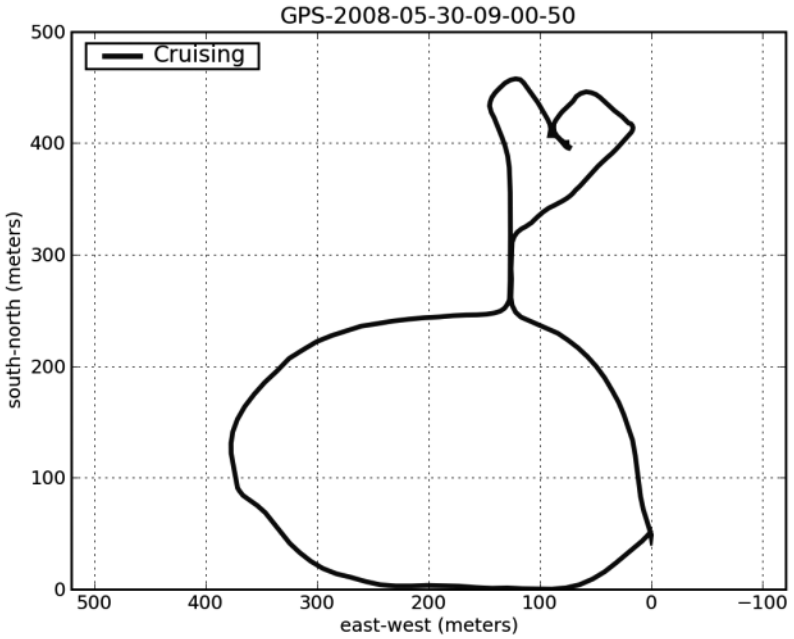

Figure 1-1. *GPS data*

We've used a substantial number of new functions, all part of the matplotlib package: `plot()`, `grid()`, `xlabel()`, `legend()`, and more. Most of them are self-explanatory:

- `xlabel(string_value)` and `ylabel(string_value)` will print a label on the x- and y-axis, respectively. `title(string_value)` is used to print a caption above the graph. The string value in the title is the file name up to the end minus four characters (so as to not display ".csv"). This is done using string slicing with a negative value, which means "from the end."
- `legend()` prints the labels associated with the graph in a legend box. `legend()` is highly configurable (see `help(legend)` for details). The example plots the legend at the top-left corner.
- `grid()` plots the grid lines. You can control the behavior of the grid quite extensively.
- `plot()` requires additional explanation as it is the most versatile. The command `plot(px, py, 'b', label='Cruising', linewidth=3)` plots `px` and `py` with the color blue as specified by the character `'b'`. The plot is labeled "Cruising" so later on, when we call the `legend()` function, the proper text will be associated with the data. Finally, we set the line width to 3.

- The function `axis()` controls the behavior of the graph axis. Normally, I don't call the `axis()` function because `plot()` does a decent job at selecting the right values. However, in this case, it's important to visualize the data properly, and that means to have both x- and y-axes with equal increments so the graph is true to the path depicted. This is achieved by calling `axis('equal')`. There are other values to control axis behavior as described by `help(axis)`.
- Lastly, `gca().axes.invert_xaxis()` is a rather exotic addition. It stems from the way we like to view maps and directions. In longitude, increasing values are displayed from right to left. However, in mathematical graphs, increasing values are typically displayed from left to right. This function call instructs the x-axis to be incrementing from right to left, just like maps.
- When you're done preparing the graph, calling the `show()` function displays the output.

Matplotlib, which includes the preceding functions, is a comprehensive plotting package and will be explored in Chapter 6.

Annotating the Graph

We'd like to add some more information to the GPS graph: we'd like to know where we've stopped and where we were speeding. For this we use the function `find()`, which is part of the PyLab package. `find()` returns an array of indices that satisfy the condition, in our case:

```
>>> STANDING_KMH = 10.0
>>> SPEEDING_KMH = 50.0
>>> Istand = find(v < STANDING_KMH)
>>> Ispeed = find(v > SPEEDING_KMH)
>>> Icruise = find((v >= STANDING_KMH) & (v <= SPEEDING_KMH))
```

We also calculate when we're cruising (i.e., not speeding nor standing) for future processing.

