Introduction to Data Analytics in Accounting using Python

Stephen Perreault, PhD, CPA

© 2017 Stephen J. Perreault

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Version 0.1.0: August 2017

Contents

1 Preface **1**
- 1.1 Who this book is for . . . 1
- 1.2 Required hardware and software . . . 2
- 1.3 Comments and feedback . . . 2

2 Version control **3**
- 2.1 Why you should care about version control . . . 3
- 2.2 Introducing Git . . . 4
- 2.3 Introducing GitHub . . . 5
- 2.4 Installing Git . . . 6
- 2.5 Creating your first repository . . . 7
- 2.6 Adding files to your repository . . . 10
- 2.7 Pushing commits to GitHub . . . 12
- 2.8 Reverting a commit . . . 14
- 2.9 Cloning a repository . . . 17
- 2.10 Advanced: Creating branches . . . 17
- 2.11 Advanced: Merging branches . . . 19
- 2.12 Summary . . . 20
- 2.13 Exercises . . . 20

3 Basic Python concepts **23**
- 3.1 Why Python? . . . 23
- 3.2 Setting up a development environment . . . 24
- 3.3 Your first program: “Hello World!” . . . 26
- 3.4 An example development workflow . . . 29
- 3.5 Mathematical operations . . . 30
- 3.6 Commenting your code . . . 30
- 3.7 Variables . . . 31
- 3.8 Strings . . . 32
- 3.9 String indexing . . . 35
- 3.10 Other basic data types . . . 36
- 3.11 Accepting user input . . . 36
- 3.12 Comparison operators . . . 37
- 3.13 Conditional statements . . . 38

3.14 Logical operators . . . 39
3.15 Iteration . . . 39
3.16 Searching for help . . . 42
3.17 Summary . . . 42
3.18 Exercises . . . 42

4 Intermediate Python concepts **47**
4.1 Introducing functions . . . 47
4.2 Global versus local variables . . . 48
4.3 Lists . . . 49
4.4 Dictionaries . . . 51
4.5 File input and output . . . 53
4.6 Reading and writing data using JSON . . . 55
4.7 Advanced: Classes . . . 57
4.8 Advanced: Inheritance . . . 61
4.9 Summary . . . 63
4.10 Exercises . . . 63

5 Collecting data from the web **67**
5.1 A brief overview of HTML . . . 67
5.2 Necessary libraries . . . 68
5.3 Viewing a webpage's HTML code . . . 70
5.4 Building a basic webscraper using Python . . . 71
5.5 Scraping multi-page websites . . . 76
5.6 Summary . . . 79
5.7 Discussion Questions . . . 79
5.8 Exercises . . . 80

6 Working with tabular data **81**
6.1 Installing Pandas . . . 81
6.2 Constructing data frames manually . . . 82
6.3 Constructing data frames from .CSV . . . 82
6.4 Printing data frames . . . 83
6.5 Changing frame indices . . . 83
6.6 Sorting data . . . 84
6.7 Arithmetic operations . . . 84
6.8 Aggregating data . . . 85
6.9 Merging data frames . . . 86
6.10 Deleting data and removing duplicates . . . 87
6.11 Descriptive statistics . . . 89
6.12 Cross-tabulation . . . 90
6.13 Summary . . . 92
6.14 Exercises . . . 92

7 Generalized linear models and forecasting 95
7.1 Preparing our data . . . 95
7.2 Creating a histogram . . . 99
7.3 Creating a scatterplot . . . 101
7.4 Installing scipy and statsmodels . . . 102
7.5 Correlation . . . 103
7.6 Linear regression . . . 104
7.7 Advanced: Logistic regression . . . 105
7.8 Advanced: Handling outliers . . . 107
7.9 Summary . . . 108
7.10 Exercises . . . 108

8 Plotting and visualization 109
8.1 Preparing our data . . . 109
8.2 Line plots . . . 110
8.3 Bar plots . . . 113
8.4 Pie chart . . . 115
8.5 Exploring other plot types . . . 116
8.6 Summary . . . 117
8.7 Exercises . . . 117

Chapter 1

Preface

1.1 Who this book is for

While data science classes are offered within the math and computer science departments at most colleges and universities, it can often be difficult for accounting students to enroll in such courses due to scheduling conflicts and extensive prerequisite requirements. As a result, accounting students often graduate with limited applied exposure to data science, despite the increasing need of the accounting profession for graduates with skills in this area. This textbook is designed to address this issue by providing a basic introduction to data manipulation, analysis, and Python programming that can be completed within the course of a single semester.

This book is meant to accompany the course *Introduction to Data Analytics in Accounting* taught at Providence College. It's audience is undergraduate accounting majors who understand basic financial accounting concepts and have some exposure to accounting information systems. The book also assumes a working knowledge of basic computer operations (installing programs, copying files, etc.), although no specific programming experience is required.

For those reading this book who are not students at Providence College, be advised that this text is not intended to be a stand-alone resource for learning basic data analysis. Rather, the book will be of most use as a supplemental resource for students enrolled in data science courses taught by knowledgable instructors who can fill in its many gaps.

1.2 Required hardware and software

Any PC capable of running a modern Windows operating system (Windows 7, 8, ot 10+) should be sufficient for following along with the book. I will assume that you are using a Windows-based PC, since this is currently the predominant operating system used in business computer labs. That being said, all of the software discussed in this text can also be installed on Mac and Linux operating systems, provided that the user is willing to read the online documentation for the various programs used.

The book has been written to use a very small number of commonly used software packages, which should be available in most campus computer labs. All are either free or have free versions available that are sufficient for completing the exercises contained within the book. The software that will be used is:

1. **Git**. An open source version control system for software development. Available at: `https://git-scm.com/downloads`.
2. **Anaconda**. An open source distribution of Python that is focused on data science. This distribution contains all of the scientific libraries that we will be using in the text. Available at: `https://continuum.io/downloads`.
3. **Notepad++**. A free source code editor and notepad replacement that supports formatting for the Python programming language. Available at: `https://notepad-plus-plus.org`.
4. **Microsoft Powershell**. A command line shell which comes pre-installed on all versions of Microsoft Windows starting with version 7.

Installation instructions for each of these software programs will be provided when the program is first introduced in the text.

1.3 Comments and feedback

This text is very much a work in progress and I would love to hear your comments, suggestions, and any errors you discover when working your way through the book. I can be reached at `sperreau@providence.edu`.

Happy programming!

Chapter 2

Version control

2.1 Why you should care about version control

Imagine that the audit engagement team which you have been assigned to is in need of a tool that can assess the valuation of an audit client's stock options using the Black-Scholes-Merton valuation model.[1] The tool needs to be simple to use, scalable, and compatible with a wide variety of operating systems. Knowing your reputation as a skilled programmer who works well under pressure, your supervisor has given you 24 hours to write the Python module to perform this assessment.

You spend the night hunched over your keyboard working frantically to complete the assigned task before the deadline. Your source code undergoes numerous changes as you squash bugs, improve performance, and add extra features that you believe the engagement team might find useful. As the sun begins to rise the next morning, you realize that you have successfully completed the task. You have developed a functional software tool that you are sure will impress your supervisor.

Before emailing the Python script to the engagement team, you decide to make some minor tweaks to your code in order to slightly improve the tool's performance. After making the seemingly harmless modification to your code, you test the script one final time and are horrified to discover that the change you made has caused an error which has rendered the program non-functional.

[1] It's not important to understand what the Black-Scholes-Merton model is to appreciate this example. However, if you're curious, BSM can be used, among other things, to determine the price of certain types of stock options.

Frantically, you scan through the script, hoping to identify the changes that you made which might have caused the program to break. You haphazardly remove a few lines of code that you think may be the culprit but that doesn't seem to do the trick - the program still crashes. In addition, this deletion causes another portion of the program to break. Feeling hopelessly lost, you begin to realize that you may be forced to start the project from scratch. What will you possibly tell your supervisor?

As the example above has hopefully demonstrated, it is incredibly important for programmers to keep track of the changes made to their source code over time. This process can seem daunting, especially for large projects with many different developers working on the code base simultaneously. Fortunately, modern developers can rely on version control systems to help manage these changes. If an error is introduced into the code, the developer can simply turn back the clock and revert to an earlier stable version.

Since we're going to be writing a significant amount of computer code as we work our way through this book, it makes sense to start by learning a bit about how version control works. In this chapter we will learn how to use version control to track changes to our individual projects. We will also learn how to create an online repository of our source code which can be shared with others and used as an online software development portfolio.

2.2 Introducing Git

In this book we will be using the widely used and popular version control system Git™. Git was created in 2005 by Linus Torvalds, the famous creator of the popular operating system Linux. In addition to providing developers with a history of all the individual changes that have been made to their projects, Git can also be used to track changes for developers working collaboratively.

As an example, let's say that you and a coworker have been assigned the task of writing a piece of software and will both need access to the source code during development. To facilitate this, you could post the code on a shared network drive or a file hosting service that you and your coworker can access. This method would probably work just fine unless the two of you are working on the same code file at the same time. If that happens, one of you would have your work overwritten and erased. Git can keep that from happening. If you and your coworker are both making changes to the same source code file, Git will save two copies. Later on, the changes can be merged together without losing any of the earlier work.

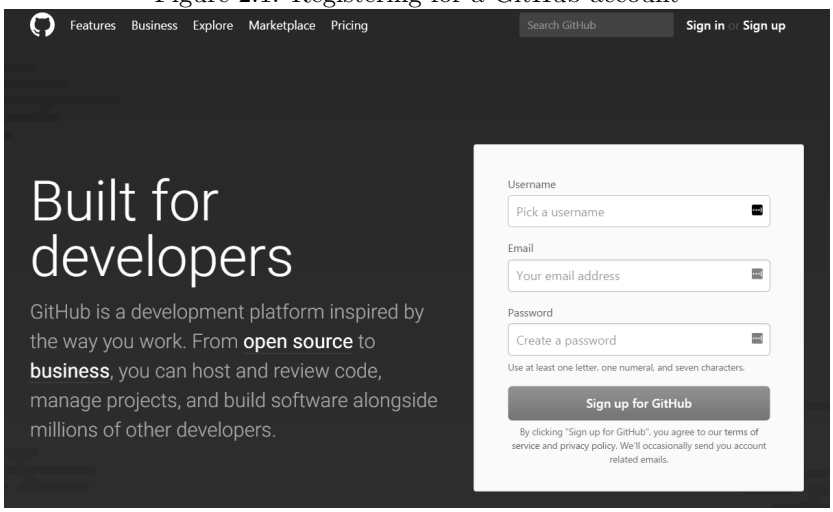
Unfortunately, Git has an reputation as being difficult for beginners to use

(which is undeserved in my humble opinion). This mostly stems from the fact that it does not have a graphical user interface; rather, users access the features of Git by typing in commands using the system terminal or command line. While there are GUI tools available for Git, working from the command line is an important skill set for an aspiring Python programmer to develop. As a result, we will be using the command line when working with Git in this book.

2.3 Introducing GitHub

While Git is handy version control system for working with projects locally on our computers, there are often times when we want to share our projects with others. To do this, we'll be using the online service GitHub™. GitHub is basically an online social media service; however, the primary type of information being shared on GitHub is source code as opposed to cat pictures![2] Throughout this book, we'll be using GitHub and Git in tandem in order to manage changes and share our projects with others.

Figure 2.1: Registering for a GitHub account



Although desktop clients are available, GitHub can be used without installing any software onto our computers. You simply have to sign up for a free GitHub account. Let's go ahead and do that now by visiting `https://github.com`. The sign up process is as easy as registering for any other social network. Note that you'll want to sign up for the free plan. GitHub also offers paid subscriptions

[2]This attempt at humor is actually not factually correct - projects posted on GitHub can contain images if the user wishes.

with more advanced features; however, these features our unnecessary for the purposes of this book.

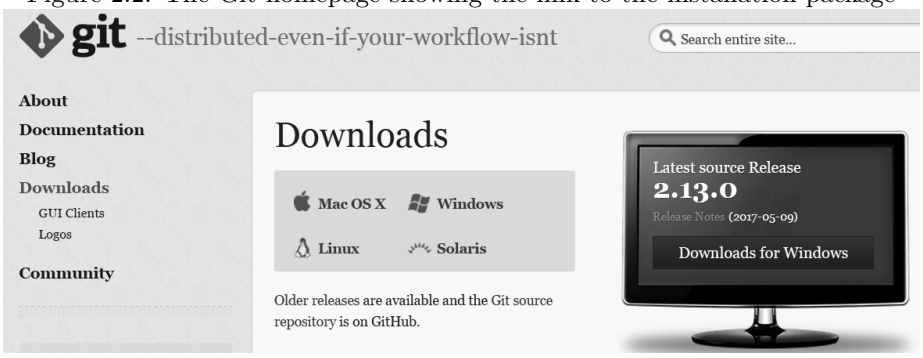
I encourage you to personalize your GitHub profile by uploading a recent picture of yourself. You may also wish to include other relevant background information or links to a webpage (if you have one). However, remember that the information which you include in your profile is publicly viewable - only provide information you would be comfortable sharing with others, including future employers. Once you're finished, let's move on to installing Git.

2.4 Installing Git

If you are using a computer that is connected to a campus network (such as a machine in an on-campus computer lab), there is a good chance that Git is already installed.[3] However, if you plan on working through this book using your personal computer (which is probably the case) you will need to install Git on that device.

You can find the installation package on Git's homepage which is located at `https://git-scm.com/downloads`. Download the Windows release and then open the executable. Accept all default installation settings by clicking "Next" until the installer begins extracting the program files to your hard drive. Once the installation is complete, you should be able to find a folder called "Git" within the Windows start menu. Inside the folder you should see a shortcut to a program called "Git Bash." Clicking this shortcut will launch the Git command line. Let's go ahead and do that now.

Figure 2.2: The Git homepage showing the link to the installation package



Launching "Git Bash" for the first time will display an empty terminal window

[3] If you see an entry for "Git Bash" somewhere on the computer's Windows Start Menu, then Git is already installed.

which should look a bit like Figure 2.2. The first line of output in the terminal should contain your Windows account name as well the hostname of the computer you are using (we will be ignoring this information). The dollar sign ($) represents the terminal prompt which is where we will be typing all of our Git commands.

Figure 2.3: An empty Git Bash terminal

With the terminal open, let's go ahead and tell Git who we are. Type in the following command:

```
git config -- global user.name "My Name"
```

Obviously, you'll need to replace the words in quotation marks with your own name! You can use your full name, a nickname, or a handle that you regularly use online. Git will attribute any changes you make to projects to the name that you provide here. You'll also need to provide Git with your email address (make sure it's the same email address you used when signing up for GitHub) using the following command:

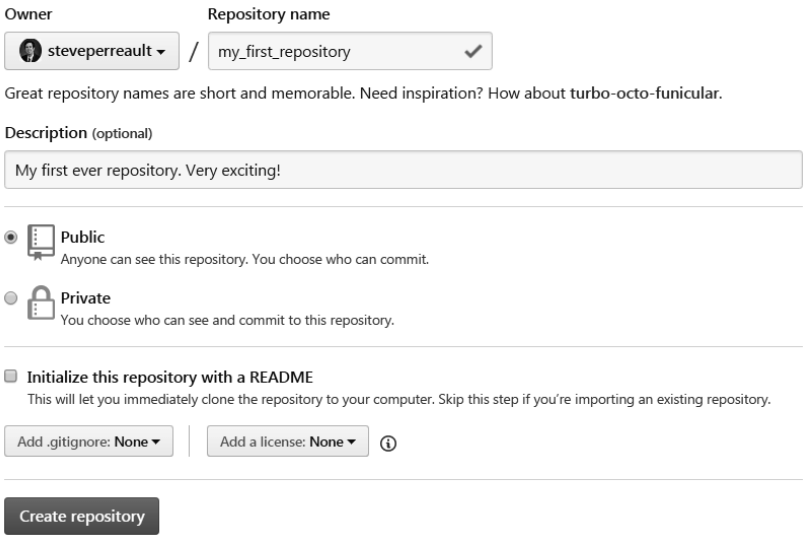
```
git config --global user.email "my_email@my_email.com"
```

2.5 Creating your first repository

Both GitHub and Git store individual projects in "repositories" (often referred to as "repos" for short). Any files that make up your projects (e.g., source code, image files, text files, etc.) can be stored inside a repository. Let's go ahead and create a repository right now.

Log into your GitHub account and click the "Create New" button (this looks like a small + sign to the left of your profile picture at the top of the webpage). Select the option to create a new repository. Name this new repository `my_first_repo` and leave the "Initialize this repository with a README" box unchecked (your screen should look similar to Figure 2.4). You can also give the repository a brief description if you like. Once finished, click the "Create repository" button.

Figure 2.4: Creating a new repository using GitHub



You've now created a repository for your project on GitHub. However, the majority of the work you do on your projects is likely going to be performed on a local computer instead of online (for example, we want to be able to work on our projects in places where we don't have an internet connection). Therefore, we are also going to create a local repository for our project using Git. We'll link the local Git and online Github repos together later in the chapter.

Before you create a new local repository, you're going to need to create a location on your hard drive (or network drive if you're using a lab computer) where that repo will live. Re-open the Git Bash terminal and enter the command:

```
mkdir ~/my_first_repository
```

This will create a directory (or folder) called `my_first_repository` where your repository and its related files will be stored on the drive.[4]

[4]This folder will be created off of the top level directory on the drive. For most Windows users, this will be `C:\users\your_user_name`. However, it may also be the top level of the network drive if you are using a PC in a campus computer lab (e.g., `G:\`). You can find the location of the top level directory by using the `~/` command.

Now navigate to the new directory you just created using the command:

```
cd my_first_repository
```

As you may have guessed, `cd` stands for "change directory." Note that if you want to navigate out of this directory and back to the top-level directory, simply use the command:

```
cd
```

Notice that the path displayed above the command prompt changes depending on the directory you are currently located in. This is demonstrated in Figure 2.5 (note that my top-level folder in this example is G:\).

Figure 2.5: Navigating directories using Git Bash

Let's navigate back to the `my_first_repository` directory. We're now going to initialize a new Git repository in this directory by using the command:[5]

```
git init
```

If successful, Git will tell us that it has initialized an empty repository in the filepath we have chosen. Congratulations - you have created your first Git repo!

[5] You can tell that this is a command which is specific to Git (as opposed to a system command such as `mkdir` or `cd`) due to the fact that is prefixed by the term `git`.

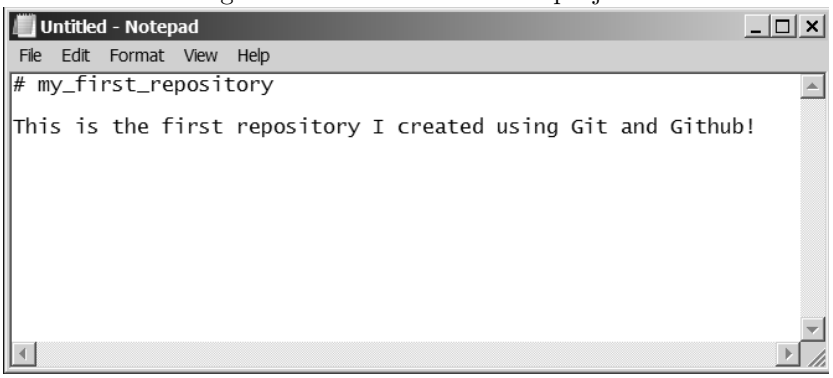
2.6 Adding files to your repository

Now that we have created a working repository, it is time to begin adding files. Every Git repository you create should contain a file that provides a brief written overview of the project. We'll be adding a file called `README.md` to our repository which contains this information.[6]

This `README.md` file must be created as a simple text file (which means that the file contains no data other than text). We can create such a file using any text editing tool. Since all copies of Windows come with the "Notepad" text editor, this is the program we will be using to create our `README.md` file. Find the Notepad program on your computer and open it (a shortcut to the program can typically be found in the Accessories folder within the Windows Start Menu).

Once Notepad has been opened, type the name of your project on the first line of the file (this is usually the same name as your repo name). Inserting the hash character (#) prior to the project name will make this text slightly larger than the remaining text in the file. (It is good practice to prefix all section headings in your `README.md` files with #). On a separate line of the file, provide a brief description of the project. Note that, in practice `README.md` should contain far more information than this; however, this level of description is sufficient for our current purposes. Figure 2.6 provides an example of what this file might look like.

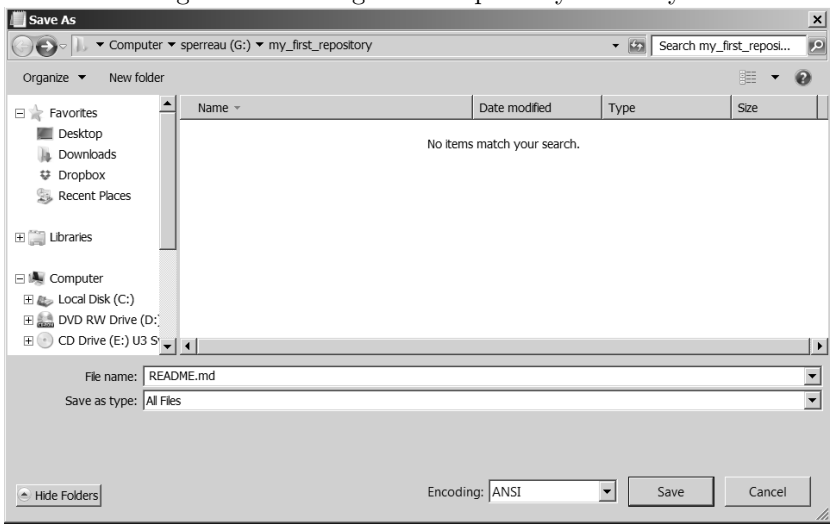
Figure 2.6: `README.md` for the project



When your text file looks sufficient, it's time to save it to the directory that we created our repository in. In Notepad, click "Save as" from the "File" dropdown menu. Then navigate to the repository directory. Type `README.md` as the "File Name" and click the "Save" button. See Figure 2.7.

[6] See `https://gist.github.com/jxson/1784669` for an excellent template for creating informative project README files.

Figure 2.7: Saving to the repository directory



We have now saved the file to the directory where we created our repository; however, we now need to specifically tell Git to add the file to our repo and begin tracking it. To do this, we'll need to re-open the Git Bash terminal.

If necessary, navigate back to the directory where the project repository is located. Then enter the following command:

```
git add README.md
```

Now that the file has been added to the repository, let's check the status of the repo using the command:

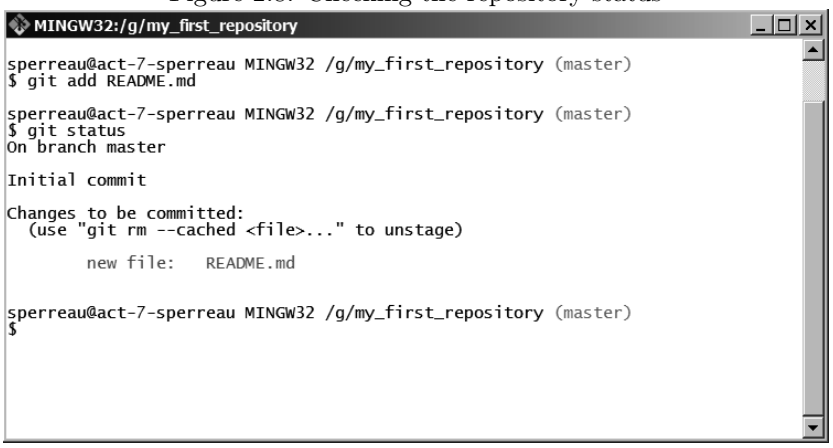
```
git status
```

The output should look something like the output in Figure 2.8.

Let's take some time to review this output. First, notice that Git has indicated that this repository is the "(master)" branch of the repository. We haven't yet created any additional branches for this project so this is to be expected. We'll talk more about incorporating multiple project branches later in the chapter.

The next line of Git output tells us that we are currently creating the "initial" (or first) commit for the repository. Whenever we save changes to a repository, we refer to this as "commiting the changes" in Git parlance. Commits also provide us with a snapshot of our project at a particular point in time, allowing us to restore the project to one of these previous states if so desired. The output

Figure 2.8: Checking the repository status



indicates this initial commit will involve adding a new file to the repository called `README.md`, as expected. Note that we should always check the status of our repository using `git status` prior to commiting any changes.

Since the status of our repo looks as expected, let's go ahead and commit the change. This can be done with the command:

```
git commit -m "Add README.md"
```

The "commit" command tells Git to commit all pending changes to the repository. We have also included the `-m` flag to associate this commit with a brief message indicating what changes are being recorded to the repository. You should always include such messages when executing a commit so you have a record of the changes made to your project over time. You can verify that your commit worked by checking the status of the repo again using `git status`. You will notice that there are no other changes scheduled to be commited.

2.7 Pushing commits to GitHub

So far we have only modified the version of the repository that is stored locally on our individual computers. However, at some point we will likely want to share our repo online with others. We can do this by linking our Git repository to our GitHub account and "pushing" the changes to the GitHub repository that we created earlier. Let's do this now.

We first need to tell Git that a remote (or online) version of our repository

exists. We do this by providing Git with the web address (url) for our GitHub repository. The format for this address is:

```
https://github.com/user_name/repo_name
```

where `user_name` and `project_name` are replaced with your GitHub username and the name of your GitHub repository.

We'll point Git to this GitHub repository using the following command:

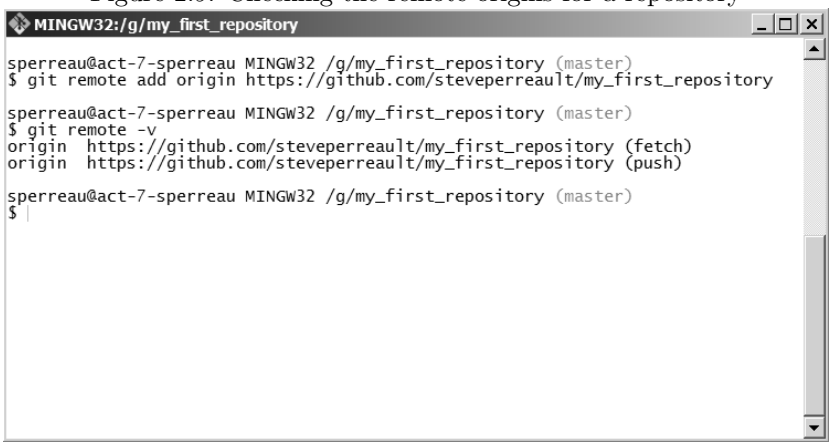
```
git remote add origin https://github.com_name/repo_name
```

The command tells Git that our project files can now be sent to a remote location (called "origin") which can be found at the github address provided. We can verify that we have configured our repository correctly be using the following command:

```
git remote -v
```

The `-v` flag tells Git to provide a verbose description of the repository which contains the url name. If your repository is set up correctly, the output will look similar to Figure 2.9.

Figure 2.9: Checking the remote origins for a repository



Note that the origin connection is listed twice with the descriptors `(fetch)` and `(push)`. This means that we are both able to *push* changes to and *fetch* changes from the online GitHub repo.

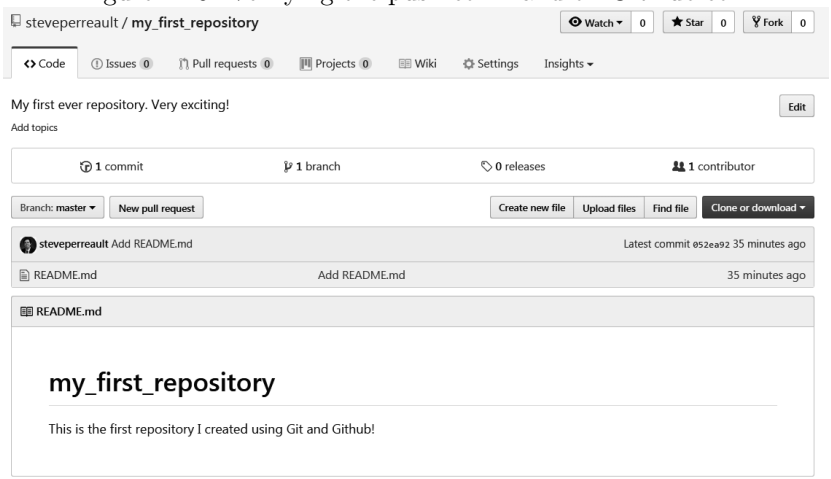
Now that we've verified that our connections are configured appropriately, we can actually push our changes up to the GitHub remote. The following command tells GitHub to push the master branch of the repo (the only branch we

have created so far) to the origin connection we just established. Note that you may be prompted to login to GitHub again when issuing this command.

```
git push origin master
```

You can verify that the push command worked successfully by visiting the GitHub repo page using your web browser. You should see that the file `README.md` has been added to the online repository and that the repo description has been appropriately updated based upon the contents of the file.

Figure 2.10: Verifying the push command on GitHub.com



2.8 Reverting a commit

Since we've only pushed one commit to the example repository we've been working with in this chapter, let's go ahead and push another one now. Open Git Bash and enter the following command from within the repository directory:

```
touch badfile.bad
```

`Touch` is a terminal command that creates an empty file. For this example, we simply want to make a change to the repository that we can commit. Adding a new file using the `touch` fulfills that objective nicely.

You can see the contents of the working directory by using the terminal command `ls`. Entering this command should display the following files which currently make up the repo directory: `README.md badfile.bad`.

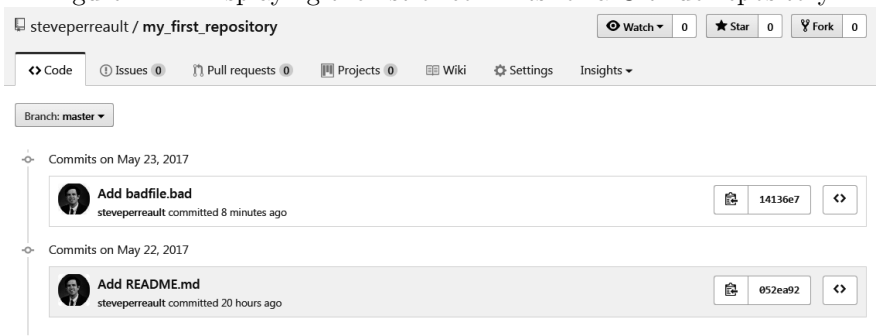
Now, using the methods discussed in the previous two sections, perform the following steps:

- Add `badfile.bad` to the local repository
- Check the local repository status, making sure that `badfile.bad` is marked as an addition to be commited
- Commit the change to the local repository with the message `"Add badfile.bad"`
- Push the new commit to the remote version of the repository on GitHub

Note that the GitHub remote origin connection that we established earlier is permanently associated with the repository unless we manually remove it (you can verify this with `git remote -v`). So you do not have to re-establish this connection every time you push commits for this repo to GitHub.

If you have completed these steps correctly, when you visit the repository page on GitHub.com you will see that `badfile.bad` has been added to the online repo. In addition, GitHub now tells us that two commits have been processed to this repository, as visibile in Figure 2.11.

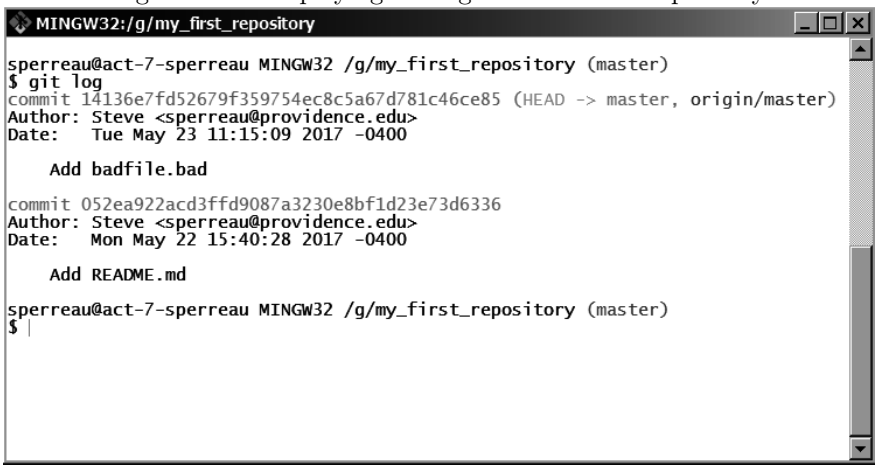
Figure 2.11: Displaying the list of commits for a GitHub repository



There may be times when we push a commit that changes a branch in some undesirable way. For example, let's assume that the last commmit pushed to GitHub was in error and that we *really* don't want `badfile.bad` to be including within the `my_first_repository` repo. In this instance, we may want to revert the repository back to what it looked like before we made the commit. We can do this using Git's `revert` command. We'll first revert the local repository to the earlier state using Git Bash and then we will push the modification of the local repo to GitHub.

To start, obtain a listing of all of the changes made to our local repository by using the command: `git log`.[7] For example, as seen in Figure 2.12, the log file for the `my_first_repository` repo currently lists two commits (Git displays the most recent commits first). In this example, we want to revert our repo back to the state that existed immediately after the first commit where `README.md` was added. In order to maintain an accurate change history, Git treats the reversion as an additional commit.

Figure 2.12: Displaying the log file for a local repository



To do this, we'll use the following Git command:

```
git revert 14136e7 --no-edit
```

where "14136e7" refers to the first seven digits of the identifier for commit we want to revert (Git does not require us to type the full identifier number).[8] The `--no-edit` flag associates a default description with the reversion commit, which is sufficient for our purposes. Once the command has been entered, displaying the log file for the repository using `git log` should now show the reversion commit as the most recent entry.

Now all that's left to do is to push this latest commit to GitHub using `git push origin master`. Once this has been done you will have successfully reverted the commit!

[7] Using the Git command `git log --oneline` will display each commit on single line. This may be useful for repositories with large numbers of commits. Also, your terminal may make your log scrollable if the output exceeds the height of its window. In this case, you can use your keyboard's arrow keys to scroll through the log. Entering `q` will return you to the terminal prompt.

[8] Note that the identifier number you type here will be unique to your specific repository.

2.9 Cloning a repository

In addition to being able to "push" a project of your own to GitHub, you can also use Git grab a copy of a repository that another GitHub user has created. This is done using Git's `clone` command.

To clone an existing repository, open Git Bash and enter the following command:

```
git clone https://github/user_name/repository_name
```

where `user_name` and `repository_name` represent the repository name and GitHub username associated with the repository to be cloned. Note that the `clone` command will do the following:

- Create a new local folder that has the same name as the repository being cloned
- Initialize the new folder as a repository using `git init`
- Copy all of the cloned repository's files and commits to the new local folder
- Create a remote connection named `origin` which points to the URL where the repository was cloned from

Pay careful attention to that last bullet point. Unless you change the default remote origin connection created by `clone`, any commits you push to GitHub will be recorded to the GitHub repository that was originally cloned (the owner of the repository will need to approve the changes).

If you want to push changes made to your locally cloned repository to *your own* GitHub account (as will usually be the case), you can use the following command:

```
git remote set-url origin https://github.com/user_name/repository_name
```

where the web address reflects the URL of the GitHub repository that you want to push changes to (this will typically be an empty repository you have created in GitHub following the steps discussed earlier in the chapter).

2.10 Advanced: Creating branches

When a repository is initially created, by default it contains a single branch called the "Master" branch. This master branch is considered the definitive

branch of the project. As such, with larger projects, changes to the master branch should be considered carefully.

However, it is possible for a repository to contain other branches which are not considered separate from the master branch. At the time of their creation, these branches contain a perfect copy of the contents of the master branch. However, because they are only a copy of the master, developers can make edits and changes to the new branch worrying about corrupting the master branch. In this way, your master branch can be insulated from inadvertent errors introduced by changes made in other branches. Ultimately, commits made in these other branches can be merged into the master branch once the changes are determined to be stable.

Let's create a new branch of the /textttmy_first_repository repository that we created earlier in the chapter. From the local directory for the repository, enter the command:

```
git branch bugfix1
```

where `bugfix1` represents the customizable name for the new branch.[9] To switch from the master branch to the new branch you just created, use the command:

```
git checkout bugfix1
```

Now push this new branch to GitHub using the command we learned earlier:

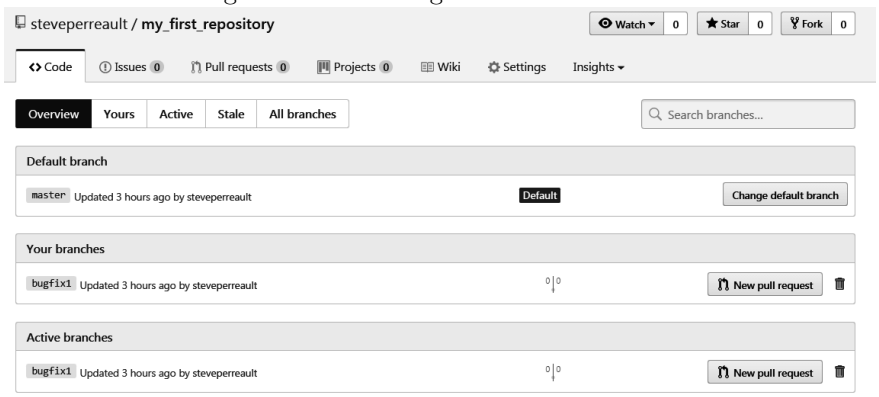
```
git push origin bugfix1
```

You can now verify that the separate branches have been pushed to GitHub, as demonstrated in Figure 2.13. Note that the commit history for the `bugfix1` branch also inherited any commits that were made to the master branch at the time that the `bugfix1` branch was created. However, going forward, any changes made to the master branch or the `bugfix1` branch will be tracked separately until the two branches are merged (as discussed in the next section).

We can now switch back and forth between the two branches using `git checkout` from within the terminal. We can switch between the branches in the remote repository hosted on GitHub as well. Note, however, that since our repository now contains two branches, we will need to be careful to ensure that we are working off of the `bugfix1` branch until we are ready to merge our changes into the master branch. In addition, we will need to remember to provide the appropriate branch name when pushing commits to GitHub.

[9] Note that software development teams often create branches to manage specific project issues, so branch names such as `bugfix1` are somewhat common.

Figure 2.13: Viewing branches in GitHub



It may be helpful to recall that we can use the `git status` command we discussed earlier to identify which branch is currently active within the terminal. Additionally, the command `git branch -r` can be used to obtain a list of all remote branches for a repository.

2.11 Advanced: Merging branches

Let's see you've finished work on the `bugfix1` branch and now want to merge the changes into the master branch. To do this, switch to the branch you wish to merge into using `git checkout` (in this case, you would switch to the master branch) and enter the following command:

```
git merge bugfix1
```

We can then push the changes to the master branch of the remote repository on GitHub using:

```
git push origin master
```

Note that if we view the commit history for the master repository on GitHub (or locally by using `git log` within the termimal), we will that the changes made to the repository as a result of the merge are listed as a separate commit. This makes it easy to revert a merge if we so wish (we would simply follow the steps for reverting a commit discussed earlier).

For purposes of working through the exercises in this book, you should never record changes (commits) to multiple branches of a project concurrently. Doing

so can cause conflicts when merging the two branches together. For example, let's say a user changes the same line of code differently within two branches that they are attempting to merge together. In this case, Git will not be able to complete the merge cleanly because it doesn't know which change is the "correct" one to be retained. Git contains a number of features that knowledgeable users can employ in order to reconcile conflicts; however, these tools are beyond the scope of this text.

2.12 Summary

This chapter discussed the reason for using a version control system when developing your software projects. It introduced the Git package for local version control and the GitHub platform for sharing project code remotely. The concept of project repositories (repos) was introduced and the process for adding files to and making changes to repositories (commits) was discussed. The chapter described how local repositories can be stored on and retrieved from the GitHub platform. Finally, basic concepts relating to branching and merging were introduced.

Note that this chapter merely scratches the surface of what you can accomplish using Git. If you are interested in learning more about how you can use Git and GitHub in your software development projects, a good starting point would be the respective documentation for the two tools. This can be found at:

- Git: `https://git-scm.com/documentation`
- GitHub: `https://guides.github.com`

2.13 Exercises

1. What is the purpose of a version control system such as Git? Mention three benefits of using such a system.
2. Why might a developer choose to also use a social media platform such as GitHub instead of just using the Git version control system locally?
3. Conduct some brief internet research to identify competing websites that offer services similar to GitHub. Do any such sites exist?
4. How can the concept of branching prevent unintended problems from occurring during the software development process?

5. Create a local repository using Git. Name the repository `my_fav_picture`. This repository should contain:

   - a README.md file which contains the name of the project and a brief description of the project.
   - a humorous image which you found online

   When finished, push the repository to your personal GitHub account.

6. The `Ch2_Ex2` repository located on this book's GitHub page contains a single file which includes the the text of my favorite poem. Clone a copy of this repository to your local drive and then:

   - Revert the most recent commit that was made to the repo
   - Modify the contents of `favorite_poem.txt` so it contains the text of *your* favorite poem.

   When finished, push the modified repo to *your own* GitHub account. (Note that you will need to change the URL that the origin remote connection points to in order to push to your own account!)

7. Create a new local branch for the repository that you created in the first exercise. Then:

   - Add an empty file called `addition.txt` to the new branch and commit the change.
   - Merge the new branch into your master branch

   When finished, push the completed project to your GitHub account.

Chapter 3

Basic Python concepts

3.1 Why Python?

We will be using the Python™ programming language to complete the data analytics tasks described in this text. The creation of Python is attributed to Guido van Rossum[1] a Dutch programmer who wanted to create a language that was easy for beginner programmers to use but powerful enough to handle large and complex projects. Over the past several decades, Python has grown to become one of the most widely used programming languages across the globe and is regularly used by accounting data scientists and is taught in many colleges and universities.

Figure 3.1: Van Rossum in 2006

Python has a number of important features which contribute to its popularity:

[1]Image attributed to Doc Searls (2006oscon_203.JPG) [CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0)], via Wikimedia Commons.

- It's free: Python is free to use and distribute meaning that cost is not a barrier to adoption.
- It's open source: Python's community-based development model has resulted in the creation of thousands of third-party libraries and modules that can handle a wide variety of computing tasks.
- It's easy to learn: Python has a simple structure and a clearly defined syntax. As such, it's a perfect language for beginner-level programmers.
- It's a "high-level" language: Python programs are abstracted from the underlying system hardware and can run on a wide variety of computers.
- It plays well with others: Python can easily be integrated with other programming languages such as C++ and Java.

3.2 Setting up a development environment

Our development environment will contain three specific tools:

1. A text editor. We will be writing all of our source code using this editor so it is important that it can recognize Python syntax highlighting. I will be using the *Notepad++* editor in this text; however there are other options available as well. Use the editor that you are most comfortable with.
2. The Anaconda distribution of the Python interpreter. The interpreter will take the source code we write and carry out the related instructions.[2]
3. A command line shell (also referred to as a terminal). We will be using this tool to interact with the Python interpreter. The shell we will be using is Windows Powershell®. You already have experiencing working with a command line shell from Chapter 2.

If you are using a computer that resides on the network of a college or university campus, there is a good chance that all three of these tools are already installed on the machine you are using. However, if you will be using a personal computer, you will may need to download and install a compatible text editor and the Anaconda Python interpreter (Windows PowerShell comes pre-installed on Windows version 7 and above).

[2]We will be using the latest version of Anaconda at the time of this book's writing (version 4.4.0), which uses version 3.6.1 of the Python interpreter. Note that the Python interpreter underwent a fairly controversial upgrade in 2008 which resulted in the version numbering switching from 2.7 to 3.X. The Python development team continues to support the 2.7 branch; however, it is currently scheduled to sunset in 2020 and we will not be using it in this book.

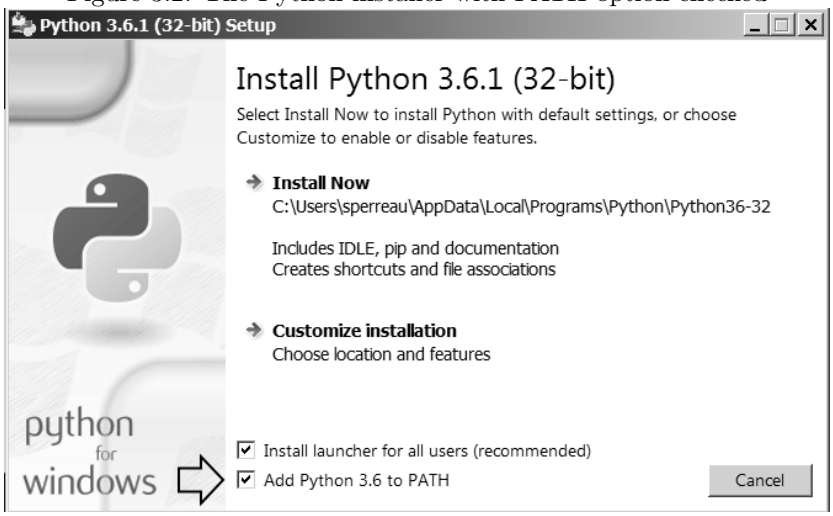
- The Notepad++ installer can be found at `https://notepad-plus-plus.org`. The installation process is straightforward and accepting all of the default installation options is sufficient.

- The latest version of the Anaconda Python interpreter can be downloaded from `https://www.continuum.io/downloads`. Before downloading, make sure that you don't already have Anaconda Python installed on your system. To do this press Win + R to open the "Run" dialog window. In the input field, type `powershell`. Then type `python` from within the PowerShell command line window. If you see a response from the Anaconda Python interpreter then Anaconda is already installed (if the interpreter does not identify itself as "Anaconda" Python, then you have only base Python installed. You will need to install Anaconda).

  The Anaconda Python installation process is relatively straightforward. Upon launching the installer, make sure the "Add Python 3.X to PATH" checkbox is selected (see Figure 3.2). Then click the "Install Now" option to proceed. [3]. The full installation will take anywhere from several minutes to an hour to complete.

  To verify that Anaconda Python has been installed, open PowerShell and type the command `python`. Python should now launch. To exit, type `quit()` and press Enter.

Figure 3.2: The Python installer with PATH option checked



[3]The installation program may look slightly different depending on the version of Anaconda you are installing

3.3 Your first program: "Hello World!"

If you have ever learned how to code before, you have undoubtedly written a "Hello World!" program. If not, you will embark upon this rite of new programmer passage now.

The first thing we need to do is set up a place where our new program will live. Launch PowerShell by pressing Win + R to open the "Run" dialog window. In the input field, type `powershell` to launch the terminal (you may want to create a shortcut to PowerShell so that you can access it easier in the future). We'll be using PowerShell to interact with our file system and to give commands to the Python interpreter.

By default, PowerShell will open at the top level directory on your hard drive. If you are using a personal machine, this will likely be `C:\users\your_user_name`. However, it may also be the top level of the network drive if you are using a PC in a campus computer lab (e.g., `G:\`). The current directory will be listed immediately prior to the `>` prompt, as shown in Figure 3.3.

Figure 3.3: The PowerShell terminal

Let's go ahead and make a directory for our first project called "hello_world"[4] using the following command:

```
md hello_world
```

[4]I usually avoid using space characters in my directory names and filenames; however, you can use them if you wish. Note that if you elect to use space characters, you will need to encapsulate the words in quotation marks when using shell commands (e.g., `cd "hello world"`).

Now change our current location to the directory we just created by typing:

```
cd hello_world
```

Notice how the path listed prior to the command prop changes to reflect the location of the current working direct.