To annotate the graph with these points, we add another plot on top of our current plot, only this time we change the color of the plot, and we use symbols instead of a solid blue line. The combination `'sg'` indicates a green square symbol (g for green, s for square); the combination `'or'` indicates a red circle (r for red, o for circle). I suggest you use different symbols for standing and speeding, not just colors, because the graph might be printed on a monochrome printer. The function `plot()` supports an assortment of symbols and colors; consult with the interactive help for details. The values we plot are only those returned by the `find()` function.

```
>>> plot(px[Istand], py[Istand], 'sg', label='Standing')
>>> plot(px[Ispeed], py[Ispeed], 'or', label='Speeding!')
>>> legend(loc='upper left')
```

Figure 1-2 shows the outcome.

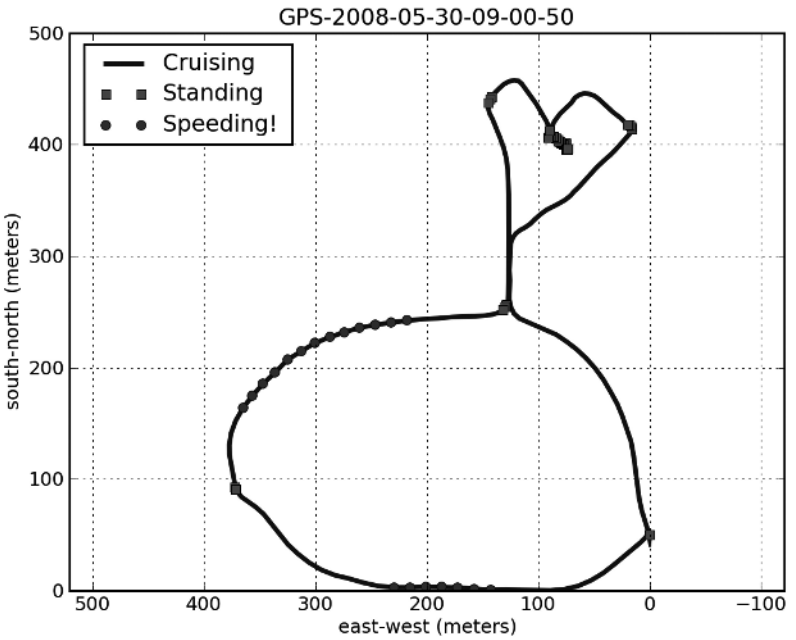

Figure 1-2. *GPS data with additional speed information*

We'd also like to know the direction the car is going. To implement this, we'll use the `text()` function, which allows the writing of a string to an arbitrary location in the graph. So to add the text "Hi" at location (10, 10), issue the command `text(10, 10, 'Hi')`. One of the nice features of the `text()` function is that you can rotate the text at an arbitrary angle. So to plot "Hi" at location (10, 10) at 45 degrees, you issue `text(10, 10, 'Hi', rotation=45)`. Our implementation of heading information involves rotating the text ">>>" at the angle the car is heading. We'll only do this ten times so as not to clutter the graph with ">" symbols. Calculating the direction the car is heading at a given point, `i`, is shown in Listing 1-9.

Listing 1-9. *Calculating the Heading*

```
dx = px[i+1]-px[i]
dy = py[i+1]-py[i]
heading = arctan(dy/dx)
```

Instead of actually using the function `arctan(dy/dx)`, we'll use the function `arctan2(dy, dx)`. The benefits of using `arctan2()` over `arctan()` are twofold: 1) there's no division that might cause a divide-by-zero exception in case `dx` is zero, and 2) `arctan2()` preserves the angle from –180 degrees to 180 degrees, whereas `arctan()` produces values between 0 degrees and 180 degrees only. The following code adds the direction symbols:

```
>>> for i in range(0, len(v), len(v)/10-1):
...     text(px[i], py[i], ">>>", \
...         rotation = arctan2(py[i+1]-py[i], -(px[i+1]-px[i]))/D2R, \
...         ha='center')
```

Figure 1-3 shows the resulting graph.