We are now write the source code for our "Hello World" program. In this book, we're going to always write our code in the Notepad++ text editor. Launch it now. This can be done either by selecting the application from the Windows Start Menu or it can be launched directly from within the PowerShell terminal using the command:

```
start notepad++
```

By default, Notepad++ should open a blank document called "new 1" when launching for the first time. If it did not, create a new file by selecting File 〉 New from the menu bar.

We now need to tell our text editor to adjust its formatting for Python syntax. We do this by selecting the following option from the menu bar: Language 〉 P 〉 Python.

Finally, let's save our blank file. From the menu bar, select File 〉 Save as. Navigate to the `hello_world` directory you just created. Name the file `hello_world`. Lastly, make sure to save the file type as "Python file (*.py, *.pyw)."[5] When this has been done, our new program directory should now contain an empty source code file (or module) called `hello_world.py`. Note that the file extention `*.py` is reserved for Python modules.

Let's now write our first lines of code. Using Notepad++, type the contents of Figure 3.4 into the `hello_world.py` filed that you just created.

Figure 3.4: hello_world.py

```
1   print ("Hello world!")
2   print ("I did it! I wrote my first Python program!")
```

Note that you should not type the numbers preceding each line into your text editor. These numbers are included merely so we can reference specific lines of code later in the book. When this text has been entered correctly, save the file by pressing Ctrl + S.

[5] If you correctly set the editor syntax formatting to the Python language, this file type should already be selected.

We have now successfully written our first Python script. We now need to test whether the Python interpreter will run it as desired. Return to PowerShell and view the contents of the `hello_world` directory with the command `ls`. The `hello_world.py` file that we just created should be the only contents of this directory (See Figure 3.5).

Figure 3.5: Contents of the `hello_world` directory

Once you have verified that you are located in the correct directory, run the script by typing:

```
python hello_world.py
```

If you have performed the steps correctly, you should see something like Figure 3.6. Congratulations! You have written your first computer program in Python!

Figure 3.6: Contents of the `hello_world` directory

3.4 An example development workflow

So far, we've articulated a series of fairly simple steps when developing our first project. These steps are:

1. Create a project folder using `mkdir`
2. Create/modify the project's python script(s) using a text editor
3. Test the project's script(s) using the python interpreter

We can also easily integrate this process with the Git / GitHub version control workflow we learned about in Chapter 2. The steps involved in this integrated process are:

1. Create a project folder using `mkdir`
2. Initialize a new repository in the project folder using `git init`
3. Create a new repository for the project on GitHub.com
4. Link the Git repository to the remote GitHub repository using `git remote`
5. Create the project's python script(s) using a text editor
6. Add the scripts to the Git repository using `git add`
7. Make the first Git commit using `git commit`
8. Push the initial commit to GitHub using `git push`
9. Make modifications to the Python script(s) as necessary
10. Test project scripts using the python interpreter
11. Commit the changes using `git commit`
12. Push the changes to GitHub using `git push`
13. Repeat steps 9-12 until program is complete

Note that for simple projects, like "Hello World", we probably wouldn't want to go through the trouble of implementing a version control system. However, as the programs we write become more complex (such as those written when completing some of the more challenging end of chapter exercises presented later in this book), we probably will want to take the time to implement version control into our workflow. A side benefit of using Git / GitHub is that, as you work through the exercises in this text, you will build a shareable GitHub portfolio that demonstrates your skill in Python and accounting data analytics.

3.5 Mathematical operations

Python supports the basic mathematical operations you are familiar with such as addition (+), subtraction (-), division (/), and multiplication (*). Within a program, they can be used like:

Figure 3.7: Basic mathematical operations in Python

```
print(2+4)
# prints the number 6
print(2-4)
# prints the number -2
print (2*4)
# prints the number 8
print (2/4)
# prints the number 0.5
print 5 % 2
# prints the number 1
```

Note that the symbol % is reserved for the modulus operator in Python. Modulus returns the remainder left over after division. For example, the expression `5 % 2` would return a value of 1 (which is the remainder left over when 5 is divided by 2).

It's also important to know that Python uses the standard PEMDAS order of operations taught in secondary school algebra classes. That means that Python will evaluate the expression `4 + 2 * 2` as 10 while the expression `(4+2) * 2` will be evaluated as 12.

3.6 Commenting your code

Go ahead and create a Python module which contains the code listed in Figure 3.7 above and then run it (for simplicity sake, it's fine to overwrite the `hello_world` project you created earlier). I'll wait until you're finished.

All done? Good. If you did everything correctly, your output should look something like Figure 3.8 below.

If you carefully compare your program output to the original source code, you'll notice that any line which was preceded by the pound sign (`#`) was ignored by the Python interpreter. This is because the pound sign is a special character in Python which is received for identifying comments. Comments can be used to

Figure 3.8: Output for our simple arithmetic program

explain the purpose of a section of code in plain English or to temporary disable a portion of a program without removing the specific lines of code from the file. It is critically important for you to develop a habit of writing well-commented code, especially when writing programs whose source code will be shared with other developers.

3.7 Variables

You can think of a variable as a container which holds some value. To declare a variable we use the assignment operator (=), placing the name of the variable on the left and the initial value it holds on the right.

For example, let's assume we want to write a simple program to calculate the circumference of a circle using the standard formula $C = 2\pi r$. Rather than writing out the full value of pi each time we want to use it, we could store its value in a variable and then simply refer to the variable name instead of the numeric constant. See Figure 3.9 as an example:

Figure 3.9: Declaring and using variables

```
1  pi = 3.14159
2  print("The circumference of a circle with a radius of 7 is: ")
3  print(2 * pi * 7)
4  # prints the number 43.98266
5  print("The circumference of a circle with a radius of 5 is: ")
6  print(2 * pi * 5)
7  # prints the number 31.4159
```

We can reassign the value of variables later by using the same assignment operator (=). This is demonstrated in Figure 3.10 where the variable r is initially declared with a value of 7 and subsequently modified to have a value of 5:

Figure 3.10: Variable reassignment

```
1  pi = 3.14159
2  r = 7
3  print("The circumference of a circle with a radius of 7 is: ")
4  print(2 * pi * r)
5  # prints the number 43.98266
6  r = 5
7  print("The circumference of a circle with a radius of 5 is: ")
8  print(2 * pi * r)
9  # prints the number 31.4159
```

3.8 Strings

So far we have only assigned numeric values to variable; however we can assign alphanumeric characters to variables as well. In fact, a "string" simply represents a list of characters in a particular order. We identify strings in Python by placing the contents of a string within quotation marks. In fact, without knowing it, you've already been using strings in the previous examples when printing output to the screen. Let's look at an example in Figure 3.11 of how we can write a program that stores strings in variables.

Figure 3.11: Storing strings in variables

```
1  name = "Tim"
2  print(name)
3  # prints Tim
4  print("His name is " + name)
5  # prints His name is Tim
```

Notice that on line 5 we've actually concatenated two strings together: the string `"Tim"` which is stored in the variable `name` as well as the string literal `"His name is"`. We use the addition operator (+) to combined these two strings together and generate the output `His name is Tim`.

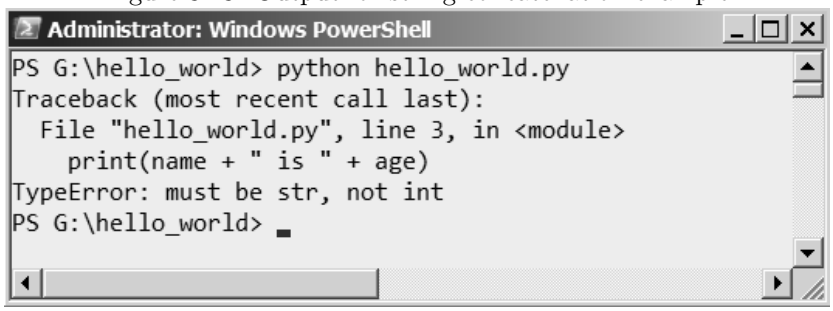
We can also concatenate strings and numeric values together but this does require one additional step. To demonstrate, what do you think the output of the program listed in Figure 3.12 would look like?

Figure 3.12: Concatenating string and non-string values

```
1  name = "Tim"
2  age = 40
3  print(name + " is " + age)
```

You probably guessed that this program would output the sentence `Tim is 40`, which is a reasonable guess. However, go ahead and try to run this program. Was the output what you expected?

Figure 3.13: Output for string concatenation example



Yikes – what happened? It looks like we've encountered our first program error! The term `traceback` indicates that this particular error is classified as an *exception*, which means that our code is syntactically correct; however Python encountered a problem when attempting to execute it. Exceptions differ from *syntax errors*, which occur when our program contains code that Python doesn't recognize (for example, if our code includes a typo).

Helpfully, Python has highlighted the line of code (3) in our file where the exception occurred and also provided us with the error type, `must be str, not int`. This particular error is informing us that Python expected the entire expression within the parentheses to have a `string` data type; however, it encountered an integer value (the variable `age`) as well. Remember from elementary math that integers are just like whole numbers but also include negative numbers (i.e., they have no decimal places!) So why did Python generate an error when it encountered the integer data type?

If you think about this carefully you'll realize that what Python has done here makes a lot of sense. Python knows that, mathematically speaking, it's impossible to add a numeric value to a string value (for example, it would be impossible to add the number 500 to the name of the month "January"). Thus, Python assumes that we have made a mistake and informs us of our error.

In order to fix our program, we need to tell Python to interpret the value of age, not as a numeric value, but as a string value.[6] That is, it should interpret the contents of age as simply representing a string of two numeric characters (4 and 0). We can do this by encapsulating the `age` variable with the function `str()`. The corrected example shown in Figure 3.14 will display the output `Tim is 40` as expected.

Figure 3.14: Converting integers to strings

```
1  name = "Tim"
2  age = 40
3  print(name + " is " + str(age))
```

As we have learned so far, Python interprets quotation marks as representing the beginning and ending of a string. You may be wondering what to do if you want a quotation mark to be included in the contents of a string itself. For example let's say we want to print the string `Tim said "hi" to me`. Typing the code:

```
print("Tim said "hi" to me")
```

would return a syntax error because Python expected the string to end when it encountered the quotation mark immediately preceding the word `hi`. We need a way to tell Python that the quotation marks surrounding the word `hi` should not be interpreted as marking the beginning and ending of a string. We can do this by inserting the backslash (\) character immediately before the these marks. Note that:

```
print("Tim said \"hi\" to me" )
```

would display the output as intended.

Other characters that can be useful when formatting strings are `\n` which inserts a new line into a string and `\t` which can be used to insert a tab into a string. For example, the command:

```
print("a\nb")
```

would display the output:

```
a
b
```

[6]Python is an example of a strongly-typed language. This means that it will not implicitly try to convert data types for us. While this requires us to think about more carefully about how we write code, it also makes it less likely that our programs will contain unintentional errors.

3.9 String indexing

We can also access individual characters within a string by using string indexing. Python assigns an index to each character in a string, with the first item having an index of 0. To access a specific character, we simply provide Python with the name of the string as well as the specific index we wish to extract (the index needs to be encapsulated in brackets). See the example in Figure 3.15.

Figure 3.15: String indexing example

```
1   foo = "abcdefg"
2   print (fop[0]) # prints a
3   print (foo[1]) # prints b
4   print (foo[6]) # prints g
5   print (foo[7]) # error, index exceeds range of the string
```

We can also start indexing from the end of the string instead of the beginning by using negative numbers. See the example in Figure 3.16.

Figure 3.16: String reverse indexing example

```
1   foo = "abcdefg"
2   print (foo[-1]) # prints g
3   print (foo[-2]) # prints f
```

We can also extract a chunk of several characters from a string using a process called *slicing*. To do so, we need to specify a starting index and an ending index separated by a colon. If we leave either the starting or ending index empty, Python will assume we mean the first and last index of the string, respectively. See the example in Figure 3.17.

Figure 3.17: String slicing example

```
1   foo = "abcdefg"
2   print (foo[0:3]) # prints abcd
3   print (foo[4:])    # prints efg
4   print (foo[:4])    # prints abcde
```

3.10 Other basic data types

Python has a standard list of data types, of which the following are common enough that you should familiarize yourself with them at this point:

- boolean: can only have a value of `true` or `false`. Useful when evaluating conditional expressions.
- int: integers; similar to whole numbers but can also include negative values
- float: floating point; represent real numbers containing a fractional part (decimal place)
- str: string, a sequence of individual Unicode characters

Python will implicitly assign variables a type at their point of declaration. For example, the declaration:

```
foo = 3.0
```

would result in Python creating a variable `foo` with value of 3.0 and a type of float.

3.11 Accepting user input

So far all of our programs have specified the values for variables within the code itself. However, most of the programs that we write will need to accept input from the user as well. We can capture user input using the `input()` function. An example of how this works is provided in Figures 3.18 and 3.19:

Figure 3.18: Accepting user input with `input()`

```
1  name = input("What is your name? ")
2  age = input("What is your age? ")
3  print("Hi, " + name + "! You are " + str(age) + " years old.")
```

This program does the following:

- Displays the string, `What is your name?`
- Accepts input from the user and stores it in the variable `name`

- Displays the string, `What is your age?`
- Accepts input from the user and stores it in the variable `age`
- Displays the a concatenated string incorporating the variables `name` and `age` (note that age is converted to a type string to prevent an execution error)

Figure 3.19: Output for accepting user input example

3.12 Comparison operators

Python recognizes a number of comparison operators that we can use to evaluate expressions:

Figure 3.20: Python comparison operators

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Comparisons constructed using these operators will return a boolean value of either `true` or `false`. See the example in Figure 3.21.

Figure 3.21: Using logical and comparison operators

```
print(5>6) # prints false
print(6<5) # prints true
print(6==5) # prints false
print(6!=5) # prints true
print("foo"="foo") # prints true
print("foo"!="bar") # prints true
```

3.13 Conditional statements

So far the programs that we have written have been pretty simple in that every line of code has been executed in a sequential order. However, we can also write programs with have branching paths that only execute based upon the result of an internal test. For example, perhaps we are writing a program that will ask the user a different set of questions dependent on whether they identify as a man or a woman. We can write such a program by using conditional statements. The simplest form is an `if` statement which evaluates a boolean expression and executes one or more commands if the evaluation returns true. This is usually paired with an `else` statement that executes if the evaluation returns false. See the example in Figure 3.22.

Figure 3.22: Using the `if` statement

```
age = input("What is your age? ")
if(age >= 18):
   print("Congratulations!")
   print("You are old enough to vote in the United States!")
else:
   print("Sorry!")
   print("You are not old enough to vote in the United States!)
```

This simple program accepts numeric input from the user and stores the response in a variable called `age`. It then prints a different response depending on whether the numeric value is greater than or equal to 18.

It is important to note that including a colon (:) after the `if` and `else` statements is required. In addition, all lines after the colon that are indented the same amount will be executed if the `if` or `else` statement is triggered.

Sometimes a program may need to consider more than two possibilities. In this case we can use the conditional statements `elif` which stands for "else if". An

example of how to use `elif` is presented in Figure 3.23.

Figure 3.23: Using the `elif` statement

```
1  favNumber = input("Can you guess my favorite number?")
2  if (favNumber==5):
3     print("Correct! 5 is my favorite number!")
4  elif (favNumber==4):
5     print("No! 4 is my least favorite number!)
6  else:
7     print("Wrong!")
```

This program contains three branches. One is executed if the user inputs the number 5, another if the user inputs the number 4, and the third if the user inputs any other number.

3.14 Logical operators

Conditional statements can be paired with logical operators to create more complex comparisons. The three logical operators that Python supports are `and`, `or`, and `not`. See Figure 3.24 for an example of how these operators can be used with a conditional statement:

Figure 3.24: Logical operators

```
1  age = input("What is your age? ")
2  if(age >= 21):
3     print("You are old enough to vote and drink alcohol.")
4  elif(age >=18) and (age < 21):
5     print("You are old enough to vote but not to drink alcohol.")
6  else:
7     print("You are not old enough to vote or drink alcohol.")
```

3.15 Iteration

While we have learned how to write programs that contain branching paths, our programs so far have been limiting to executing each instruction a single time. However, we will likely want to develop programs that have the capability to use the same code some specified number of times. In order to do this we need to

learn about iteration, which is an incredibly important computer programming concept to understand.

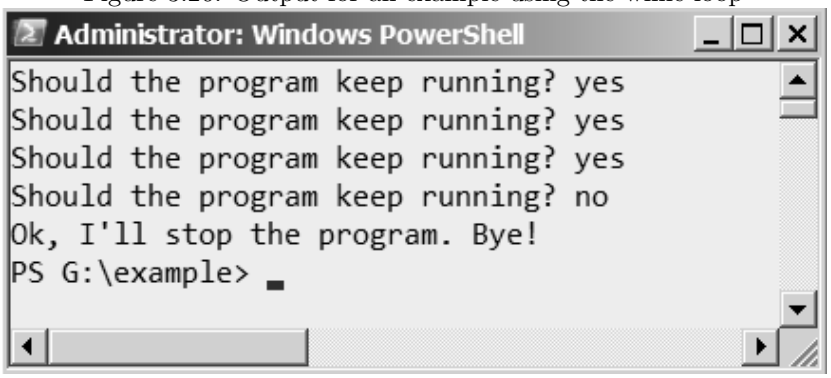
The basic idea is straightforward. We start with a test that returns a boolean result of either `true` or `false`. If the result is `true`, our program executes a block of code. Afterwards, the program evaluates the original test again. If the result is still `true`, the code block executes again. The program continues to loop through the code until the test returns a value of `false`.

The simplest type of iterative loop is called the `while` loop. An example program that uses this `while` as well as sample output are presented in Figures 3.25 and 3.26.

Figure 3.25: An example using the `while` loop.

```
1 keepRunning = "yes"
2 while (keepRunning == "yes"):
3    keepRunning = keepRunning("Should the program keep running? ")
4 print("Ok, I'll stop the program. Bye!")
```

Figure 3.26: Output for an example using the while loop



Let's work our way through this example. The program first declares a variable `keepRunning` which holds the string `yes`.[7] It then evaluates the expression to the right of the `while` command which tests whether the value of `keepRunning` is `yes`. This expression initially evaluates as `true` so the program enters the loop. Within the loop, the program prompts the user to answer the question `"Should the program keep running"?`. Note that the value of `keepRunning`

[7] In this book we'll be using the Lower CamelCase convention for naming our variables and functions. This means that when several words are joined together, the first letter of the first word is lowercase but the first letters for subsequent words are uppercase. Naming conventions in programming are primarily an aesthetic choice; however they do sometimes generate heated debate.

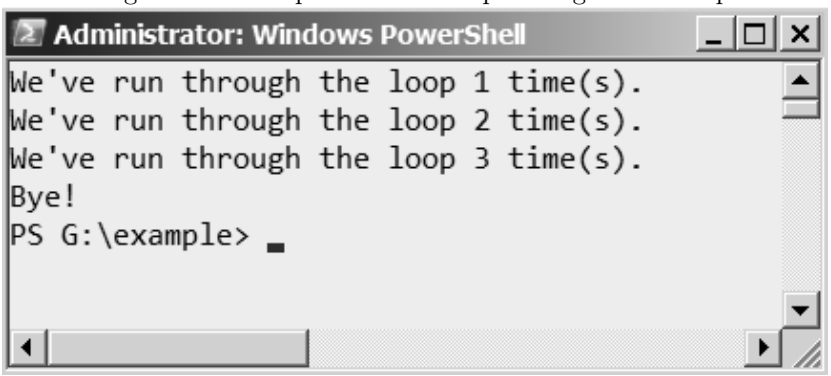
is then overwritten with the input the user provides. Since there are no more lines of code within the block, the program re-evaluates the expression to the right of the `while` command. If the expression still evaluates as `true` (i.e., `keepRunning` still holds the string `yes`) then the program enters the loop as well and the process repeates. If the expression evaluates as `false`, the program skips over the loop and prints `"Ok, I'll stop the program. Bye!"`. Note that, in this example, the code within the loop could theoretically execute an infinite number of times so long as the user keep entering in the string `yes` when prompted.

There also may be times where we want a program to execute over a loop some predetermined number of times. In this case, we would want to use a different type of iterative loop called a `for` loop. An example of a program using this type of loop (as well as the respective output) is provided in Figures 3.27 and 3.28.

Figure 3.27: An example using the `for` loop.

```
1  for i in range(0,3):
2     print("We've run through the loop " + str(i++) + " time(s).")
3  print("Bye!")
```

Figure 3.28: Output for an example using the for loop



The first line of code declares a new variable, `i`. The command `in range(0,3)` then tells Python to iterate over a sequence that contains 3 items beginning with 0 (i.e., the sequence 0, 1, 2). To begin, `i` is initially assigned the value of the first item in the sequence (`0`). The program then enters the loop. Within the loop, the message `We've run through the loop X time(s)!` is displayed, with X representing the value of `i+1`. Note that if we instructed Python to simply display the value `i` instead of `i+1`, the user would see the sequence 0, 1, 2 instead of 1, 2, 3. As discussed earlier in the chapter, we also need to tell Python to convert `i` to a string using `str()`. The loop then executes again,

this time with `i` taking the value of the next item in the sequence (1). When finished, the program then iterates through the sequence one final time (i = 2) before breaking out of the loop and displaying the `Bye!` message.

3.16 Searching for help

As a novice programmer, you will frequently encounter situations where you need a bit of help. Your first stop for help should usually be the latest version of the Python documentation which can be found at `https://docs.python.org/3`. As the "authoritative guidance" prepared by the Python Foundation, you can be assured that this documentation is up-to-date and accurate.

Search engines such as Google™, can also be helpful tools for finding answers to your programming problems. However, you should be aware that many online programming websites have weak or non-existent methods for quality control. Thus, you should attempt to verify that any source code you plan to adopt into your own programs comes from a reputable source.[8]

3.17 Summary

This chapter began by presenting a brief background of the Python programming language and then discussed how to set up a simple development environment. Common data types were presented as well as functions for displaying output and accepting user input. Constructing conditional statements using comparison and logical operators was also introduced. The chapter concluded with a discussion of iteration using both `for` and `while` loops.

3.18 Exercises

1. Indicate the data type for each of the following expressions (your choices are bool, int, str, or float):
   (a) 3
   (b) 3.0
   (c) s
   (d) -3

[8] Certain websites, such as Stack Overflow™, have adopted a community rating feature which allegedly prioritizes high quality responses to user questions.

(e) True

(f) bar

2. Indicate the output that would be returned from each of the following expressions:

   (a) "pythonrocks"[1]

   (b) "pythonrocks"[2]

   (c) "pythonrocks"[:2]

   (d) "pythonrocks"[2:]

3. Indicate the output that would be returned from each of the following expressions:

   (a) 6 >= 4

   (b) 3 != 3

   (c) 2 >1

4. Indicate whether each of the following expressions would evaluate as `true` or `false` assuming the variable `foo` is previously declared as `foo = 12`:

   (a) if (foo == 13) or (foo == 12)

   (b) if (foo == "12")

   (c) if (foo not 12)

   (d) if (foo == 12) and (foo == 13)

   (e) if (foo not 13)

5. How would you write the following code more efficiently using a `for` loop?

```
1  print(2)
2  print(4)
3  print(6)
4  print(8)
5  print("Bye!")
```

6. Write a program that takes the age of the user and converts it to dog years (you can assume that one dog year is equivalent to one human year). For example, if you are 98 years old in human years that means you are 14 years old in dog years.

7. Write a program that takes a user's weight (in kilograms) and height (in centimeters) and calculates their Body Mass Index (BMI). Based upon their BMI, the program should then indicate whether the user is considered underweight (BMI of 18 or less), normal weight (BMI of greater than 18 but less than 26) or overweight (BMI of 26 or greater). Note that the formula for calculating BMI is weight/height^2.

As a check, the BMI for a user with a height of 180cm and 70kg would be 21.6.

8. Write a program that takes the users name and displays it back in reverse order. For example, if the user indicates that her name is `Sarah`, the program should print the name `haraS`.

9. Modify the program in Figure 3.27 so it prints the messsage `"We've run through..."` a specific number of times requested by the user. Note that the `input()` function you have learned about assigns using the string data type and that strings can be converted to integers using the function `int()`.

10. Write a program that takes a user's name and iterates over each letter in the name using a `for` loop. For each round of iteration, your program should print the message `Letter X of your name is "A"`, where X represents the position of the letter and A represents the letter itself.

    For example, if the user entered the name `Tom`, the program would output:

    ```
    Letter 1 of your name is T
    Letter 2 of your name is o
    Letter 3 of your name is m
    ```

11. Write a program that translates the user's name into Pig Latin. English words are translated into pig latin by taking the first letter of the word, moving it to the end of the word, and adding "ay". For example, the name "Steve" in Pig Latin would become "Tevesay."

12. **Portfolio project:** The *Fibonacci sequence* is an integer sequence identified by 13th century mathematician Leonardo of Pisa that is characterized by the fact that every number after the first two is the sum of the two proceeding ones. The first ten items in the sequence are: `1, 1, 2, 3, 5, 8, 13, 21, 34, 55`.

    Write a Python program that displays the first 25 numbers in the *Fibonacci sequence*. Each line of output should display both the number itself and its position in the sequence. For example:

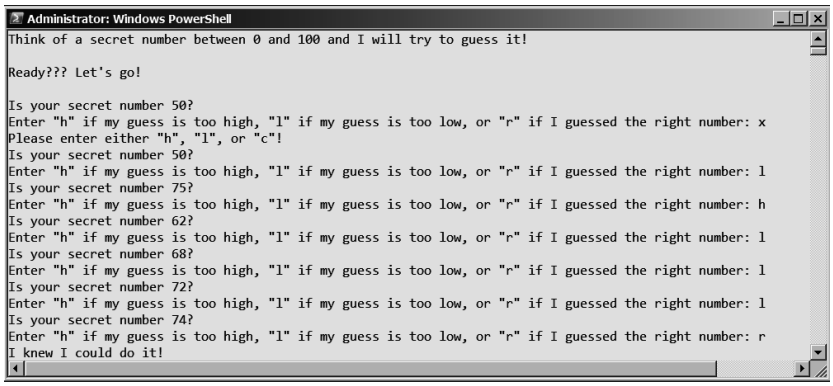
    ```
    1:  1
    2:  1
    3:  2
    4:  3...
    ```

    As a check, the 15th number in the Fibonacci sequence is 610 and the 25th number is 75,025.

13. **Portfolio project:** Write a program that guesses a secret number. The program should contain the following features:

- It should ask the user to think of a number between 0 and 100.
- It should guess what the number is and prompt the user to indicate whether the guess was too high, too low, or correct.
- It should repeat the preceding step until the user indicates that the number is correct.
- It should display an error message if the user enters a command that the program does not recognize.

As an example, your program's output might look something like:



Hint: Remember that you can round floats to the nearest integer by using the function `int()`.

Chapter 4

Intermediate Python concepts

4.1 Introducing functions

When writing Python programs, we will often to write code that can be reused in a variety of different circumstances within the program. We can do so by defining functions. An example of a Python function and the related program output is provided in Figures 4.1 and 4.2.

Figure 4.1: A function example

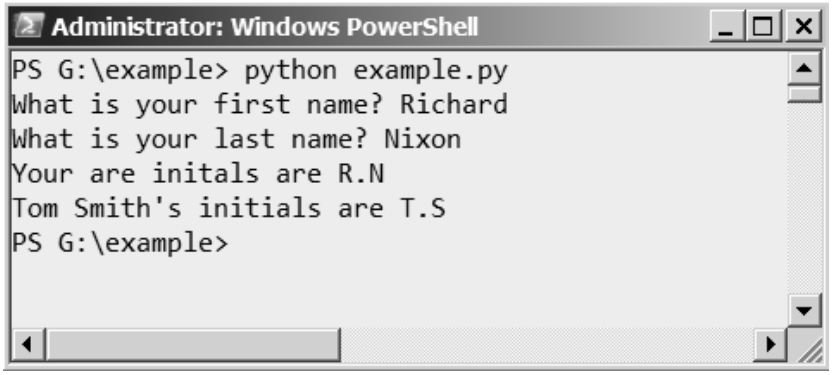
```
1  def PrintInitials(firstName, lastName):
2      print(firstName[0] + "." + lastName[0])
3
4  firstName = input("What is your first name? ")
5  lastName = input("What is your last name? ")
6  print("Your initals:", end=" "), PrintInitials(firstName, lastName)
7  print("Tom Smith's initials:", end=" "), PrintInitials("Tom", "Smith")
```

Let's break down the syntax for this function.

- First, we tell Python to define a new function called `PrintInitials` using the command `def`.

- We thenn tell the function to expect two arguments each time it is called, `firstName` and `lastName`.
- We end the function definition with a colon `I`.
- We write the specific code that the function should execute (each line of code must be indented to be attached to the function).

Figure 4.2: Output for a function example



Now whenever we want to call this specific function, we simply call it by name and encapsulate the specific argument values we want to pass in parentheses separated by a comma. The program presented in Figure 4.1 provides an example of passing variable as arguments, `PrintInitials(firstName, lastName)`, as well as specific strings, `PrintInitials("Tom","Smith)`.[1] Be aware that all arguments in Python are passed by reference. This means that if you change the value of a variable passed as an argument within a function, the value also changes for any other part of your program that has access to that variable.

When writing functions, we can also use the `return` expression to pass an argument back when the function exits. For example, the function presented in Figure 4.3 calculates the user's initials in a new variable called `initials` which is returned to the calling function `print()`.

4.2 Global versus local variables

It is important to understand that variables which are defined inside a function body have what is called *local scope*. This means that these variables can only

[1]This program also contains a simple "hack" to make sure that the initials are printed on the same line as the preceding text. While the `print("")` function normally appends a newline character at the end of its output, the keyword argument `end = " "` tells the function to append a space instead.

Figure 4.3: A example using `return`

```
1   def ReturnInitials(firstName, lastName):
2      initials = firstName[0] + "." + lastName[0] + "."
3      return initials
4
5   firstName = input("What is your first name? ")
6   lastName = input("What is your last name? ")
7   print("Your initals are", end=" ")
8   print(ReturnInitials(firstName, lastName))
```

be accessed by code contained within the function itself. Conversely, variables which have *global scope* can be accessed by the entirety of the program. Figure 4.4 provides an example of declaring variables with local versus global scope.

Figure 4.4: Global versus local scope

```
1   myGlobalVar = "Hello!" # a global variable
2   def myFunction():
3      myLocalVar = "Hello!" # a local variable
4      return myVar
5
6   print myGlobalVar # prints "Hello!"
7   print myLocalVar # exception - variable is out of scope
```

4.3 Lists

All of the programming examples we have reviewed so far have involved simple types of data (primarily numbers and strings). However, we will often encounter programming problems where we need to structure different types of data together in complex ways using compound data types. The first compound data type we will review is called the `list`. Lists are simply containers of data elements that are organized from first to last. We can declare lists using the following syntax:

```
numbers = [1, 2, 3, 4, 5]
animals = ["cow", "mouse", "horse", "pig"]
prices = [1.99, 2.99, 3.00, "free", 9.99]
```

As you can see, the contents of lists are placed within brackets and separated by commas. Lists can store any of the types of data that we have learned about

so far (such as `string`, `int`, and `float`) and can even store combinations of different data types. We can then access individual elements with a list by referring to the specific index we want to access, similar to the string indexing method we learned about in Chapter 2.

`numbers[0]` returns the integer `1`
`animals[1]` returns the string `mouse`
`prices[2:4]` returns a list containing the float `2.99` and the string `free`.

We can also use the `for loops` that we learned about in Chapter 2 to iterate over the elements in a list. For example, the code presented in Figure 4.5 prints the elements of two lists using iteration.

Figure 4.5: Lists and iteration

```
1   animals = ["cow", "mouse", "horse", "pig"]
2   numbers = [1, 2, 3, 4, 5]
3
4   for i in animals:
5      print(i);
6
7   for i in numbers:
8      print(str(i))
```

We can also create lists that have more than one dimension. For example, let's say we wanted to create a list of X and Y coordinates. We could initialize such a list as follows:

```
coords = [[1, 2], [1, 7], [2, 3]]
```

If we then wanted to access the first set of coordinates (1, 2) we could type:

```
coords[0]
```

By nesting our bracketed terms, we can access specific elements within a multidimensional list. For example, if we wanted to access a specific coordinate within a set (for example, the Y coordinate 7 from the second set of coordinates) we could type:

```
coords[1][1]
```

It's also important to understand that lists are mutable. This means that we can assign a new value to a list after it has been created. For example, if we wanted to change the value of the first element in a list, we could do so using the assignment operator as follows:

```
myList[0] = "foo"
```

Python also supports a large number of manipulation methods for lists that are worth reviewing. A brief description of some of these methods is provided below.

- len(): Returns the number of elements in a list.
  `len(myList)`
- append(): Adds a new element after the last element in a list.
  `myList.append("qux")`
- pop(): Removes the last element from a list. If an index is provided within parentheses, this method will remove that specific element from a list.
  `myList.pop()`
  `myList.pop(1)`
- insert(): Inserts a new element into a list at the index provided
  `myList.insert(1, "foo")`
- remove(): Removes the first element in the list which contains the value specified
  `myList.remove("foo")`)
- index(): Find the index for the first occurance of an element in a list.
  `myList.index("foo")`
- sort(): Sort in ascending numeric or alphabetical order (list elements must be either entirely numeric or entirely string).
  `myList.sort()`

4.4 Dictionaries

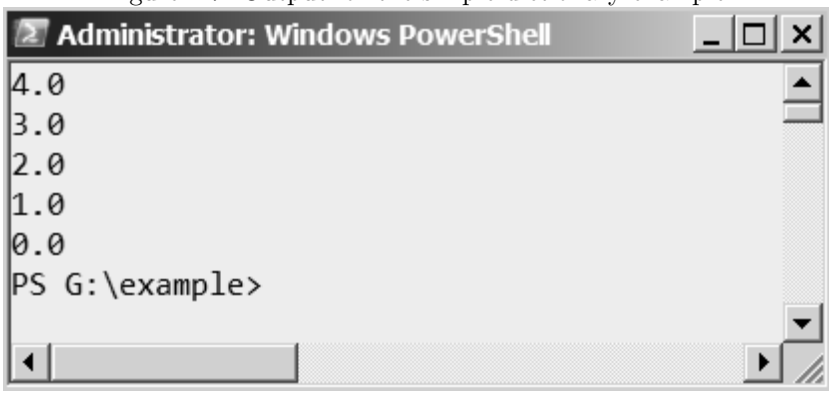
Dictionaries are similar to lists in that they can contain elements of various data types. However, while the indices of lists are required to be integers (e.g., 1, 2, 3), dictionaries can have indices (specifically referred to as "keys") of any type. A dictionary is really just a data structure that contains a collection of these key-value pairs.

Dictionaries are initialized using curly braces, values are preceded by colons, and key-value pairs are separated by commas. After initialization, specific values in the dictionary can be referenced by using their respective key. The code example and output provided below demonstrate the concept (see Figures 4.5 and 4.6)

Figure 4.6: A simple dictionary

```
1  gradePoints = {"A": 4.0, "B": 3.0, "C": 2.0, "D": 1.0, "F": 0.0}
2  print(gradePoints["A"])
3  print(gradePoints["B"])
4  print(gradePoints["C"])
5  print(gradePoints["D"])
6  print(gradePoints["F"])
```

Figure 4.7: Output for the simple dictionary example



This program creates a dictionary which assigns grade point values to keys representing university letter grades. Each grade point value is then accessed by reference to its respective letter.

It is important to note that we should never try to access a key that doesn't exist, For example, if we modified the code above to include the line `print(gradePoints["E"])`, Python would generate an error and halt the program. This can be prevented by using a conditional expression to confirm that a key exists prior to attempting to access its value:

```
if "E" in gradePoints:  print (gradePoints["E"])
else:  print ("Grade not found.")
```

We can loop through dictionaries to obtain keys and corresponding values by using the `items.()` method. Figure 4.8 demonstrates this concept.

This program prints of the key-value pairs in the `employees` dictionary. During iteration, the name each key is stored in the variable `k` while the related value is stored in the variable `v`. Note also that, since `v` contains integer values in this example, it needs to be passed as a string using `str()` to the `print()` function.

Figure 4.8: Iterating over a dictionary

```
employees = {"Name": "Steve", "Job": "Accounting professor", "Age":
    90}

for k, v in employees.items():
print (k + ": " + str(v))
```

Dictionary values can also be easily changed by referencing their related key:
`employees["Name"] = "Bob"`

New key-value pairs can by using the assignment operator:
`employees["Weight"] = 150`

Finally, key-value pairs can be deleted from a dictionary by using the `del` statement:

```
del employees["Age"]
```

Be warned that it can be dangerous to make significant changes to dictionaries since the content will change for other functions that reference the same dictionary. Before modifying a dictionary's contents, consider whether it would be more useful to create a modified copy of the dictionary. The `copy()` method can be used to create a dictionary copy as follows:

```
employees2 = employees.copy()
```

4.5 File input and output

While we have already learned how to create programs which can collect data input from a user, the programs we will write to handle accounting data will often need to be able to read/write data from/to a file. Python 3+ incorporates a number of handy i/o functions and methods that can make working reading and writing files fairly simple.

To begin, let's create a new file:

```
myFile = open("myFile.txt", "w", newline="")
```

This function creates a new file object called `myFile` which points to the path of the file to be opened (\myFile.txt). The second paramater (`"w"`) tells Python

to open the file for writing. If we omit the second parameter, Python assumes that are simply opening an existing file for reading only. In this case, if the file does not exist, Python will generate an error. The final parameter `newline=""` is required for properly formatting a CSV file on Windows-based systems.

Once we have created a new file object, we can modify the file by using the `write()` method as follows:

```
myFile.write("Mary had a little lamb")
```

This line of code will append the string `Mary had a little lamb` to the file referenced by the myFile object. This method will also return the number of characters contained in the string (including spaces) which can be useful when checking for errors.

Python contains many methods that can be used to read data from a file. We can read the entire contents from a file using the `read()` method as follows:

```
newFile.read()
```

Or we can read a single line from a file using `readline()`:

```
newFile.readline()
```

We can also read lines from a file one at a time by iterating over the file object as follows:

```
for line in newFile:  print(line, end="")
```

However, reading entire lines of data is often not terribly useful. You may normally find yourself wanting to read individuals data elements which are separated by some specific delimiting character, such as a space or a comma. For example, let's assume that we want to write a program that reads a file `names.txt` which contains the following data:

```
sam bob julie steve rachel
```

We want our program to read each of the names in the file individually and store them in a list called `nameList`. The code snippet presented in Figure 4.9 demonstrates how this program might be written.

This program begins by opening the `names.txt` file. Note that we have omitted the second parameter from the `open()` function because we want Python to open the file for reading only. Next, we create an empty list `names` which will store the data elements that we read from the `names.txt` file.

Figure 4.9: Reading a space delimited file

```
1   file = open("names.txt")
2   names = []
3
4   for line in file:
5      for word in line.split(" "):
6         names.append(word)
7
8   print("The contents of the names list are:")
9   print(names)
10  file.close()
```

We then use a pair of nested `for` loops to read each name individually. The first loop simply iterates over each line in the file. However, the second loop iterates over each *word* in the line by splitting the line into individual segments delimited by the space character (" "). Each word read is then added to the list using the `append()` method that we discussed earlier. Finally, the program print the new contents of the `names` list data structure. If we wanted, we could then access individual items of the list by referencing its specific index (e.g., `print(names[0])`.

When we are done writing to the file, we need to call `.close()` to close the file and free up the system resources taken up by the file object. Note that any references your program makes to a file object after it has been closed will fail.

4.6 Reading and writing data using JSON

While the methods that we have learned about so far are useful for reading and writing simple data, they have some drawbacks. First, the `read()` and `write()` methods only work with string data types. Therefore, numeric values would need to be converted to a different data type, using a function like `int()`, if they need to be manipulated. In addition, these simple methods are really not very convenient when trying to read/write more complex data structures like lists or dictionaries.

Thankfully, Python supports the ability to save more complicated data types using the JSON (Javascript Object Notation) data interchange format. JSON is commonly used by many different types of applications so adding support for it into your programs can be quite useful (you can read more about the JSON format at `https://wwww.json.org`).

To encode an object using JSON, we pass the data to be encoded as well as an open file object to the `dumps()` function. Data encoded in the JSON format can be decoded by using the `load()` function. An example of a simple program which demonstrates the use of these functions is presented in Figure 4.10.

Figure 4.10: Reading and writing using JSON

```
1   import json
2
3   names = ["sam", "bob", "julie", "steve", "rachel"]
4   newNames = []
5
6   file = open("names.json", "a")
7
8   json.dump(names, file)
9   file.close()
10
11  file = open("names.json")
12  newNames = json.load(file)
13  print("The contents of newNames is:")
14  print(newNames)
15  file.close()
```

A brief review of this code is in order. The first line of the program imports the `json` module which allows us to use Python's JSON-specific encoding/decoding functionality.[2]. The program then creates a list called `names` which contains five elements. It also initializes an empty list called `newnames`. A new file object (called `file`) is then initialized and mapped to the path `\names.json` (the .json file extension is typically used to denote JSON-encoded data files). Note that the program passes the `"a"` parameter since `names.json` is a new file that will be written to.

The program then calls the `dump` function which is part of the `json` module, as discussed earlier. The first argument passed to the function is the name of the list we wish to write, while the second argument represents the name of the file object to be written to. On line 9, the program closes the `file` object and removes it from memory. At this point, if we viewed the working directory of our program, we would see that, in addition to our Python script, it now contains a file called `names.json`.

The program then creates a brand new file object and associates it with the `\names.json` file path. Note that the `"a"` argument has been omitted from the call to the open function because we want Python to open an existing file for

[2]You can think of Python modules as simply collections of source code which can provide access to additional Python functionality. We'll be using many different Python modules as we advance through this text

reading only (this is the default function call). The program then uses the `load` function from the `json` module to load the contents of the `names.json` file to the empty `newNames` list. The program then prints the contents of `newNames` to ensure that the JSON data was successfully decoded. Finally, the program closes the file object and removes it from memory.

4.7 Advanced: Classes

So far we have learned about various types of data as well as how to create various functions to manipulate that date. You can perhaps imagine that it might be helpful if there were a way to somehow bundle together certain types of data, as well as their related functions into a single unitary package.

Python is an example of what is called an *object oriented programming language*. As such, Python allows us to create unique data structures (called classes) which contain both data and specific functions that interact with that data (called interfaces). These classes are essentially blueprints for creating specific data types (called objects) that contain the data types and functions specified in the classes. This probably all sounds very confusing but I promise that it will become more clear as we talk through the following example.

Imagine we have been hired to develop a simple program for tracking a client's accounts receivable. The specifications that the client has for this program are relatively simple:

- Each invoice added to the program should be assigned a unique invoice number
- For each invoice, the program should keep track of the customer name, the amount due, and whether the invoice has been paid
- The program must provide the ability for users to modify the amount due on the invoice
- The program must provide the ability for users to display the customer name, amount due, and payment status on demand

As we begin, we need to think carefully about how this program is going to store the invoice data that it is responsible for keeping track of. For example, we could store the invoice number as an integer varaiable called `invoiceNo`, the customer's name in a string called `customer`, the amount owed in a float called `amount`, and the payment status (paid or upaid) in a boolean variable called `paid`. We would then also need to think about creating functions that

would allow the user to modify the amount due, set the payment status, and to display the attributes of each invoice. Perhaps we could call these functions `modifyAmount()`, `setPaid()`, and `display()`, respectively.

Obviously, each invoice will need its own set of variables and functions. So how might we keep track of which variables and functions are assigned to each invoice? Well, we could adopt a naming convention which would allow the specific invoice to be inferred from a variable/function name (e.g., `inv1_customer`. `inv2_customer`, `inv3_setPaid()`, etc.) This makes conceptual sense but keeping track of all of these variables and functions would become unmanageable as the number of invoices grows in size. In addition, this approach would generate an enormous amount of repetitive code.

Since all invoices have essentially the same set of attributes, it would be much easier to instead develop a blueprint (called a "class") which describes the specific variables and functions that are needed by each invoice. Then every time a new invoice needs to be created, we can use this blueprint to create an instance of the class (called an "object") which represents the specific invoice. Let's review Figure 4.11 for an example of how this might work.

To create a class, we use the `class` statement followed immediately by the name of the class that we wish to create (in this case, we are creating a new class called `Invoice`. All of the code associated with the class definition is then indented after this initial statement.

The first line of the class definition (line 2 in Figure 4.11) is reserved for the class's "documentation string." This is simply a variable with a string type that typically contains a brief description of the class. The exact contents of the documentation string are entirely up to you.[3]

Line 3 of the program initializes a new variable `invoiceNoCounter`. This variable is referred to as a "class variable" because it will be shared by all instances of the class. The purpose of this variable is track the number of invoices that have been recorded in the system for purposes of assigning each invoice a unique invoice number. This variable is initially assigned a value of 0.

On line 5 we define a class function called `__init__`. Functions that are defined within class definitions are referred to as "methods" and we will be using this terminology from this point forward. The double underscores before and after the method name indicates that this is a special name reserved by the Python interpreter. The `__init__` method contains code that will be executed each time time a new instance of a class is created. Every class definition you create should contain a `__init__` method.

[3]The document string is stored in the class variable `__.doc__`

Figure 4.11: A simple program for tracking accounts receivable

```
1   class Invoice:
2      "Common base class for invoices"
3      invoiceNoCounter = 0
4
5      def __init__(self, customer, amount):
6         self.customer = customer
7         self.amount = amount
8         Invoice.invoiceNoCounter += 1
9         self.invoiceNo = Invoice.invoiceNoCounter
10        self.paid = False
11
12     def changeAmount(self, amount):
13        self.amount = amount
14
15     def setPaid(self):
16        self.paid = not self.paid
17
18     def display(self):
19        print("Customer: " + self.customer, end =", ")
20        print("Invoice #: " + str(self.invoiceNo), end = ", ")
21        print("Amount: " + str(self.amount), end = ", ")
22        print("Paid: " + str(self.paid))
23
24  invoice1 = Invoice("Acme Company", 1541.99)
25  invoice2 = Invoice("XYZ Inc.", 4750.15)
26  invoice1.display()
27  invoice2.display()
28  invoice1.setPaid()
29  invoice2.changeAmount(4500.99)
30  invoice1.display()
31  invoice2.display()
```

Note that the `__init__` method within our Invoice class has three arguments. The first argument, `self`, is always included as the first argument for any class method that you create. `self` is simply an identifier used to refer to a specific instance of a class. Be advised that you do not have to specifically pass the self argument when calling a function, Python will implicitly pass it for you. The second paramter, `customer`, will take the name of the customer associated with a specific invoice being created. The final parameter, `amount`, will take the dollar amount of the invoice being created. The body of the `__init__` method takes the later 2 parameter values and assigns them to two new instance variables called `customer` and `amount`, respectively (variables defined within a class that are shared with class instances are refereed to as "members.") The prefex `self` indicates that the new members should be initialized uniquely for each specific

instance of the class. This ensures that each class instance will have its own unique `customer` and amount members.

The third line of the `__init__` method (Line 8) increments the `invoiceNoCounter` class variable which we declared earlier by 1. Recall that class variables are shared by all instances of the class, which we reference this variable using the class name `Invoice`. The fourth line of the method assigns this value to a new member `invoiceNo`. The `self` prefix indicates that this variable will be created uniquely for each instance of the class. The final line of the method creates a new instance member called `paid` which will indicate whether the invoice has been paid. This member is assigned an initial value of `False`.

Line 12 defines a new class method `changeAmount` which is used to change the value of an invoice object. The method takes the value passed as `amount` and assigns it to a new instance member of the same name.