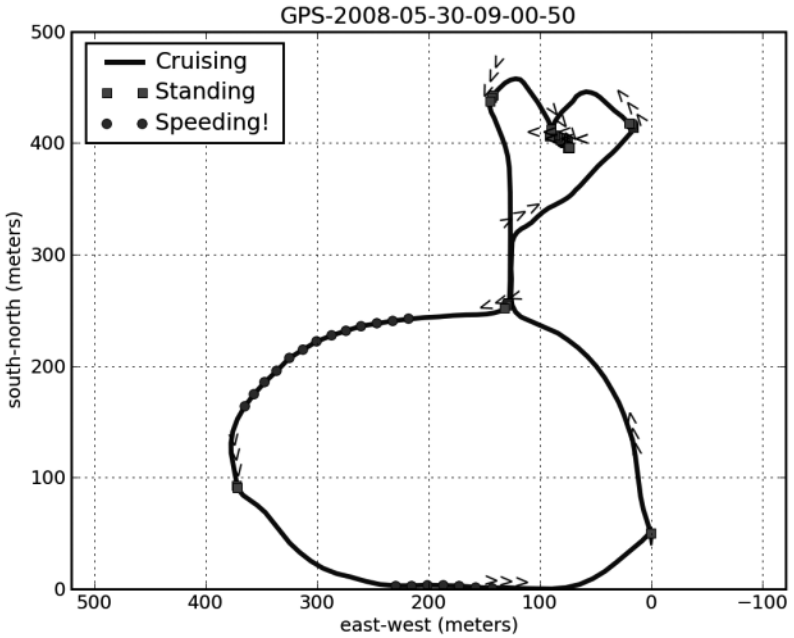

Figure 1-3. *GPS graph with heading*

Velocity Plot

We now turn to plotting a graph of the speed. This is a lot simpler:

```
>>> figure()
>>> t = (t-t[0])/60.0
>>> plot(t, v, 'k')
>>> plot([t[0], t[-1]], [STANDING_KMH, STANDING_KMH], '-g')
>>> text(t[0], STANDING_KMH, \
...     " Standing threshold: "+str(STANDING_KMH))
>>> plot([t[0], t[-1]], [SPEEDING_KMH, SPEEDING_KMH], '-r')
>>> text(t[0], SPEEDING_KMH, \
...     " Speeding threshold: "+str(SPEEDING_KMH))
>>> grid()
>>> title('Velocity')
>>> xlabel('Time from start of file (minutes)')
>>> ylabel('Speed (Km/H)')
```

We start by opening a different figure with the `figure()` command. We proceed by changing the timescale units to minutes, a value easier for most humans to follow than seconds. Selecting the proper units of measurement is important. Most people will find it easier to follow the sentence "I drove for 30 minutes" as opposed to "I drove for 1800 seconds." We also set the time axis to start at `t[0]`. Next we plot the velocity as a function of time, in black. Good

graphs require annotation, so we choose to add two lines describing the thresholds for standing and speeding as well as text describing those thresholds. To generate the text, we combine the text "Standing threshold" with the threshold value (after casting it to a string) and use the + operator to concatenate strings. Last, of course, are the title, x and y labels, and grid. Figure 1-4 shows the final result.