Line 15 defines a new class method `setPaid` which toggles the value of the `self.paid` data member. Note that the expression `not self.paid` returns the inverse of the current value of `self.paid` (i.e., if the current value of `self.paid` is `true`, the expression returns `false`.) Finally, the class method `display` defined on line 18 prints the value of the instance's members.

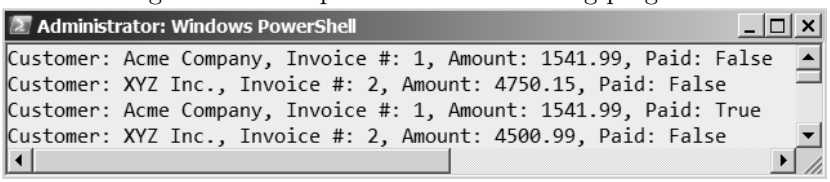
With the class definition complete, we can new create individual instances of the class. Let's review the expression included in line 24 of the program and repeated below:

```
invoice1 = Invoice("Acme Company", 1541.99)
```

This statement object called `invoice1` which has a class type of `Invoice`. Remember that the `__init__` method is called when an instance of a class is created, that this method takes three arguments, and that Python automatically passes the first argument `self` for us. This means that Python expects us to pass a value for the remaining `customer` and `amount` arguments when initializing the object. In the code above, the string `Acme Company` is passed as the `company` argument and `1541.99` is passed as the `amount` argument.]

The remaining lines of the program call various other object methods. For example, the call to `invoice1.setPaid()` on Line 28 toggles the `paid` data member within the `invoice1` object. The call to `invoice2.setamount` on Line 29 passes the value `4500.99` as the argument `amount` which is used to update the value of the `invoice2.amount` data member. Finally, the calls to the `display` method display the value of the data members for each object. The output that would be displayed upon running the full program is presented in Figure 4.12.

Figure 4.12: Output for the AR tracking program



4.8 Advanced: Inheritance

In the previous section, we created a simple Python class definition for creating invoice objects. This class contained data members for an object's invoice number, customer name, amount, and payment status as well as specific methods for accessing and manipulating those members.

Suppose that our client is happy with our work so far and has now asked us to modify our Python program to also track accounts *payable* invoices. Our revised program should share many of the features of our previous program to track accounts receivable invoices: it should store the payables invoice number, whether it has been paid, and should provide the ability to modify payable amounts, update the paid status, and display the invoice attributes.

Figure 4.13: The parent invoice class

```
1   class Invoice:
2      "Common base class for invoices"
3      invoiceNoCounter = 0
4
5      def changeAmount(self, amount):
6         self.amount = amount
7
8      def setPaid(self):
9         self.paid = not self.paid
10
11     def display(self):
12        print("Customer/Vendor: " + self.company, end =", ")
13        print("Invoice #: " + str(self.invoiceNo), end = ", ")
14        print("Amount: " + str(self.amount), end = ", ")
15        print("Paid: " + str(self.paid))
```

We could start from scratch and create a new class definition for accounts payable invoices. However, we could instead modify our original invoice class to contain only data members and methods that are common across all invoice objects. We could then create two derived classes from this parent invoice class

that relate to accounts payable and accounts receivable invoices specifically. These derived classes would inherent the common attributes from the parent class. The example in Figure 4.13 demonstrates how the parent class could be defined, while the derived classes are presented in Figure 4.14.

Note that the parent `Invoice` class is very similar to the version of the class presented earlier. It still contains a class variable for ensuring that accounts receivable invoices contain a unique ID number. It also has methods to change the amount of the invoice, set the payment status, and display the invoice attributes. Note that we modified the `display` method slightly due to the fact that the company listed on a receivables invoice would be referred to as the "customer," while the company listed on a payables invoice would be referred to as the "vendor."

Figure 4.14: The derived classes

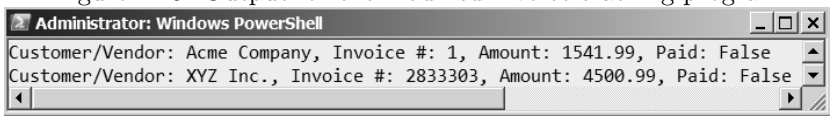
```
1   class AccountsReceivable(Invoice):
2      "Derived class for accounts receivable invoices"
3      def __init__(self, customer, amount):
4         self.company = customer
5         self.amount = amount
6         Invoice.invoiceNoCounter += 1
7         self.invoiceNo = Invoice.invoiceNoCounter
8         self.paid = False
9
10  class AccountsPayable(Invoice):
11     "Derived class for accounts payable invoices"
12     def __init__(self, vendor, amount, invoiceNo):
13        self.company = vendor
14        self.amount = amount
15        self.invoiceNo = invoiceNo
16        self.paid = False
```

Derived classes are declared similarly to their parent class; however the parent class to be inherited from is identified in parentheses after the class name. Both derived classes will share all of the same methods as the `Invoice` parent class, with the exception of the `__init__` which has been overriden in each of the derived class definitions. This is due to the fact that we want our accounts payable and accounts receivable objects to be initialized with different arguments. For example, note that initializing an `AccountsPayable` object requires the passing of an `invoiceNo` argument which is assigned to `self.invoiceNo`, while initializing the `AccountsReceivable` object assigns a `self.invoiceNo` value generated from the parent class `Invoice.invoiceNoCounter`. This is due to the fact that invoice numbers for the AR invoices should be generated internally, while the invoice numbers for the AP invoices need to be supplied by the vendors.

For an example of how these derived classes could be instantiated and how to call their methods, carefully review the following code excerpt and the related output presented in Figure 4.15.

```
invoice1 = AccountsReceivable("Acme Company", 1541.99)
invoice2 = AccountsPayable("XYZ Inc.", 4750.15, 2833303)
invoice1.display()
invoice2.changeAmount(4500.99)
invoice2.display()
```

Figure 4.15: Output for the modified invoice tracking program



4.9 Summary

We began this chapter by introducing functions, which are useful programming tools for writing reusable code. We then discussed lists and dictionaries, which are commonly used data structures for handling more complex data. We learned the basic steps involved in saving to and reading from files, including encoding/decoding using the JSON format. The chapter concluded with an introductory discussion of object oriented programming concepts including classes and inheritance.

4.10 Exercises

1. What is the difference between local versus global scoping of variables? Why do you think programmers generally prefer to avoid using variables with global scope?

2. Assume that we've created a list with the following assignment:
   `var myList = [1, 4, ["foo", 3, "bar"], "baz"]`
   Specify the output that would be returned by each of the following expressions:

   (a) `myList[0]`

   (b) `myList[2]`

   (c) `myList[:3]`

   (d) `myList]-2]`

(e) `myList[2][2]`

(f) `myList[::]`

(g) `4 in x`

(h) `myList.len()`

(i) `myList.index(4)`

3. Assume that we've created a list with the following assignment:
`var myList = [1, 4, ["foo", 3, "bar"], "baz"]`
Indicate how `myList` would look after calling each of the following methods (assume the methods are called sequentially).

   (a) `myList.pop()`

   (b) `myList.index(1, "bar")`

   (c) `myList.append("qux")`

   (d) `myList.remove(1)`

   (e) `myList.sort()`

4. The following code stub starts the definition of a class called `JournalEntry`. Objects instantiated using this definition should have the members `accountName` and `amount`. Complete the code stub by replacing the commented code `#Line 1` and `#Line 2`.

```
class JournalEntry:
    def __init__(self, accountName, amount):
        #Line 1:  Create a member for the account name
        #Line 2:  Create a member for the journal entry amount
```

5. In Exercise 3.6 you were asked to write a program which displays the first 25 numbers in the *Fibonacci sequence*. Modify this program so that this calculation and related printout is encapsulated in a function. The function should take a single parameter, `num`, which is specified by the user and indicates the number of elements in the sequence to print.

6. Write a program that will ask the user to enter a temperature in Fahrenheit and prints the temperature in Celsius. The program should contain a function which takes a single argument, `farTemp`, and returns the converted temperature. Note that the conversion formula is *C = (F - 32) * (5/9)*. As a check, 76 degrees Farenheit is equivalent to approximately 24 degrees Celsius.

7. Write a function that combines two lists of identical length by alternating each element and returns the combined list. For example, if the first list is `["foo", "bar", "baz", "qux"]` and the second list is `[1, 2, 3, 4]`, the function would return `["foo", 1, "bar", 2, "baz", 3, "qux", 4]`.

8. Write a function that takes two lists and return a new list that contains only the unique values from both lists. For example, if the first list is `[1, 2, 3, 4]` and the second list is `[6, 5, 4, 3]`, the function would return `[3, 4]`.

9. Create dictionary that contains a series of keys representing abbreviations of five US states (e.g., NY, MA, CA) whose related valuates are the city capitals of each state (e.g., New York City, Boston, Sacramento). Write a program that will prompt the user to enter the abbreviation of a US state. If the user enters the abbreviation of a state included in the dictionary, the program should print that state's capital. If the state is not included in the dictionary, the program should return a message indicating that fact.

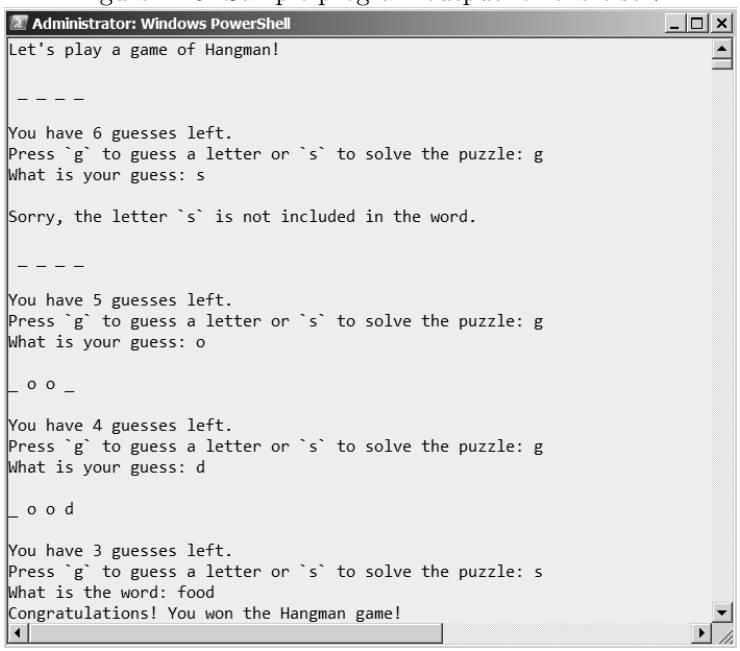
10. **Portfolio project:** Write a simple program that creates a database of your friends' birthdays. The program should give the user the options to 1) add a new friend and birthday to the database, 2) remove a friend and birthday from the database, 3) change the birthday for a friend already included in the database, 4) print the current friends and birthdays stored in the database, and 5) export the contents of the database to a JSON file. You should use conditional statements to manage user input. You may want to consider using a dictionary as the organizational structure supporting your database.

11. **Portfolio project:** The file `wordlist.txt` posted on the textbook website contains a list of 69,903 English words. Using this word list, create a game of Hangman using Python. The program should read the word list file and randomly choose a single word from the list. It should then provide the user with a limited number of chances to guess individual letters that make up the word. An example of what such a program might look like is presented in Figure 4.16.

    Hint: To help choose a random word from the list provided, consider using the `randint()` function which is provided as part of Python's `random` module. For example, `"number = random.randint(1,10)` will assign a random number between 1 and 0 to the variable `number`.

12. **Portfolio project:** Modify the example in Figure 4.11 so that it is interactive and also keeps track of a new invoice attribute - the payment due date. The user should have the ability to manually add new invoices and display the attributes of invoices that have already been entered. Hint: You will likely want to use a list data structure to store the invoice objects your program generates. For a further challenge, add the ability to delete specific invoices (it may be helpful to review some of the standard Python list methods that were discussed earlier in the chapter).

Figure 4.16: Sample program output for exercise 7



Chapter 5

Collecting data from the web

Websites often contain useful data for performing accounting analytic tasks. For example, a client may post certain relevant accounting data on an internal corporate website, or an auditor may want to gather pricing data from the external website of a vendor for purposes of performing valuation test work. Thankfully, it is relatively easy to write Python programs that can gather data from the web.

5.1 A brief overview of HTML

HTML is an acronym that stands for HyperText Markup Language. You can think of HTML as essentially a programming language for displaying information in a web browser. It is comprised of a collection of short statements (referred to as "tags") which tell the browser how website content should be presented to the viewer. At their most basic level, tags can apply simple formatting to website text. For example, the tag `<b>` can be used to make text appear bolded. Within a webpage's HTML document, this tag could be used as follows:

```
Hello!  <b> This text is bolded </b> but this text is not.
```

If a web browser were to then read this line of HTML code, it would display the following:

Hello! **This text is bolded** but this text is not.

Note that HTML tags are always encapsulated in angle brackets and the closing tag is always prefaced with a forward slash (/). HTML contains a wide variety of tags that can be used to display information in a myriad of different ways, such as including information in tables, or in graphics and videos. A brief list of some common HTML tags is presented in Figure 5.1. For our purposes, learning about HTML in great depth is beyond the scope of this book[1] That being said, once we understand how information used in a particular website is tagged, we can use that knowledge to gather data that is of interest to our analyses. For example, if we were interested in obtaining all of the information that is displayed on a website in **bolded** text, we could write a Python program that reads the website's HTML code and saves any data encapsulated in `<b>` tags. This is commonly referred to as "web scraping".

Figure 5.1: Common HTML tags

Tag	Description
`<HTML>`	The entire HTML document
`<HEAD>`	The header of the document
`<TITLE>`	The title of the document
`<BODY>`	The body part of the document
`<P>`	A paragraph
`<DIV>`	A custom container unit
`<H1>`	A section heading (subheadings are `<H2>` through `<H6>`)
`<IMG>`	An image
`<LI>`	A list item
`<TABLE>`	A table
`<TR>`	A table row
`<TD>`	A data cell for a table
`<THEAD>`	The header for a table
`<TH>`	A header cell for at table
`<A>`	A hyperlink

5.2 Necessary libraries

We will be writing our web scraper using the free Python library *Beautiful Soup* written by Leonard Richardson. The project's website, which includes the latest release and documentation, is located at:

[1] That being said, understanding HTML can be an invaluable skill for a budding data scientist. For beginners, I recommend Jon Duckett's book *HTML and CSS: Design and Build Websites* which is published by John Wiley & Sons.

`https://www.crummy.com/software/BeautifulSoup/`

To give Python the ability to access the web and pass HTML data to *Beautiful Soup*, we'll be using the free Python library *Requests*. The website for the library, which also includes the latest release and documentation, is located at:

`http://docs.python-requests.org/en/latest`

We'll be using the Python package management system, *Pip* to install both libraries. [2] *Pip* allows Python packages to be installed using the command:

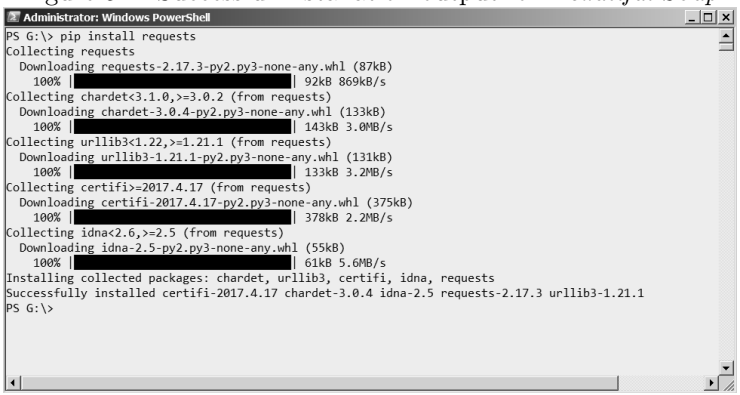
```
pip install package-hame
```

In our case, we'll be running the following two commands within PowerShell to install *Beautiful Soup* and *Requests*:

```
pip install beautifulsoup4
pip install Requests
```

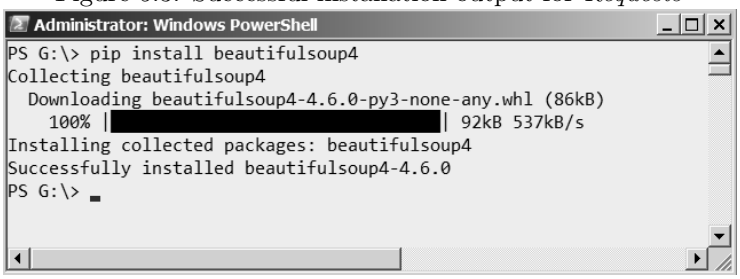
If the packages have installed correctly, *Pip* should display output similar to Figures 5.2 and 5.3.

Figure 5.2: Successful installation output for *Beautiful Soup*



[2] If you're using Python version 3.4 or later, *Pip* should already be installed on your machine. You can verify this by by typing `pip` from your terminal console. If for some reason *Pip* is not installed on your system, you can follow the installation instructions available at `https://docs.python.org/3/installing/`

Figure 5.3: Successful installation output for *Requests*



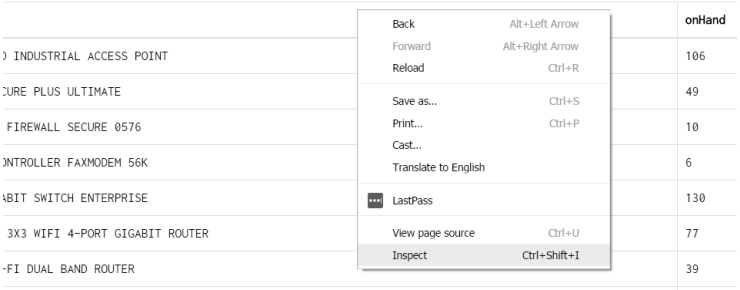
5.3 Viewing a webpage's HTML code

Prior to scraping the data from a web page, we need to understand how the data on that particular page is being tagged. We can do that by viewing the page's HTML source code within a web browser. In this example, we'll be scraping a website that contains a fictional company's ending inventory information. You'll note that this website lists inventory data across a series of three individual webpages which can be navigated using hyperlinks in the lower-left corner of the page. This website can be found at:

`http://www.steveperreault.com/textbook.inventory1.html`

Once you've navigated to the site, right click anywhere in the browser frame and open the *Inspector* pane, as demonstrated in Figure 5.4. The inspector tool allows us to see the underlying HTML code for any portion of a webpage that we click on. [3]

Figure 5.4: The *Inspector* menu option

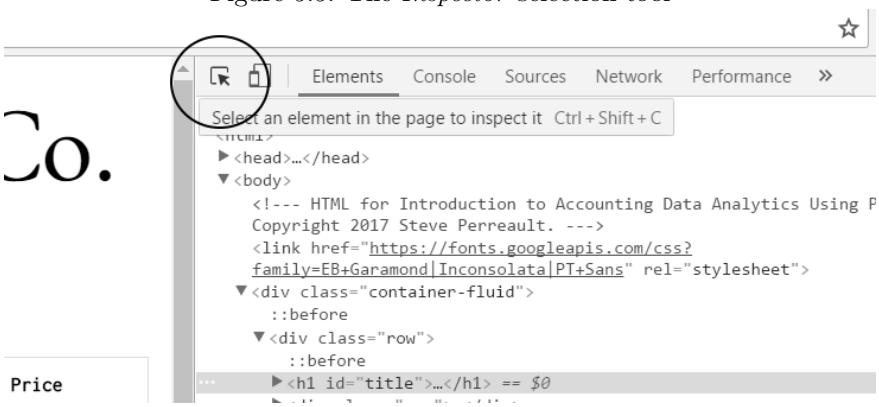


Activate the selection tool from within the *Inspector* pane (see Figure 5.5) and then click on any of the tabular inventory data contained in the browser pane.

[3]The option to select this tool may look slightly different if you are using another browser; in this case we are using Google Chrome.™

The `Inspector` page will change to display the HTML code associated with the inventory data (see Figure 5.6).

Figure 5.5: The *Inspector* selection tool



You will notice that the inventory table as a whole is encapsulated within `<table>` tags. Individual rows of a table are contained within `<tr>` tags. The first row of the table represents the header and is enclosed within `<thead>` tags and contains five cells for the table headings, each enclosed within `<th>` tags. The body of the table is encapsulated within `<tbody>` tags. Finally, cells containing standard data are enclosed within `<td>` tags. This is the standard scheme for tagging tabular data in an HTML document.

5.4 Building a basic webscraper using Python

Now that we understand how the data we are interested in has been tagged, we can write a Python program to collect that data using *Beautiful Soup* and *Requests*. To keep things understandable, we'll develop the application slowly in individual pieces.

First, we need to tell Python to import the *Requests* module so we have access to the tools we need to access the web.

```
import requests
```

Now, let's create a new variable, called `url` which will store the address of the webpage we want to scrape (represented as a string).

```
url = "http://www.steveperreault.com/textbook/inventory1.html"
```

Figure 5.6: The HTML source for the inventory data table

```
▼<table class="table table-bordered">
  ▼<thead> == $0
    ▼<tr>
        <th>Warehouse</th>
        <th>SKU</th>
        <th>Description</th>
        <th>onHand</th>
        <th>Price
              </th>
      </tr>
    </thead>
  ▼<tbody>
    ▼<tr>
        <td>C</td>
        <td>EN876923</td>
        <td>WAC104 DUAL BAND INDUSTRIAL ACCESS POINT</td>
        <td>106</td>
        <td>59.99</td>
      </tr>
    ▼<tr>
        <td>D</td>
        <td>EN687311</td>
        <td>3YR NSA 3800 SECURE PLUS ULTIMATE</td>
        <td>49</td>
        <td>6566.00</td>
      </tr>
```

Next, we will call the *Requests* method `get` which will return webpage data from the address we provide. We'll pass the `url` variable we just created to this method and store the returned data into a new object called `response`.

```
response = requests.get(url)
```

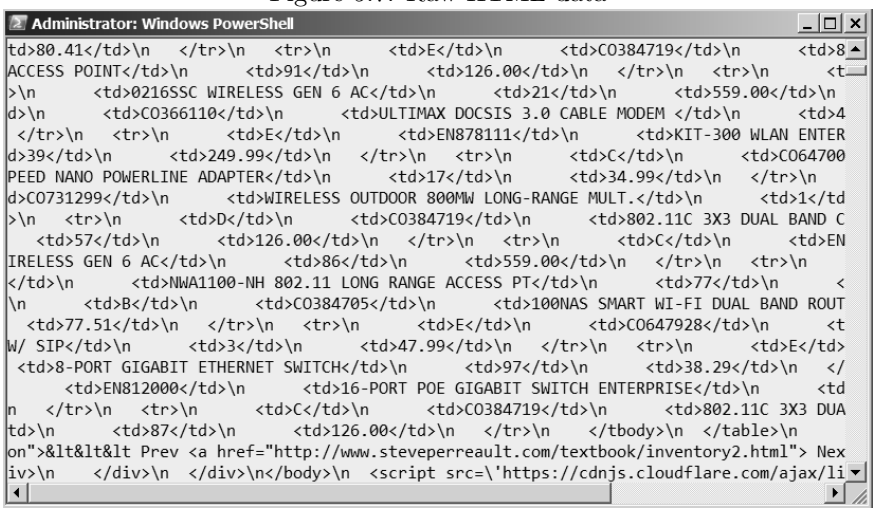
The HTML content of the webpage is stored in a method called `content`. We can verify that the object was created correctly by printing the contents of that method as follows.

```
print (response.content)
```

If we've done everything correctly, running this program should display the raw HTML from the webpage we are attempting to scrape, similar to that shown in Figure 5.7. While this is a good first step, we now need to modify the program to strip out the extraneous information, including the HTML tags, and display only the data we are interested in. *Beautiful Soup* has several helpful methods that will aid us in this effort.

First, let's modify our program to also import the *Beautiful Soup* methods we need. Note that the `from` prefix ensures that we are importing from the `bs4` module which is compatible with Python version 3+.

Figure 5.7: Raw HTML data




```
from bs4 import BeautifulSoup
```

We'll then create a new *Beautiful Soup* object from the `requests` object that was created earlier.

```
soup = BeautifulSoup(response.content)
```

This new `soup` object contains all of the HTML presented in the original document (you can `print()` its contents to verify this). We're only interested in capturing the data contained within the table and, as we learned earlier, that data is encapsulated within `<tbody>` tags. We can use the `find()` method to identify only data contained with the `tbody` tag and save that into a new object called `table`.

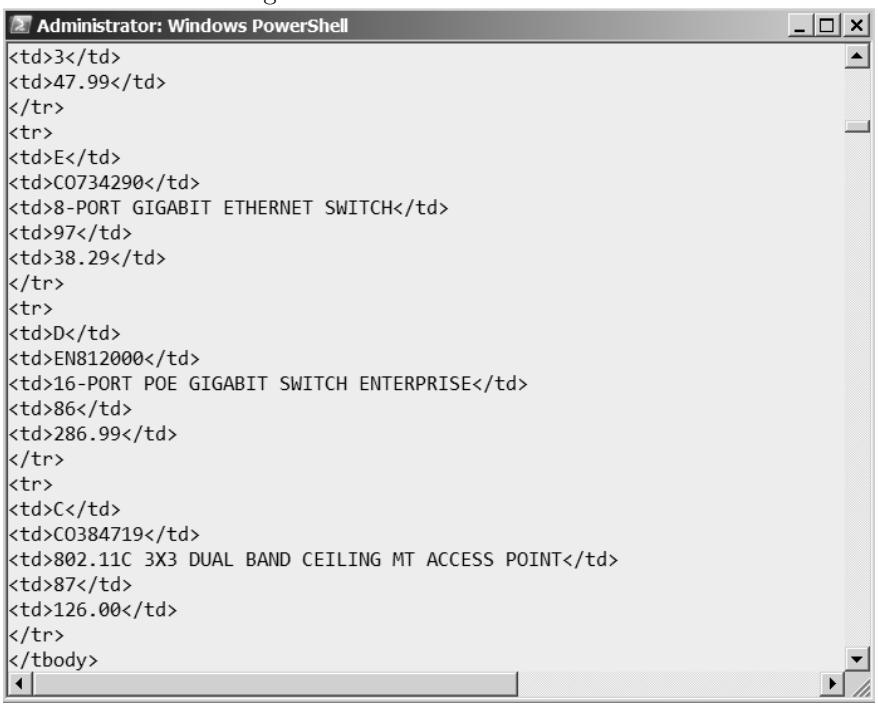
```
table = soup.find("tbody")
```

We will then call the object's `prettify()` method which formats the content by placing each HTML tag on its own line. Then we'll print out the contents of the `table` object so we can see what we've collected so far.

```
table.prettify()
print(table)
```

Again, if we've done everything correctly, the output of our program should look similar to that presented in Figure 5.8. Note that the program has only captured the data contained within the `tbody` tags – all of the other HTML code has been removed from the object.

Figure 5.8: "Prettified" table data



Remember that within our table body, rows are identified by the `<tr>` tag, while individual data cells are identified by the `<td>` tag. We want to take the data within each of these cells and store it in a data structure that can then be accessed and manipulated by Python. Thankfully, the list data structure that we learned about in Chapter 4 is perfect for storing tabular data. Let's go ahead and create a new list called `inventory` now.

```
inventory = []
```

We now need to tell our Python program to grab the data that is contained in the `table` object and place it into our new `inventory` list structure. Specifically, the program should read through the `table` object and, every time it encounters the `<tr>` tag, it should make a new row in the `inventory` list. Then, when it encounters data encapsulated within `<td>` tags, we want it to copy that data as a new element into the `inventory` list. This process should proceed until Python has processed the entire `table` object.

This sounds just like an iterative process which means we'll be able to apply the iteration concepts from Chapter 3. The specific loop we'll want to include in our program is presented in Figure 5.8.

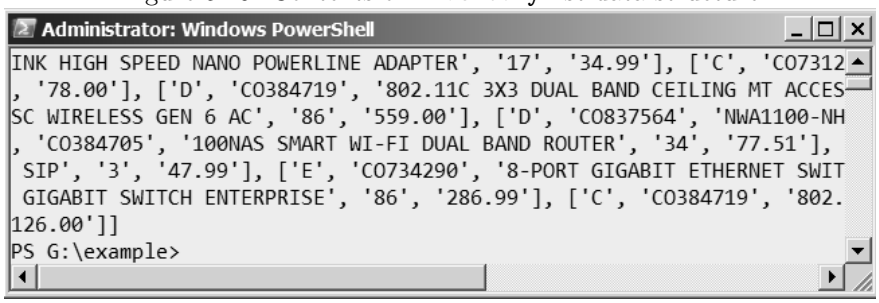
Figure 5.9: Copying data to a list

```
1 for row in table.findAll("tr"):
2    rowContents = []
3    for cell in row.findAll("td"):
4       rowContents.append(cell.string)
5    inventory.append(rowContents)
```

Note the first expression uses *Beautiful Soup's* `findAll` method to tell Python to iterate over all of the rows in the table which are identified by the `<tr>` tag. When a new row is found, the program then initializes a new list called `rowContents` which will be used to store the data elements in that row. On line 3, a second `for` loop is used to iterate over all of the cells in the row chich are identified by the `<td>` tag. Every time a new cell is encountered, its content (contained within the `string` member) is appended to the end of the `rowContents` list. Finally, when all the cells in a row have been processed, `rowContents` is appended to the `inventory` list. The process then starts again for the next row in the table.

After the completion of this process, the `inventory` list data structure should contain the inventory data that were are interested in. If we were to display its contents, the output would look similar to that presented in Figure 5.10.

Figure 5.10: Contents of `inventory` list data structure



It is likely the case that we will want to save the contents of our list to a file so they can be used in other programs. We could write the code to do this ourself, iterating over the elements in the list and saving each to a text file (we learned how to do this in Chapter 4) However, Python contains a handy built-in module called `csv` which can more easily output the contents of our `inventory` data structure into a comma-delimited format. We're going to use this module now to save us some code.

First, we'll import the module, create a new file object called `output` and asso-

ciate the file object with the file `inventory.csv`[4] Recall that the `a` parameter ensures that a new `inventory.csv` will be created if one does not exist already. This was covered earlier in Chapter 4.

```
import csv
output = open("inventory.csv", "w", newline="")
```

We then call the `writer` method which will create a new object writerObj that will be used to write data to the file object we just created.

```
writerObj = csv.writer(output)
```

Finally, the method `writerows` is used to write all the rows in the `inventory` list to the CSV file we specified.

```
writerObj.writerows(inventory)
```

Executing this program will create a new file in our working directory `inventory.csv` which contains a comma delimited list of all of the inventory data from the website. We can open the file in a spreadsheet program such as Microsoft Excel™to verify that its contents.

There is still one last thing that we're missing - the table headers. Recall that we only told *Beautiful Soup* to parse data within `<tbody>` tags; however, the table's column headers are contained within `<thead>` tags. We could write some additional code to tell `Beautiful Soup` to gather that data for us and then add it to the `inventory.csv` file following the steps we just reviewed. However, unless you are working with a table that has a very large number of columns, it is often easier to simply insert the table headers into the CSV file manually. We can do this using the `writerow` method which will write a single row of data to the `inventory.csv` file. Be sure to make this call prior to writing the rest of the tabular data to the file.

```
writerObj.writerow(["Warehouse", "SKU", "Desc", "onHand", "Price"])
```

The full source code for the program so far and the related CSV file (as viewed in Microsoft Excel) are presented in Figures 5.11 and 5.12, respectively.

5.5 Scraping multi-page websites

While the program we have written does an excellent job scraping a single webpage, our inventory list is not yet complete. Recall that the complete

[4] `.CSV` is the common file extension for comma-delimited files.

Figure 5.11: A basic web scraper

```
1   import requests
2   from bs4 import BeautifulSoup
3   import csv
4
5   url = "http://www.steveperreault.com/textbook/inventory1.html"
6
7   response = requests.get(url)
8
9   soup = BeautifulSoup(response.content)
10
11  table = soup.find("tbody")
12
13  inventory = []
14
15  for row in table.findAll("tr"):
16     rowContents = []
17     for cell in row.findAll("td"):
18        rowContents.append(cell.string)
19     inventory.append(rowContents)
20
21  output = open("inventory.csv", "w", newline="")
22  writerObj = csv.writer(output)
23  writerObj.writerow(["Warehouse", "SKU", "Desc", "onHand", "Price"])
24  writerObj.writerows(inventory)
```

inventory list is spread across a series of three sequentially numbered webpages(`inventory1.html, inventory 2.html,` and `inventory3.html`) and that the scraper we have written only collects the data included on the first page.[5].

We could address this issue creating three separate programs that each contain modified versions of the `url` string which point to the three webpages we want to gather data from. However, there is a more elegant solution to this problem which uses iteration to grab the data that we need.

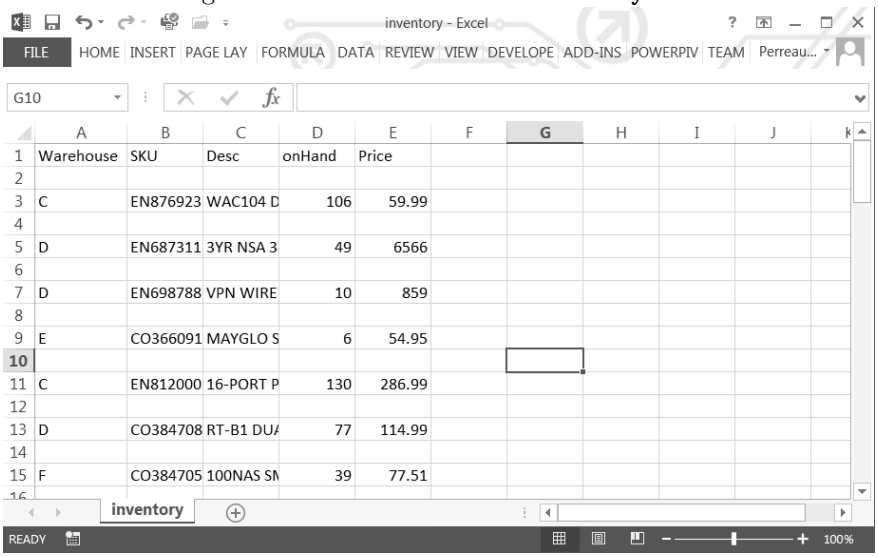
Let's first modify the string `url` as follows:

```
url = "http://www.steveperreault.com/textbook/inventoryX.html"
```

We'll then encapsulate the code which gather ands writes the data to the `inventory` list within a `for` loop which we will set to run a series of three times (once for each webpage we want to gather data from):

[5]It is often the case that, when webpages contain data which spans across multiple pages, the webpage names will be numbered sequentially

Figure 5.12: The contents of `inventory.csv`




```
for i in range(1,4):
```

Each time the loop executes, we will use string indexing to create a new url string (`newUrl`) which replaces the `X` character with the iteration count as follows:

```
newUrl = url[0:-6] + str(i) + url [-5:]
```

Note that this expression is slicing the `url` string six spaces before the end (i.e., removing `.html` portion), inserting the iteration count which is held in the integer `i`, and then appending the last five characters of `url` (`.html`) to the end of the new string.

There's a couple of additional changes we need to make. First we need to move the list constructor `inventory = []` outside of the loop because we don't want it to be overwritten each time the loop iterates. In addition, we need to make sure that the creation of the `writerObj` object and calls to its methods occur after the loop has finished executing. This is because we only want to write the `inventory` list structure to the `CSV` file after the data has been fully collected.

The complete and final source code for our basic webscraper is presented in Figure 5.13.

Figure 5.13: A basic web scraper with support for loading multiple webpages

```
import requests
from bs4 import BeautifulSoup
import csv

url = "http://www.steveperreault.com/textbook/inventoryX.html"

inventory = []

for i in range(1,4):
   newUrl = url[0:-6] + str(i) + url[-5:]
   print("Gathering data from " + newUrl)
   response = requests.get(newUrl)
   soup = BeautifulSoup(response.content)

   table = soup.find("tbody")

   for row in table.findAll("tr"):
      rowContents = []
      for cell in row.findAll("td"):
         rowContents.append(cell.string)
         inventory.append(rowContents)

output = open("inventory.csv", "w", newline="")
writerObj = csv.writer(output)

writerObj.writerow(["Warehouse", "SKU", "Desc", "onHand", "Price"])
writerObj.writerows(inventory)
```

5.6 Summary

The chapter begin with a discussion of common HTML tags and reviewed how the HTML code of a webpage can be viewed. It then discussed how to use the *Beautiful Soup* and *Requests* libraries to build a simple Python program to scrape tabular data from single-page websites. Use of iteration to scrape from multi-page websites was also discussed.

5.7 Discussion Questions

1. What does HTML stand for? What do the terms that make up this acronym refer to?

2. Imagine you are developing a web scraper and want to all of the image data

contained within a webpage. How would you modify the code presented in Figure 5.13 to gather such data?

5.8 Exercises

1. **Portfolio Project**: Develop your own interactive web scraper. The program should prompt the user to type a URL address and then generate a CSV file which contains all tabular data contained within the page. Test that your program works by finding a webpage that contains data in tabular form and providing it to the program you have written.

   Wikipedia, in particular, is a rich source of tabular HTML data. A few examples of Wikipedia pages containing such data are:

   - List of cities by population
     https://en.wikipedia.org/wiki/List_of_cities_proper_by_population
   - List of countries by median age
     https://en.wikipedia.org/wiki/List_of_countries_by_median_age
   - List of countries by labor force size
     https://en.wikipedia.org/wiki/List_of_countries_by_labour_force

Chapter 6

Working with tabular data

6.1 Installing Pandas

We'll be using the *Python Data Analysis Library* (Pandas) module to help us analyze tabular data. Pandas adds data frame functionality to Python, similar to what is included in languages such as R. Data frames are similar to the traditional spreadsheets that you are familiar with, in that they allow for mixed data types to be presented within labeled rows and columns. Pandas also contains a wide variety of analysis tools that are useful in accounting analytics contexts. The project's website, which includes the latest release and documentation, is located at:

```
https://pandas.pydata.org
```

Similar to how we have previously installed other Python modules, we'll be installing Pandas from with Powershell using the command:

```
pip install pandas
```

In addition to Pandas, the installer should also install the numpy module, upon which Pandas is based. When writing code that will use Pandas functionality, we will always need to import both modules as follows:

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

6.2 Constructing data frames manually

A data frame is simply a table with labeled rows and columns. Let's imagine that we want to create a data frame using the inventory data that we collected in the previous chapter. We would begin by first creating a dictionary where each column in our table is included as a key along with a corresponding list of values (refer back to Chapter 4 for a discussion of dictionaries). For example:

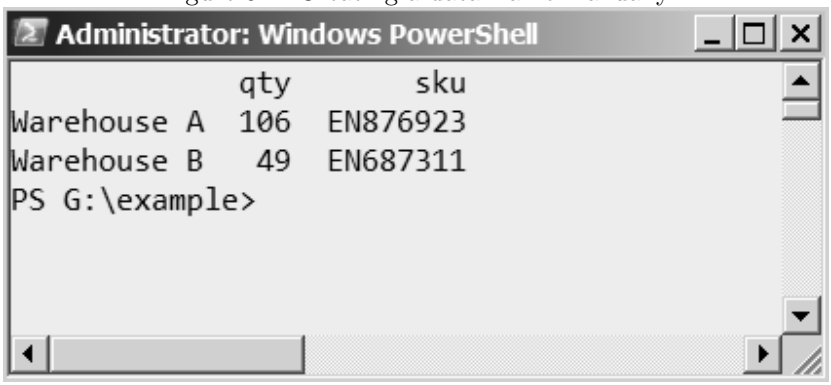
```
data = {"sku":  ["EN876923", "EN687311"], "qty":  [106, 49]}
```

We can then create the data frame by calling the `DataFrame` function and passing the dictionary as well as a list representing the headers for each row:[1]

```
inventory = pd.DataFrame(data, index=["Warehouse A", "Warehouse B"])
```

If we then print out the `table` data frame, we would see the following:

Figure 6.1: Creating a data frame manually



6.3 Constructing data frames from .CSV

Of course, you will rarely want to create a data frame manually. Rather, you will typically populate your data frames using an external source, such as a CSV file. Let's create a new data frame from the `inventory.csv` data file we prepared in the previous chapter. To do so, we would call Pandas' `read_csv` function as follows:

[1]The `pd` prefix to the `DataFrame` function indicates that we are calling the function contained within the Pandas namespace. This prefix should always be used before calling a Pandas function.

```
inventory = pd.read_csv("inventory.csv")
```

Note that, by default, Pandas will assign sequential numeric row headers, similar to a typically spreadsheet. These header values are referred to as the data frame's "index."

6.4 Printing data frames

Like other data types, Pandas data frames can be printed to the screen by passing them to the `print()` function. We may occasionally want to print a portion of a frame to screen to ensure that it has been set up properly. To avoid printing the entire contents of the data frame, we can call the `head()` which, by default, returns the first five rows of data. `t`his method can also take the parameter `n=X`, where X represents a custom number of rows to be printed.

6.5 Changing frame indices

Any column within a data frame can become the frame's index by using the `reset_index()` and `set_index()` functions. By passing the `drop=True` parameter to `reset_index` we remove the existing sequential numeric index from the data frame. We then call `set_index` with the name of the column header representing the new index as the sole parameter.

For example, let's say we want to index our `inventory` data frame by the name of the warehouse the inventory is located in. This could be done with the following expression:

```
newInventory = inventory.reset_index(drop=True).set_index("Warehouse")
```

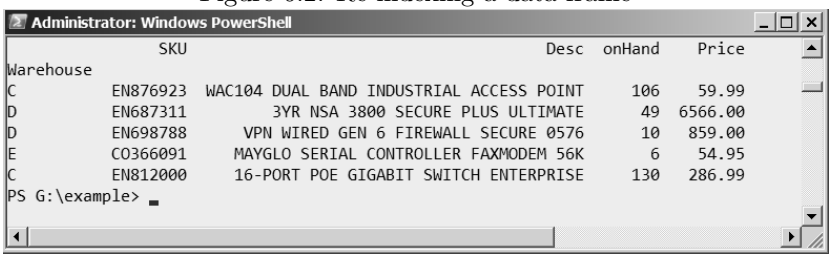
To verify that our re-indexing worked as intended, we could print the first five rows of the new data frame as follows:

```
print(new.Table.head())
```

The respective output is presented in Figure 6.2.

That being said, modifying a data frame's index can present some problems if we try to transform our data frames in the future. As a result, we'll stick with the original sequential numeric index for the examples presented in this chapter.

Figure 6.2: Re-indexing a data frame



6.6 Sorting data

Data frames can be sorted by index using `sort_index()`. This method will sort in ascending order; however a frame can be sorted in descending order by passing the `ascending = False` parameter. By default, the method returns a new data frame, however you can tell Pandas to sort and overwrite the existing data frame by passing `inplace = True`. For example:

```
inventory.sort_index(inplace=True, ascending=False)
```

Pandas can sort data frames by data value using the `sort_values()` method and passing the header of the column to sort by. For example, if we wanted to sort the `inventory` data frame by price, we could use the following expression:

```
inventory.sort_values("Price", inplace=True, ascending=False)
```

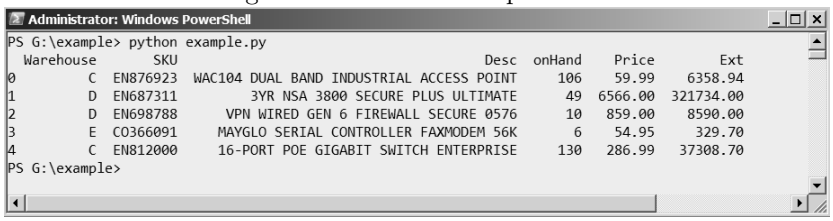
6.7 Arithmetic operations

Pandas also provides support for the four basic arithmetic operations (addition, subtraction, multiplication, and division). Let's say that we want to take the `onHand` and `Price` columns from the `inventory` data frame, multiply the two, and create a new column that represents the extended price.

```
inventory["Ext"] = inventory.Price * inventory.onHand
```

Note that the term in brackets represents the header for the new column and the and the calculation is performed using the standard Python multiplication operator. Other arithmetic operations can be performed using the various other operators. The output for this example if provided in Figure 6.3.

Figure 6.3: Arithmetic operations



6.8 Aggregating data

When working with accounting information, we will often find the need to summarize or aggregate data. For example, perhaps we want to group sales of individual products by category or summarize the value of journal entries affecting a particular account. Pandas contains a function called `groupby()` which allows us to perform this task easily. We just need to pass the the column to aggregate by as well as an aggregation function which can be selected from the options presented in Figure 6.4.

Figure 6.4: A sample of common `groupby()` aggregation functions

Function name	Description
`sum()`	Sum the numerical rows in the group
`prod()`	Multiply the numerical rows in the group
`mean()`	The average of the numerical rows in the group
`count()`	The number of rows in the group

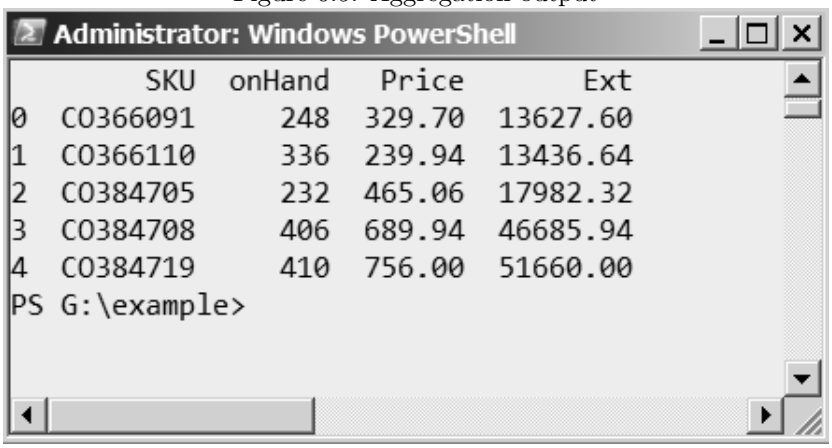
For example, let's say we want to identify the total quantity of inventory items contained in the `newInventory` data frame across all warehouses. Since each item has a unique SKU number, we could use `groupby()` to aggregate inventory by this identifier and aggregate the sum of the grouped items to a new dataframe. We would use the following expression:

```
sumInventory = inventory.groupby("SKU").sum().reset_index()
```

Note that this requires us to also reset the numeric sequential index to fit the new data frame using `reset_index`. Omitting this call will default to using the first column of data as the index, which could present problems with transforming the frame going forward.

Printing the `sumInventory` data frame would yield the output presented in Figure 6.4.

Figure 6.5: Aggregation output



6.9 Merging data frames

Note that the contents of the `sumInventory` data frame omit the project description and warehouse name. This is because both values are non-numeric and, thus, cannot be aggregated using the `sum()` aggregation function. Fear not - we insert the description of each SKU back into `sumInventory` by merging it with the `newInventory` data frame.

The `merge()` method can easily be used to combine rows that share a common index. As parameters, it takes the two frames to be merged, and the name of the column to be used to perform the match. If the column index has the same name within both names, you can simply pass that name, as follows:

```
table = pd.merge(frame1, frame2, on="index")
```

If the column index has a different name, you can specify the name of the index to be used as follows:

```
table = pd.merge(frame1, frame2, left_on="index1", right_on="index2")
```

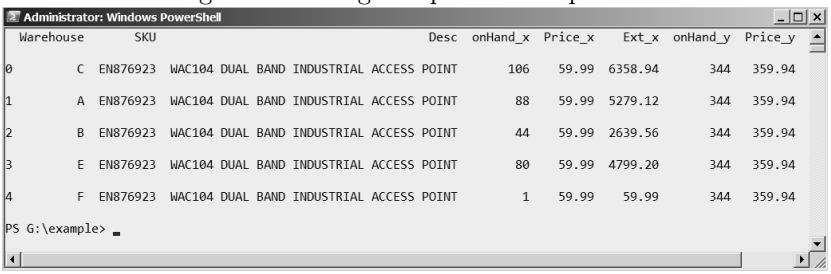
Note that, within this expression, the left and right keys refer to the first and second parameters passed to the method, respectively.