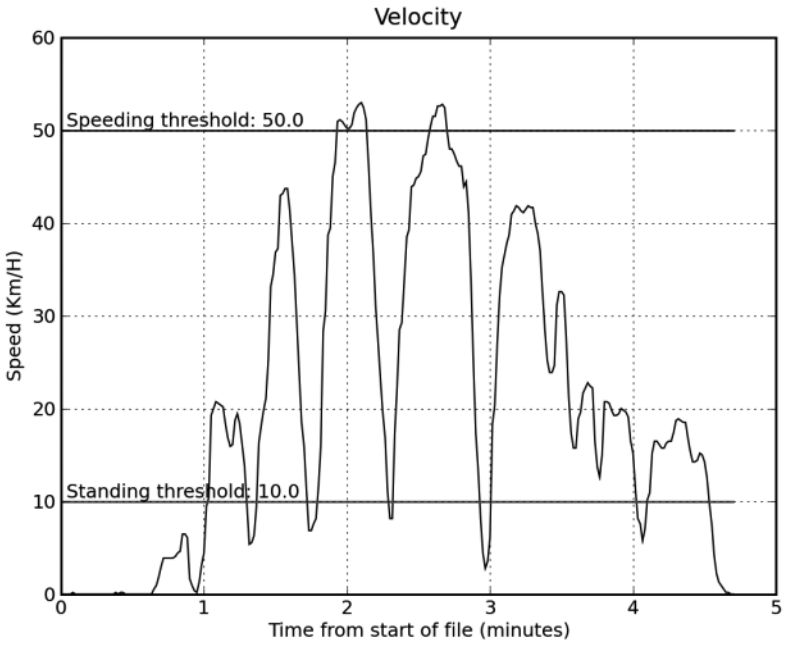

Figure 1-4. *Velocity over time*

Subplots

We'd also like to display some statistics. But before we do that, it would be preferable to combine all these plots (GPS, velocity, and statistics) into one figure. To do this, we use the `subplot()` function. `subplot()` is a matplotlib function that divides the plot into several smaller sections called subplots and selects the subplot to work with. For example, `subplot(1, 2, 1)` informs subsequent plotting commands that the area to work on is 1 by 2 subplots and the currently selected subplot is 1, so that's the left side of the plot area. `subplot(2, 2, 2)` will choose the top-right subplot; `subplot(2, 2, 4)` will choose the lower-right subplot. A selection I found most readable in this scenario is to have the GPS data take half of the plot area, the velocity graph a quarter, and the statistics another quarter.

Text

Sometimes, the best way to convey information is using text, not graphics. We'll be limiting our work to the statistics quarter for this section. Our first task is to get rid of the plot frame and the x and y ticks. We just want a plain canvas to display text on. This is achieved by issuing the following:

```
>>> subplot(2, 2, 4)
>>> axis('off')
```

The first call to `subplot()` selects our region of work as the lower-right quarter. The second line removes the axes and hides the frame box.

It's time to calculate some statistics. It appears that GPS data is being sent in regular intervals, typically one second. So to calculate the time spent standing, in seconds, we calculate the length of the vector `Istand`. Likewise, to calculate the time speeding, we can calculate the length of `Ispeed`. To estimate how much these were in percent values, we divide the length of the `Istand` and `Ispeed` vectors by the length of the velocity vector and multiply by 100. To calculate the average speed, we use the `mean()` function, which is part of PyLab.

We also would like to calculate the total distance traveled. The distance can be calculated as the sum of the distances between each two consecutive data points. The function `diff()` returns a vector of the differences of the input vector.

```
>>> diff([1, 4, 0, 2])
array([ 3, -4,  2])
```

This is really useful because now to calculate the distance we can do the following:

```
>>> sum(sqrt(diff(px)**2+diff(py)**2))
1652.1444099624528
```

which in turn yields the total distance traveled.

To automate the whole process of printing the statistics, we store the text to be printed in the variable `stats`, a list of strings. We also use a method of formatting strings similar to C's `printf()` function, although the syntax is a bit different. `%s` indicates a string; the `%f` indicates a floating point number, in our case `%.1f` indicates a `float` with one digit after the decimal point; and `%d` indicates an integer. The following generates the statistics text:

```
>>> Total_distance  = float(sum(sqrt(diff(px)**2+diff(py)**2))/1000.0)
>>> Stand_time  = len(Istand)/60.0
>>> Cruise_time = len(Icruise)/60.0
>>> Speed_time  = len(Ispeed)/60.0
>>> Stand_per   = 100*len(Istand)/len(v)
>>> Cruise_per  = 100*len(Icruise)/len(v)
>>> Speed_per   = 100*len(Ispeed)/len(v)
>>> stats=['Statistics', \
...    '%s' % filename, \
...    'Number of data points: %d' % len(y), \
...    'Average number of satellites: %d' % mean(sats), \
...    'Total driving time: %.1f minutes:' % (len(v)/60.0), \
...    '    Standing: %.1f minutes (%d%%)' % \
...    (Stand_time, Stand_per), \
...    '    Cruising: %.1f minutes (%d%%)' % \
...    (Cruise_time, Cruise_per), \
...    '    Speeding: %.1f minutes (%d%%)' % \
...    (Speed_time, Speed_per), \
...    'Average speed: %d km/h' % mean(v), \
...    'Total distance traveled: %.1f Km' % Total_distance ]
```

To print the text on the canvas, we again use the `text()` function, in a `for` loop, iterating over every string of the `stats` list.

```
>>> for index, stat_line in enumerate(reversed(stats)):
...    text(0, index, stat_line, va='bottom')
...
>>> plot([index-.2, index-.2])
>>> axis([0, 1, -1, len(stats)])
```

We've introduced two new functions. One is `reversed()`, which yields the elements of `stats`, in reversed order. The second is `enumerate()`, which returns not just each row in the `stats` array but also the index to each row. So when variable `stat_line` is assigned the value `'Average speed...'`, the variable `index` is assigned the value `8`, which indicates the ninth row in `stats`. The reason we want to know the index is that we use it as location on the y-axis. Lastly, the vertical alignment of the text is selected as bottom as suggested by the parameter `va='bottom'` (`va` is short for vertical alignment).