With this knowledge, we can now merge the contents of the `sumInventory` and `inventory` frames into a new data frame as follows:

```
sumInventoryExt = pd.merge(inventory, sumInventory, on="SKU")
```

Printing the output of sumInventoryExt will return the following:

Figure 6.6: Merge output with duplicates



A review of this output indicates that this is not quite what we want. It seems like each row of `sumInventoryExt` contains multiple columns for the `onHand`, `Price`, and Ext variables and that Pandas has attached either a `_x` or `_y` suffix to these columns. This is because the `sumInventory` data frame contains the inventory listing at the summarized level, while `inventory` does not. As a result, Pandas cannot find key matches within these three columns across the data frame and so it includes data from both frames when creating `sumInventoryExt`. This is not helpful to us, so let's do something about it.

6.10 Deleting data and removing duplicates

Let's create a new data frame called `skuList` which contains only a listing of individual SKU numbers and their respective conditions. We'll build this new frame from the `inventory` frame, eliminating the data that we don't need using the following commands:

```
skuList = inventory.drop("Warehouse", 1)
skuList.drop("Price", 1, inplace = True)
skuList.drop("onHand", 1, inplace = True)
skuList.drop("Ext", 1, inplace = True)
```

The first expression creates a new data frame which is set to the same value of the `inventory` frame with the `Warehousn` column removed. The parameter texttt1 which is passed to the `drop` method tells Pandas that `Warehouse` is a column label (passing a value of 0 would denote a row label). The next three lines of code remove the remaining unneeded columns from the `skuList` frame. Recall that the parameter `inplace = True` tells Pandas that the existing `skuList` frame should be modified.

Alternatively, we could construct the new frame `skuList` and simply include only the `SKU` and `Desc` columns from the `inventory` frame as follows:

```
skuList = inventory[["SKU","Desc"]]
```

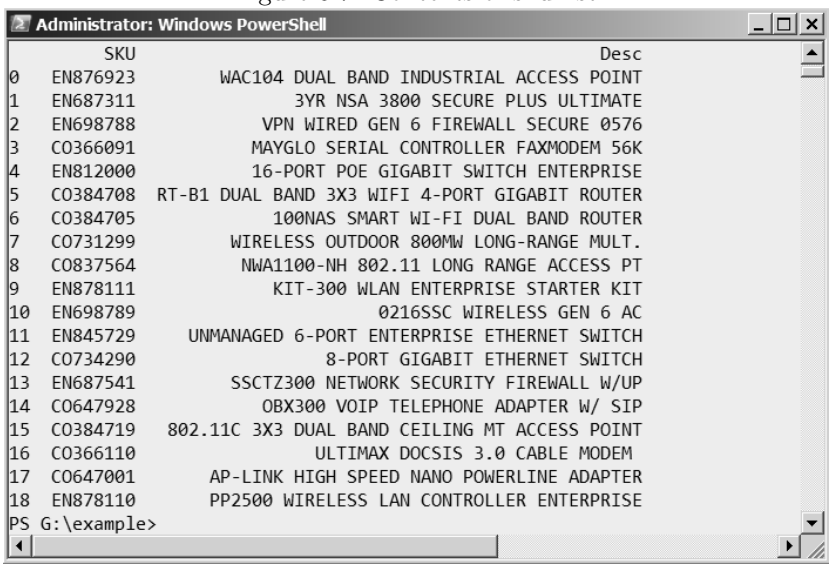
Either way, the end result is a new data frame called `skuList` which includes only two columns labeled `SKU` and `Descr`. However, this new frame still contains a large number of duplicate rows which we should remove using the `drop_duplicates()` method.

```
skuList = skuList.drop_duplicates().reset_index(drop=True)
```

Note that we also use the `reset_index()` method to drop the old index and build a new one based upon the new number of rows in the data frame.

Printing out this new table should result in output similar to Figure 6.7, which represents a list of unique inventory SKUs and their related descriptions.

Figure 6.7: Contents of skuList



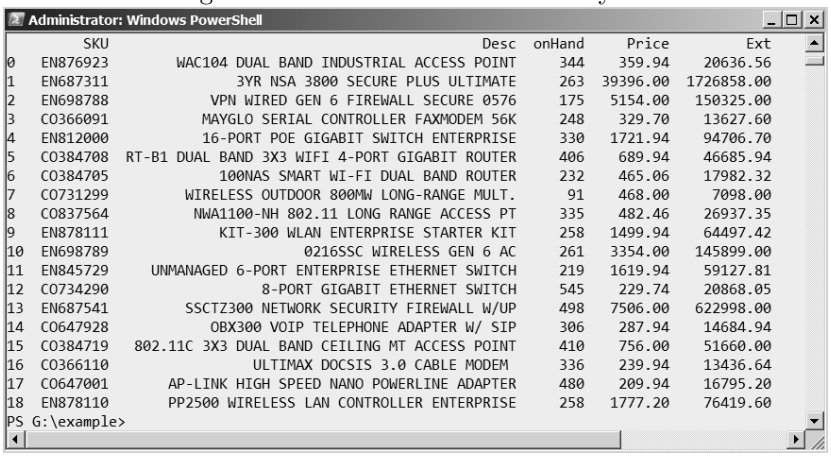
We can now merge `skuList` with the `sumInventory` frame we created earlier to created a summarized inventory listing which also includes SKU numbers and product descriptions as follows:

```
sumInventoryExt = pd.merge(skuList, sumInventory, on="SKU")
```

We can then print out the contents of `sumInventoryExt` to verify that the merge was performed properly. This should result in output that looks similar

to Figure 6.8.

Figure 6.8: Contents of sumInventoryExt



	SKU	Desc	onHand	Price	Ext
0	EN876923	WAC104 DUAL BAND INDUSTRIAL ACCESS POINT	344	359.94	20636.56
1	EN687311	3YR NSA 3800 SECURE PLUS ULTIMATE	263	39396.00	1726858.00
2	EN698788	VPN WIRED GEN 6 FIREWALL SECURE 0576	175	5154.00	150325.00
3	CO366091	MAYGLO SERIAL CONTROLLER FAXMODEM 56K	248	329.70	13627.60
4	EN812000	16-PORT POE GIGABIT SWITCH ENTERPRISE	330	1721.94	94706.70
5	CO384708	RT-B1 DUAL BAND 3X3 WIFI 4-PORT GIGABIT ROUTER	406	689.94	46685.94
6	CO384705	100NAS SMART WI-FI DUAL BAND ROUTER	232	465.06	17982.32
7	CO731299	WIRELESS OUTDOOR 800MW LONG-RANGE MULT.	91	468.00	7098.00
8	CO837564	NWA1100-NH 802.11 LONG RANGE ACCESS PT	335	482.46	26937.35
9	EN878111	KIT-300 WLAN ENTERPRISE STARTER KIT	258	1499.94	64497.42
10	EN698789	0216SSC WIRELESS GEN 6 AC	261	3354.00	145899.00
11	EN845729	UNMANAGED 6-PORT ENTERPRISE ETHERNET SWITCH	219	1619.94	59127.81
12	CO734290	8-PORT GIGABIT ETHERNET SWITCH	545	229.74	20868.05
13	EN687541	SSCTZ300 NETWORK SECURITY FIREWALL W/UP	498	7506.00	622998.00
14	CO647928	OBX300 VOIP TELEPHONE ADAPTER W/ SIP	306	287.94	14684.94
15	CO384719	802.11C 3X3 DUAL BAND CEILING MT ACCESS POINT	410	756.00	51660.00
16	CO366110	ULTIMAX DOCSIS 3.0 CABLE MODEM	336	239.94	13436.64
17	CO647001	AP-LINK HIGH SPEED NANO POWERLINE ADAPTER	480	209.94	16795.20
18	EN878110	PP2500 WIRELESS LAN CONTROLLER ENTERPRISE	258	1777.20	76419.60

PS G:\example>

6.11 Descriptive statistics

Pandas supports a number of description statistics functions that can be applied to rows or columns in a data frame. We can apply the parameter `1` or `0` to which tells the function whether to perform the calculation using a column or row, respectively (Pandas assumes `1` if no value is provided). For example, if we wanted to calculate the average price for the inventory items contained within the `sumInventoryExt` data frame, we could use the following expression:

```
print(sumInventoryExt["Price"].mean())
```

A sample of some of the more common descriptive functions is presented in Figure 6.9

In addition, the Pandas function `describe()` can be used to obtain some quick summary statistics. It can be called as follows:

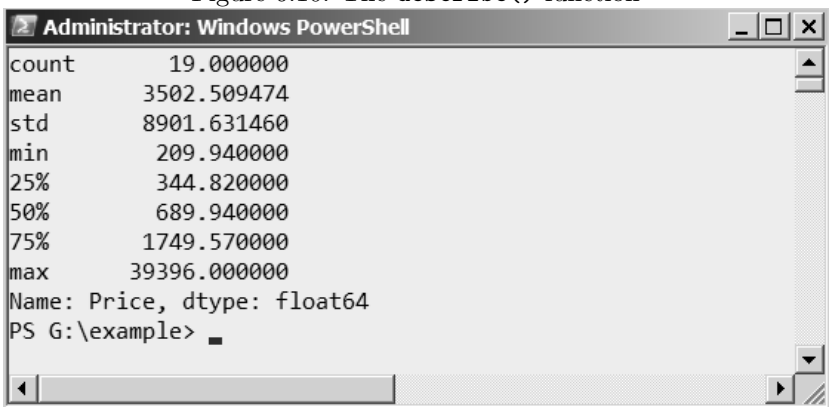
```
print(sumInventoryExt["Price"].describe())
```

An example of its output is provided in Figure 6.9.

Figure 6.9: A sample of common descriptive functions

Function name	Description
`mean()`	Mean
`median()`	Median
`mode()`	Mode
`std()`	Standard deviation
`sem()`	Standard error of the mean
`skew()`	Skewness
`kurt()`	Kurtosis

Figure 6.10: The `describe()` function



6.12 Cross-tabulation

Pandas can also create cross-tabulations, which are joint frequency distributions with rows and columns representing different values of two categorical variables. We can use cross-tabs to identify potential relationships between the variables (or factors) being examined.

For example, let's say that we want to understand whether the quantity of an inventory item on hand is related to the price of that item. Our initial prediction is that this relationship would be a negative one; that is, the more expensive the item, the fewer the quantity we would expect in inventory.

Since our data needs to be categorical, we'll create two new data frames which indicate whether the inventory items have above or below average prices and quantities. This can be done using the descriptive functions we discussed in the previous section:

```
price_cat = inventory["Price"] > inventory["Price"].mean()
quant_cat = inventory["onHand"] > inventory["onHand"].mean()
```

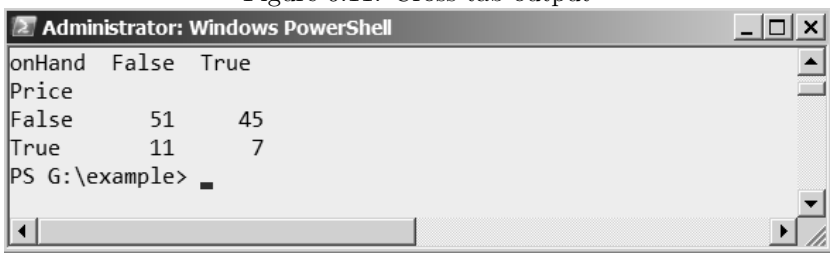
Printing out either of these two data frames will return a list of true/false values which indicate whether the specific inventory item met the criteria we specified (i.e., a value of `True` in the `price_cat` frame would indicate that a specific inventory item has a price that is above the average).

We'll then create and print a cross-tabulation of the joint frequencies of the two factors using the `crosstab` function as follows:

```
print(pd.crosstab(price_cat, quant_cat))
```

The related output is provided in Figure 6.11.

Figure 6.11: Cross-tab output



The results of this analysis do not seem to support our prediction. For the inventory items that have a higher than average price (`Price=True`) there are more individual items that have a below average quantity than an above average quantity (11 versus 7). Furthermore, this same relationship appears to exist for the inventory items that have a lower than average price (`Price=False`, 51 versus 45).

Note, however, that the frequency distribution across our four cells suggests that our data is highly skewed and that we should interpret this result with a great degree of caution. That is, it appears that there are some abnormally high priced items in inventory that are dramatically inflating the average price (this can be confirmed by reviewing the data or calling the `skew()` function discussed earlier). We'll learn how to deal with data outliers such as this in the next chapter.

6.13 Summary

In this chapter we discussed how to work with tabular data using Pandas data frames. We reviewed how data frames can be created (either manually or by importing from CSV files) and how their indices can be manipulated. We learned how to sort, aggregate, and perform arithmetic operations on data frame values. We learned how data values can be aggregated based upon common keys and how to delete unecessary data and remove duplicates from a frame. Functions that can be used to generate descriptive statistics from data frame values were then presented. The chapter concluded with a brief discussion of how Pandas can generate a cross-tabulation report.

6.14 Exercises

1. As you know, Pandas is a software library for performing data science using the Python programming language. Conduct some brief internet research to determine if alternatives to Python/Pandas exist for performing data science. If so, name these alternatives.

2. Conduct some brief internet research regarding the history of the Pandas library. Who is responsible for its development? How is the library distributed?

3. Manually create a data frame that contains a list of the four most recent Accounting Standards Updates (ASUs) that have been issued by the FASB (this information can easily be found via the FASB's website at `http://www.fasb.org`). The data frame should contain columns which list the ASU number, its title, and the the year that it was issued. Print the frame to screen to ensure that it was constructed properly.

4. Modify the data frame you prepared for question 1 so that the row index labels reflects the year the ASU was issued. Again, print the frame to screen to ensure that this change was properly made.

5. Modify the data frame you prepared in questions 1 and 2 by adding an addition column called `effectiveDate` which reflects the effective date of the ASU. Again, print the frame to screen to ensure that the new addition is appropriately included.

6. **Portfolio project**: You are interested in understanding whether people, on average, consume more alcohol in climates that are cold versus those that are warm. Perform an analysis to see if there is a relationship between per capita alcohol consumption and average temperature. You should follow these steps:

- Using the web scraper you developed in Chapter 5, download the average yearly temperature by country as a CSV file from https://en.wikipedia.org/wiki/List_of_countries_by_average_yearly_temperature
- Using the same method, download the per capita alcohol consumption by country from https://https://en.wikipedia.org/wiki/List_of_countries_by_alcohol_consumption_per_capita
- Import the data into two data frames, performing any data clean up that is necessary.
- Using Pandas' descriptive statistics functions, determine the average amount of alcohol consumed across all countries. Does the alcohol consumption data vary significantly from the mean (you will likely need to use the `std()` function to answer this question).
- Use cross-tabulation to determine if alcohol consumption patterns (above or belove average) might be related to local climate.

Chapter 7

Generalized linear models and forecasting

In this chapter we will discuss how to use Python to conduct a simple forecasting analysis using linear regression. Linear regression is statistical technique used to determine the strength of a relationship between two or more dicrete variables. It is particularly useful for developing forecasts and, as such, is widely used in accounting contexts, especially in auditing and managerial accounting. This chapter assumes that you already have a basic understanding of inferential statistics and linear regression from your earlier college coursework and will focus primarily on the technical skills needed to conduct a regression analysis in Python[1].

In the examples presented in this chapter, we well be using linear regression to examine the relationship between the credit rating that a company assigns to its customers and the timeliness of customer payment. We will then use this insight to develop a model that will use a customer's credit rating to estimate how quickly it will pay off its account.

7.1 Preparing our data

To begin, scrape the contents of the accounts receivables transaction detail located at `http://www.steveperreault.com/textbook/receivables.htm` to a CSV file following the steps outlined in Chapter 5. Then save the import the

[1] A basic level of understanding regarding regression can be obtained from most entry-level college statistics textbooks

data into a new Pandas dataframe as discussed in Chapter 7 (use the default sequential numerical row index).

The data dictionary for this data set is as follows:

- `amount`: the amount of the receivable
- `inv_date`: the date the invoice was generated
- `due_date`: the date that payment of the invoice is due
- `paid_date`: the date the invoice was paid
- `company_name`: the name of the customer responsible for paying the invoice
- `street`: the customer street address
- `city`: the city in which the customer is located
- `state`: the state in which the customer is located
- `score`: a custom code (between 0-100, in increments of ten) which indicates the perceived creditworthiness of the customer based upon evaluations made by the *Roger Wilco Company* credit and collections department. Higher numbers suggest higher perceived creditworthiness.

Let's add two custom columns to our data frame. First, we want to add a column that indicates the age of the receivable at the time it was paid (in days). Essentially, we need to calculate the difference between the invoice date and the date it was subsequently paid. We might be tempted to first try an arithmetic operation like this (assume we've named our data frame `recs`):

```
recs["Age"] = recs["paid_date"] - recs["inv_date"]
```

You will note that this returns the following error:

```
unsupported operand type(s) for -:  'str' and 'str'
```

Python is telling us here that the columns `paid_date` and `receivables_date` have a string type and that we can't perform substraction on strings (d'oh!). To fix this problem, we need to inform Python that these two columns (as well as `"due_date"`) should be treated as dates. Thankfully, Python has a handy function called `to_datetime` that can be used to convert strings data types to dates. In this example, it can be used as follows:[2]

[2]We could also convert the date columns when initially reading the CSV file by passing the `parse_date=True` and `keep_date_col=True` parameters to the `read_csv` function after the filename.

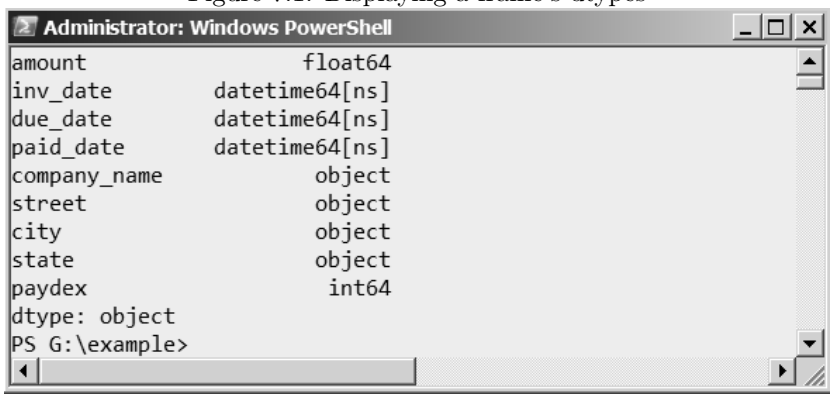
```
receivables["inv_date"] = pd.to_datetime(recs["inv_date"])
recs["due_date"] = pd.to_datetime(recs["due_date"])
recs["paid_date"] = pd.to_datetime(recs["paid_date"])
```

We can confirm that the appropriate data type has been assigned for each column by printing the frame's `dtypes` as follows:

```
print(rec.dtypes)
```

The related output is presented in Figure 7.1. The data types `float` and `int` should be familiar to you now, but you'll notice that the output also includes two new data types, `datetime` and `object`.

Figure 7.1: Displaying a frame's dtypes



The `object` data type is simply Panda-terminology for strings. Note thas type has been appropriately assigned to the `company_name`, `street`, `city`, and `state` columns. The `datetime` type is simply a special version of the string data type that Pandas uses when working with dates. We can confirm that our date fields `inv_date`, `due_date`, and `paid_date` appropriately have this type. Note that it's usually a good idea to verify the `dtypes` of your data frames prior to performing data analysis.

With our data columns now having the appropriate type, we can create a new column, `Age`, which reflects the difference between the invoice data and the payment date as follows:

```
recs["age"] = recs["paid_date"] - recs["inv_date"]
```

By default, Pandas will give the new `age` column a special data type called `timedelta`, which is a subclass of the `datetime` type. The `timedelta` time has advanced functionality that can be used to easily express differences between dates or times by seconds, hours, days, minutes, etc.

Next, we want to create an additional column which indicates whether the invoice was paid late. This variable will contain a binary value (`true` or `false`), so it is perfect for the `bool` type discussed earlier. We'll write a simple expression that determines whether the value in the `paid_date` field is greater than the value in the `due_date field`. The return value (either `True` or `False`) will be stored in a new column within the data frame called `Late`.

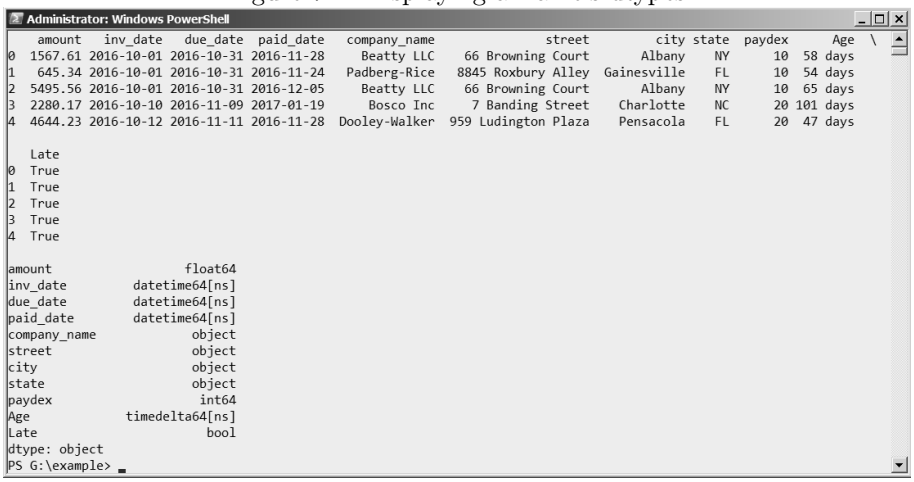
```
recs["late"] = recs["paid_date"]>recs["due_date"]
```

To verify that we've done everything correctly, let's print the first 5 rows of the data frame and the frame's `dtypes` as follows:

```
print(recs)
print(recs.dtypes)
```

The output should look something like Figure 7.2. (Note that the row contents have been split in the display since the size of the table is greater than the size of the terminal window. Don't worry - this is for display purposes only and the contents are indeed stored in a single row within the data frame).

Figure 7.2: Displaying a frame's dtypes

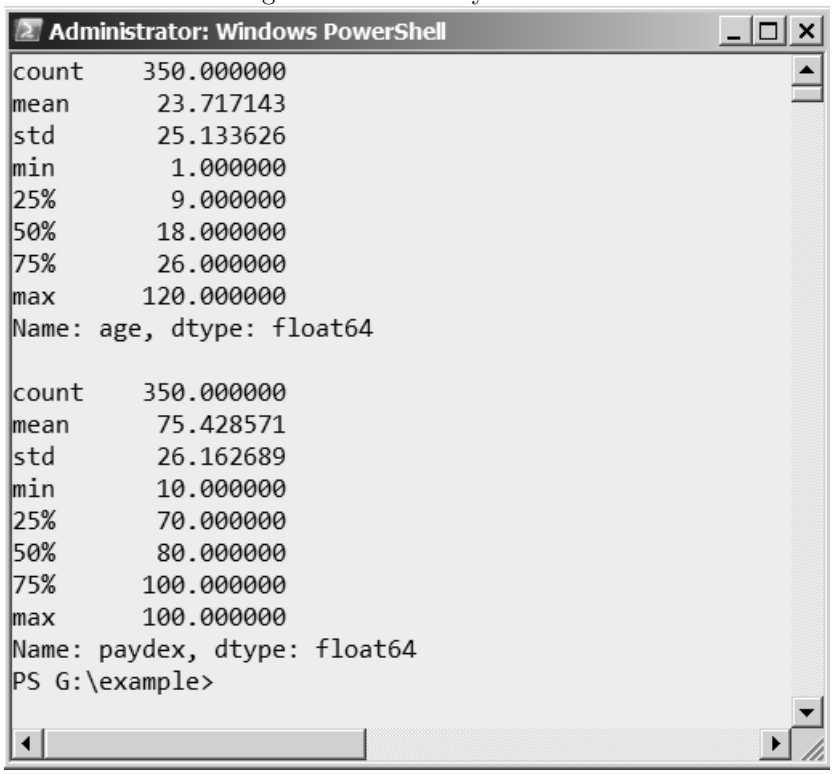


Since we are attempting to learn whether a customer's credit rating is associated with payment timeliness, our initial variables of interest for this analysis are `age` and `score`. Using the `descibe` function that we learned about earlier, let's go ahead and print some summary statistics for these variables using the `describe()` function we learned about in the previous chapter:

```
print(recs["score"].describe(),"\n")
print(recs["age"].dt.days.describe())
```

Since `age` has a Timedelta data type, we use the the `dt.days` attribute to return the difference expressed in days. We have also included a newline character (`\n`) in the first `print` call simply to place a blank line between the statistics for each column. Your output should look similar to Figure 7.3. We can see that the average age of an invoice is 23.7 days, half are paid within 18 days, and three quarters are paid with 26 days. We can also see that the credit score (`score`) associated with a customer receivable is 75, although it appears that there is a significant portion of the customer base that has a lower credit score.