Tying It All Together

Finally, Listing 1-10 shows the combined code to analyze and plot all GPS files in directory `data`.

Listing 1-10. *Script `gps.py`*

```
from pylab import *
import csv, os

# constant definitions
STANDING_KMH = 10.0
SPEEDING_KMH = 50.0
NMI = 1852.0
D2R = pi/180.0

def read_csv_file(filename):
    """Reads a CSV file and returns it as a list of rows."""

    data = []
    for row in csv.reader(open(filename)):
        data.append(row)
    return data


def process_gps_data(data):
    """Processes GPS data, NMEA 0183 format.

Returns a tuple of arrays: latitude, longitude, velocity [km/h],
time [sec] and number of satellites.
See also: http://www.gpsinformation.org/dale/nmea.htm.
    """
```

```
    latitude    = []
    longitude   = []
    velocity    = []
    t_seconds   = []
    num_sats    = []

    for row in data:
        if row[0] == '$GPGSV':
            num_sats.append(float(row[3]))
        elif row[0] == '$GPRMC':
            t_seconds.append(float(row[1][0:2])*3600 + \
                float(row[1][2:4])*60+float(row[1][4:6]))
            latitude.append(float(row[3][0:2]) + \
                float(row[3][2:])/60.0)
            longitude.append((float(row[5][0:3]) + \
                float(row[5][3:])/60.0))
            velocity.append(float(row[7])*NMI/1000.0)

    return (array(latitude), array(longitude), \
        array(velocity), array(t_seconds), array(num_sats))


# read every data file, filter, and plot the data
for root, dirs, files in os.walk('../data'):
    for filename in files:
        # create full file name including path
        cur_file = os.path.join(root, filename)
        if filename.endswith('csv'):
            y = read_csv_file(cur_file)
        else:
            continue

        # only files with the .csv extension from here on

        # process GPS data
        (lat, long, v, t, sats) = process_gps_data(y)

        # translate spherical coordinates to Cartesian
        py = (lat-min(lat))*NMI*60.0
        px = (long-min(long))*NMI*60.0*cos(D2R*lat)

        # find out when standing, speeding, or cruising
        Istand = find(v < STANDING_KMH)
        Ispeed = find(v > SPEEDING_KMH)
        Icruise = find((v >= STANDING_KMH) & (v <= SPEEDING_KMH))
```

```
# left side, GPS location graph
figure()
subplot(1, 2, 1)

# longitude values go from right to left,
# we want increasing values from left to right
gca().axes.invert_xaxis()

plot(px, py, 'b', label=' Cruising', linewidth=3)
plot(px[Istand], py[Istand], 'sg', label=' Standing')
plot(px[Ispeed], py[Ispeed], 'or', label=' Speeding!')

# add direction of travel
for i in range(0, len(v), len(v)/10-1):
    text(px[i], py[i], ">>>", \
        rotation = arctan2(py[i+1]-py[i], \
            -(px[i+1]-px[i]))/D2R, ha='center')

# legends and labels
title(filename[:-4])
legend(loc='upper left')
xlabel('east-west (meters)')
ylabel('south-north (meters)')
grid()
axis('equal')

# top-right corner, speed graph
subplot(2, 2, 2)

# set the start time as t[0]; convert to minutes
t = (t-t[0])/60.0
plot(t, v, 'k')

# plot the standing and speeding threshold lines
plot([t[0], t[-1]], [STANDING_KMH, STANDING_KMH], '-g')
text(t[0], STANDING_KMH, \
    " Standing threshold: "+str(STANDING_KMH))
plot([t[0], t[-1]], [SPEEDING_KMH, SPEEDING_KMH], '-r')
text(t[0], SPEEDING_KMH, \
    " Speeding threshold: "+str(SPEEDING_KMH))
grid()

# legend and labels
title('Velocity')
xlabel('Time from start of file (minutes)')
ylabel('Speed (Km/H)')
```

```
        # right-side corner, statistics data
        subplot(2, 2, 4)

        # remove the frame and x-/y-axes. we want a clean slate
        axis('off')

        # generate an array of strings to be printed
        Total_distance  = float(sum(sqrt(diff(px)**2+diff(py)**2)) \
            /1000.0)
        Stand_time  = len(Istand)/60.0
        Cruise_time = len(Icruise)/60.0
        Speed_time  = len(Ispeed)/60.0
        Stand_per   = 100*len(Istand)/len(v)
        Cruise_per  = 100*len(Icruise)/len(v)
        Speed_per   = 100*len(Ispeed)/len(v)
        stats=['Statistics', \
        '%s' % filename, \
        'Number of data points: %d' % len(y), \
        'Average number of satellites: %d' % mean(sats), \
        'Total driving time: %.1f minutes:' % (len(v)/60.0), \
        '    Standing: %.1f minutes (%d%%)' % \
        (Stand_time, Stand_per), \
        '    Cruising: %.1f minutes (%d%%)' % \
        (Cruise_time, Cruise_per), \
        '    Speeding: %.1f minutes (%d%%)' % \
        (Speed_time, Speed_per), \
        'Average speed: %d km/h' % mean(v), \
        'Total distance traveled: %.1f Km' % Total_distance ]

        # display statistics information
        for index, stat_line in enumerate(reversed(stats)):
            text(0, index, stat_line, va='bottom')

        # draw a line below the "Statistics" text
        plot([index-.2, index-.2])

        # set axis properly so all the text is displayed
        axis([0, 1, -1, len(stats)])
show()
```

Figure 1-5 shows the final results.

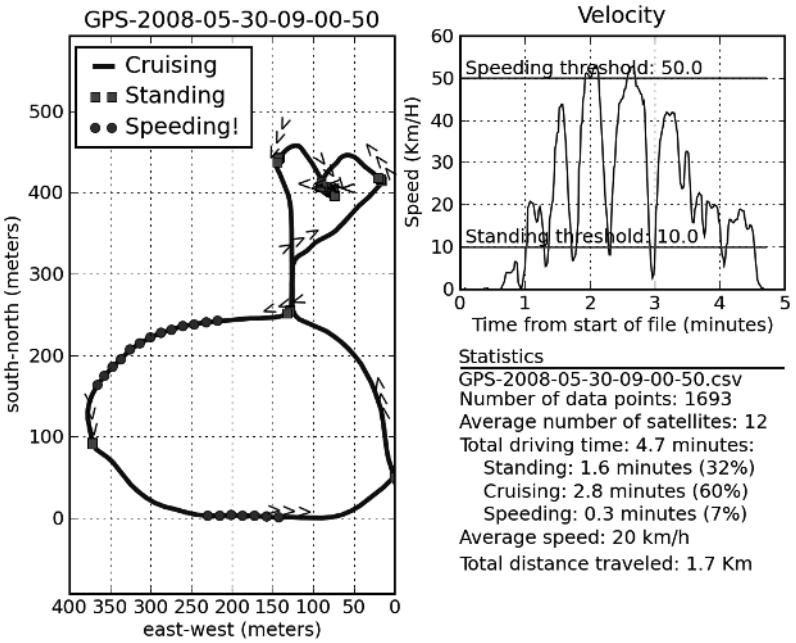

Figure 1-5. *Output of gps.py on some GPS data*

Final Notes and References

The GPS problem described here is research in nature: a computation, an intermediate result, not an end product. Research, or R&D work, especially feasibility studies, requires rapid responses. This means using readily available tools as much as possible and combining them to get the job done. If those tools are inexpensive, or free, that's yet another reason to use them.

Throughout the book, we will examine different packages and modules and see how they may be used to perform data analysis and visualization. The theme we'll be using is open software, including software published under the GNU Public License (GPL) and the Python Software Foundation (PSF) license. Examples of these tools include GNU/Linux and, of course, Python.

There are several benefits to developing data analysis and visualization scripts in Python:

- Developing and writing code is quick, appealing for research work.
- Readily available packages further increase productivity and ensure accurate results.
- Scripts introduce automation. Modifying an algorithm is easily done.

Scripts will be numerous and explained in detail, and I aim to cover most of the issues you are likely to encounter in the real world. Examples include scripts to deal with binary files, to combine data from different sources, to perform text parsing, to use high-level numerical algorithms, and much more. Scripts will be written in Python: some will be simple one-liners, others more complex. Special attention will be given to data visualization and how to achieve pleasing results in Python.

If you'd like to read more about Python in general (and not necessarily for data analysis and visualization), the Python official web site is an excellent resource:

- Python Programming Language—Official Website, `http://www.python.org`