Figure 7.3: Summary statistics



7.2 Creating a histogram

A necessary assumption of simple linear regression is that for a given value of our independent variable (`score`), the dependent variable (`age`) is normally distributed. We can check the normality of our data by creating a simple histogram.

The module *matplotLib* and its sub-module *pyplot* is used to provide plotting

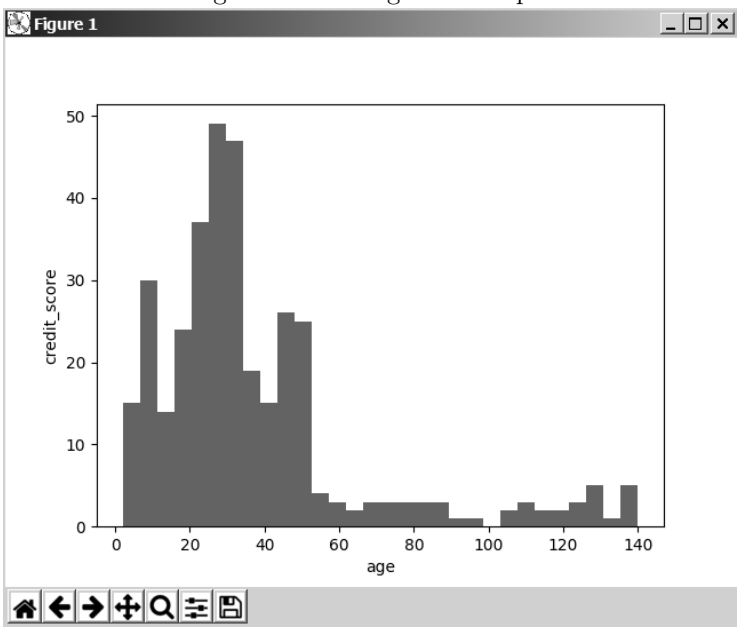
functionality for Pandas. If you're using Python from within a campus computer lab, it's likely matplotLib is already installed. If not, it can be installed by running the following commands from within your terminal:

```
pip install matplotlib
```

Note that we can use the library from within our programs by including the following command in our Python scripts.

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

Figure 7.4: Histogram example



With the library successfully installed, let's go ahead and create that histogram. We'll use pyplot's `hist()` function and pass the `age` column that we want to plot. This function also takes a parameter (`bins`) represents the interval used to display the output along the x-axis (we'll choose a bin size of 20). We will then assign some simple labels to the histogram axes before finally displaying the plot on the screen using the `show()` function.

```
plt.hist(x=recs["age"].dt.days, bins=20)
plt.xlabel("age")
plt.ylabel("frequency")
plt.show()
```

As we can see in Figure 3.4, the data appears to have a somewhat normal (bell-

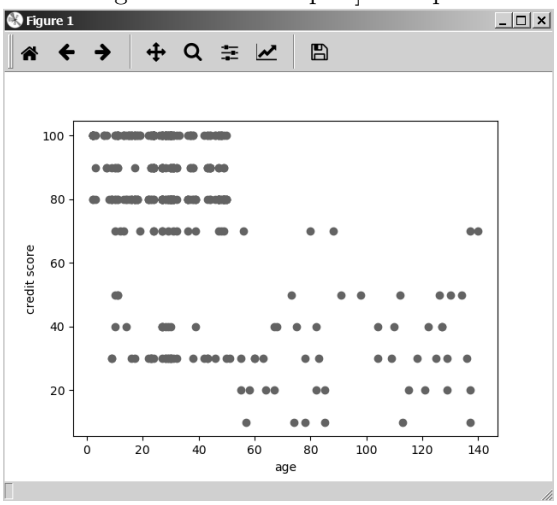
shaped) distribution, although it is right-skewed. We might want to adjust the data to account for this skewness (for example, we can use a log transformation) but since this is an introductory text, we'll accept the data as is.

7.3 Creating a scatterplot

Let's also visually examine the relationship between credit score and payment timeliness with a simple scatterplot using pyplot's `scatter()` function. We'll also add labels to the x and y axes in order to make the plot more interpretable. The output is presented in Figure 7.5.

```
plt.scatter(recs["age"].dt.days, recs["score"])
plt.xlabel("age")
plt.ylabel("credit score")
plt.show()
```

Figure 7.5: Scatterplot] example



As we can see from the histogram, the data seems to be clustered along the upper left and upper right corners of the chart, suggesting a possible negative relationship between the two variables (although it does not appear that this possible relationship is very strong).

The full Python code for setting up our data frame, calculating the summary statistics, and displaying the simple histogram are presented in Figure 7.6.

Figure 7.6: Data preparation and summary statistics

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

#import and set up data frame
recs = pd.read_csv("accounts_receivable.csv")
recs["inv_date"] = pd.to_datetime(recs["inv_date"])
recs["due_date"] = pd.to_datetime(recs["due_date"])
recs["paid_date"] = pd.to_datetime(recs["paid_date"])

#calculate age of receivable and late status
recs["age"] = recs["paid_date"] - recs["inv_date"]
recs["late"] = recs["paid_date"]>recs["due_date"]

#summary statistics
print(recs["score"].describe())
print(recs["age"].dt.days.describe(), "\n")

#display histogram
plt.hist(x=recs["age"].dt.days, bins=20)
plt.xlabel("age")
plt.ylabel("frequency")
plt.show()

#display scatterplot
plt.scatter(recs["age"].dt.days, recs["score"])
plt.xlabel("age")
plt.ylabel("credit score")
plt.show()
```

7.4 Installing scipy and statsmodels

To perform the statistical analysis discussed in the rest of this chapter, we'll be relying on the *scipy* and `statsmodels` modules. These module contains the probability distributions and statistical functions we need to complete our example. If you are using a data science oriented Python distribution (such as Anaconda), this package should already be installed for you. Otherwise, the package needed to install scipy and statsmodels on Windows machines must be downloaded and installed manually (scipy needs to be installed prior to statsmodels). The latest version of the package file currently be downloaded from:

`http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy`

Be sure to download the package file that matches your version of Python (e.g., 3.6). When downloaded, the file should be copied to the directory that Python is installed on your machine. The package can then be installed using the following command from within the terminal:

```
pip install "file name"
```

where file name is the name of the package file downloaded. You can verify that pip has installed correctly by running the command `pip install scipy` within the terminal. The terminal output, `requirement has already been satisfied`, indicates successful installation. We'll import the modules by inserting the following code at the top of our script:

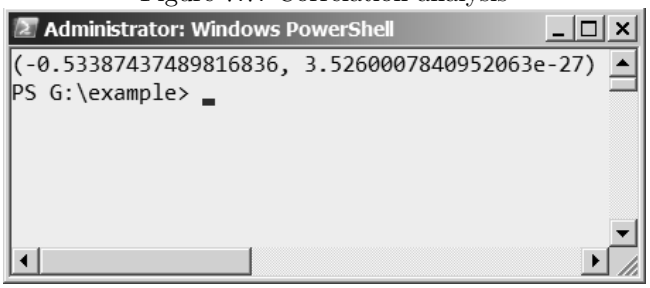
```
import scipy
from scipy.stats import pearsonr
import statsmodels.api as sm
```

7.5 Correlation

With our data frame fully set up, we can now perform our first test to see whether a customer's credit score is related to the timeliness of payment. We will calculate a Pearson correlation coefficient to measure the linear relationship between the two variables. This can be done using scipy's `pearsonr()` function which takes the two variables being examined as parameters and returns the correlation coefficient and p-value. We'll nest the `pearsonr()` function call in a `print()` call in order to print the analysis to the screen. The program output is presented in Figure 7.5.

```
print(pearsonr(recs["age"].dt.days, recs["score"]))
```

Figure 7.7: Correlation analysis



As indicated in the output, the relationship between credit score and payment timeliness appears to be negative. That is, the higher the customer's credit

score, the earlier it is likely to be paid. In addition, this relationship appears to be highly significant, with a p-value of less than .01 (note the use of scientific notation in the program output). As a result, we can conclude that there does appear to be a statistically significant relationship between the two variables, although it is perhaps not as strong as the company might hope.

7.6 Linear regression

We can use ordinary least squares to estimate a function which can be used to predict payment timeliness based upon a customer's credit rating. We'll express the function in the format :

$$age_i = \beta_0 + \beta_1 score_i + \epsilon_i$$

To make things a bit easier for us, we'll create two new variables, `x` and `y` which store the values of our independent and dependent variables respectively.

```
y = recs["age"].dt.days
x = recs["score"]
```

The statsmodel ols regression function does not automatically include the constant term, so we'll add that now:

```
x = sm.add_constant(x)
```

We'll then specify the model by passing the dependent and independent variables to the statsmodel's `ols()` function and save the return value in a new variable called `mod`. It is typically good practice to store statistical models in their own unique variables in order to allow for future access/modification.

```
mod = sm.OLS(y, x)
```

We'll then use statsmodel's `fit()` function to estimate the model and save the return value in a new variable called `results`.

```
results = mod.fit()
```

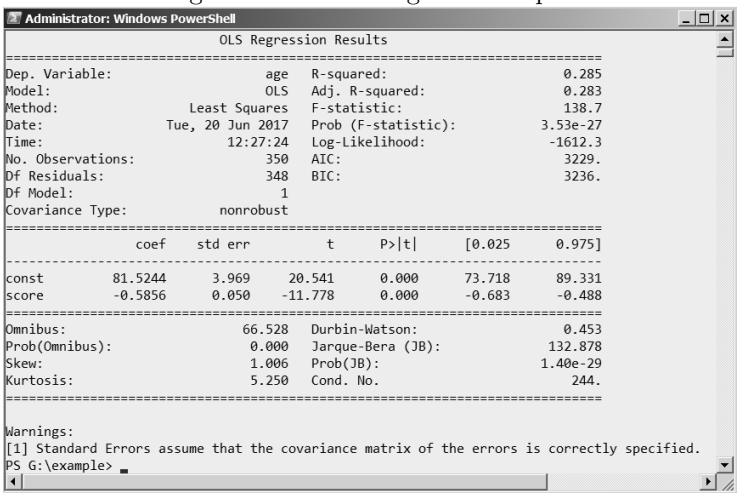
Finally, we'll print the estimation statistics using the object's `summary()` method.

```
print(results.summary())
```

If you've done everything correctly, your output should look similar to what is

presented in Figure 7.8.

Figure 7.8: Linear regression ouput



Based upon our review of this output, we can see that the overall model is statistically significant (F=138.7, p <.01). However, the model's R^2 (coefficient of determination) suggets suggests that customer credit score only explains approximately 28% of the variance in the dependent variable, receivable age at time of payment.

Finally, the coefficient estimate for the credit score variable indicates that a one point increase in customer credit score is associated with a .5856 reduction in the number of days before payment is made. We could use this output to develop the following simple forecasting model that can estimate payment timeliness based upon any value of customer credit rating:

$$age_i = 81.52 + (-.59)score_i$$

Be advised again, however, that the model's R^2 suggests this is a relatively poor model and that there other factors that influence payment timeliness that are not incorporated in the customer's credit score. Indeed, this may suggest that the company's credit model may need to be revisited.

7.7 Advanced: Logistic regression

We may also want to use linear modeling in circumstances where the dependent variable is binary. For example, perhaps we want to develop a model to

predict whether an invoice will be paid late (e.g., greater than 30 days) based upon the customers credit score. This type of analysis can be performed using statsmodel's `Logit()` function. This time, we'll pass the binary variable `Late` which we defined earlier in the chapter, as well as our credit score variable (`score`) to the function.

First, we'll create new values for `x` and `y`, again representing our independent and dependent variables, respectively. We'll than add the constant term to `x`.

```
y = recs["late"]
x = recs["score"]
x = sm.add_constant(x)
```

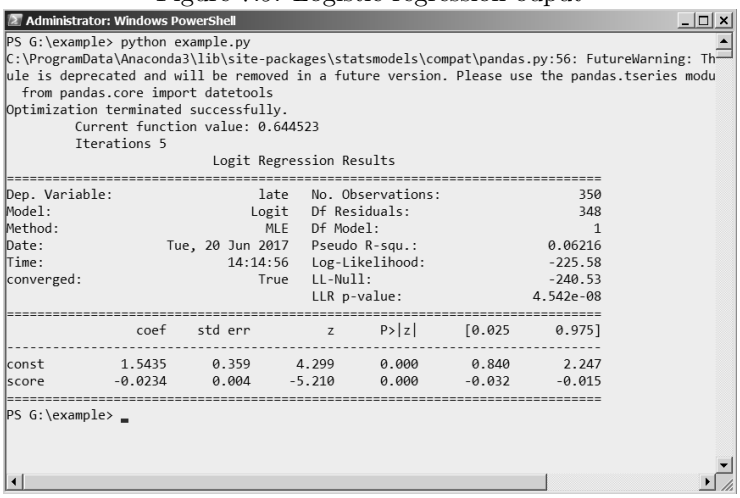
We can then specify the model as new variable, `mod`, by passing the `x` and `y` variables to the `logit()` function.

```
mod = sm.Logit(y, x)
```

Finally we will fit the model to a new variable, `results`, and display the estimation statistics. The related output is presented in Figure 7.9.

```
results = mod.fit()
print(results.summary())
```

Figure 7.9: Logistic regression ouput



As indicated in the output, the z-score for the `score` variables is statistically significant ($p<.01$), and negative. To improve the interpretability of the coefficients, we can display the odds ratio by taking the exponential of each of the coefficients. We can do this using numpy's `exp()` function. The following ex-

pressions will print a list of the odds ratios for each of the independent variables in the model:

```
odds = np.exp(results.params)
print(odds)
```

Recall that the odds ratios allow us to express how a one unit change in the independent variable effects the odds of a specific dependent variable outcome. For example, if the odds ratio for the `score` independent variable is 0.98, this means that a one point increase in customer credit score is associated with a 2% reduction in the odds of an invoice being paid late.

7.8 Advanced: Handling outliers

The statsmodel function `outlier_test()` is a useful little tool for detecting outliers in ols regression. Calling the function returns a dataframe that contains three columns: the studentized residuals, the unadjusted p-value, and the Bonferroni-corrected p-value (default). We can call the function and store return value in a new data frame called `test` as follows:

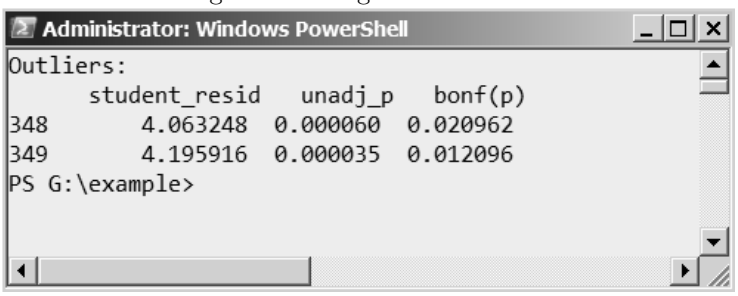
```
test = results.outlier_test()
```

We can then identify any observation that has a studentized residual that has a statistically significant p-value (in our case, this will be any p <0.05). We will store these observations in a new data frame called `outliers`:

```
outliers = test[test["bonf(p)"] < 0.05]
```

Printing the data frame will result in output similar to that included in Figure 7.10.

Figure 7.10: Significant outliers



As you can see, this analysis has identified two potential outliers, located at

row index 348 an 349. If we wanted, we could create a new data frame which excludes these two outliers using the
textttdrop() function. This function takes a dataframe and a list of row index values and returns the frame contents with the specified rows removed. In this example, it would be used as follows.

```
newRecs = test.drop(test.index[[348,349]])
```

7.9 Summary

We began the chapter by discussing how to display basic summary statistics for a data set, including how to prepare histograms and scatterplots. We then discussed how to perform simple a correlation analysis using scipy and statsmodel, before reviewing linear and logistic regression. The chapter concluding with a brief discussion of outlier identification using studentized residuals.

7.10 Exercises

1. **Portfolio project**: In Chapter 6, Exercise 6, you wrote a simple program that used cross-tabulation to determine if alcohol consumption patterns are related to local climate. Keeping both alcohol consumption and average temperature as continuous (i.e., non-binary) variables, improve the sophistication of your analysis by performing the following steps:

    - Prepare summary statistics for both variables
    - Create a histogram for both variables
    - Prepare a scatterplot, placing alcohol consumption on the y axis and average temperature on the x axis. Based upon your visual inspection, does a relationship appear to exist between the two variables?
    - Conduct a simple Pearson correlation analysis between the two variables. What is the nature of the correlation (if any)? Is it statistically significant?
    - Perform a linear regression analysis using alcohol consumption as the dependent variable (y) and average temperature (x) as the independent variable. Is the coefficient on the independent variable statistically significant? If so, how should it be interpreted?

Chapter 8

Plotting and visualization

Plotting and visualizing data is a very important part of any data analysis project. In is regularly the case that the findings of a project can be communicated more efficiently and persuasively in visual, as opposed to tabular, form. In this chapter we'll review some of the basic plotting and visualization functions available in Python's pyplot library.

8.1 Preparing our data

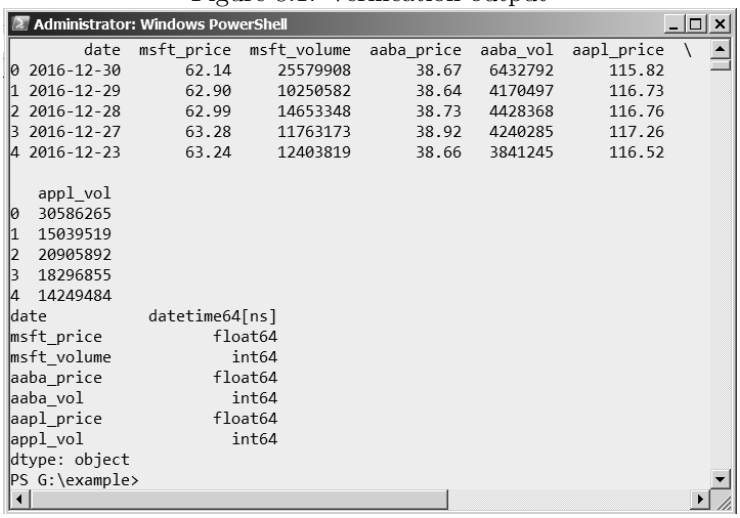
We'll be working with a simple data set that contains the 2016 stock price and trading volume activity for a group of three major tech companies. This is data similar to what a CPA might review as part of an analytical procedures auditing task.

The data can be scraped from `http://www.steveperreault.com/textbook/stockanalysis.htm` to a CSV file following the steps outlined in Chapter 5. The data should then be imported into a new Pandas dataframe as discussed in Chapter 6 (use the default sequential numerical row index). Recall that you will need to change the data type of the `date` column to `datetime` following the steps discussed in Chapter 7. Verify that you've performed the import correctly by printing the first five rows of the frame as well as the columns' data types using the following commands:

```
print(table.head());
print(table.dtypes)
```

Your output should look similar to Figure 8.1.

Figure 8.1: Verification output



8.2 Line plots

Let's begin our exploration of plotting in Python by creating a simple line chart displays the stock price for the three companies included in the data frame throughout 2016. Since we only need the stock price data to prepare this chart, let's create a new data frame called `price` which contains only that data.

```
price = table.drop(table.columns[[2, 4, 6]], axis = 1)
```

Note that we're using the `drop()` function here in a slightly different way than we're used to, passing the column numbers instead of the column names. This command tells Pandas to drop the 3rd, 5th, and 6th, columns from the data frame.[1] The `axis = 1` parameter simply tells the function that we want to delete columns instead of rows (we would pass `axis = 0` to refer to rows). We now have a nice new data frame that contains a `date` column and five additional columns containing daily stock price for the five companies of interest.

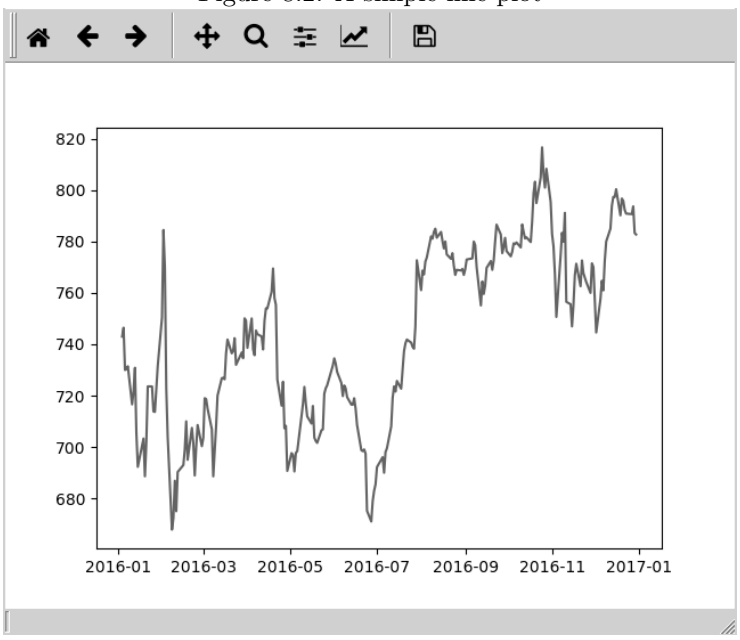
A basic line plot can be created by passing the y and x variables, respectively, to pyplot's `plot()` function. Let's first plot the change in Alphabet Inc.'s stock price over time. It makes the most sense to plot the stock price on the y axis and the date on the x axis, so we'll call the function as follows:

[1]Remember that dataframe columns are zero-indexed, so the first column is actually referred to as column zero.

```
plt.plot(price.date.values,price.goog_price)
```

If we show the function (using `plt.show()`), we should see something similar to Figure 8.2.

Figure 8.2: A simple line plot



Let's now modify our code so that the stock price history for each of the five companies in the dataset is contained on the same plot. We could do this by making a series of three sequential calls to `plot()` as follows:

```
plt.plot(price.date.values,price["aapl_price"].values)
plt.plot(price.date.values,price["msft_price"].values)
plt.plot(price.date.values,price["aaba_price"].values)
```

However, this is not a terribly efficient way to write code and would become extremely cumbersome if we had a lareg numebr of columns we wanted to plot. Instead, let's use a `for` loop to iterate across the three price columns and add each to the plot. Note that we'll begin our iteration at the column index 1 (since index 0 contains the `date` column which is already passed as the x parameter to `plot()`.

```
for column in price.columns[1:]:

    plt.plot(price.date.values,price[column].values)
```

There are a couple of additional steps we can take to make this plot look a bit nicer and more functional. First, we need to add a legend so that it will be clear which company each line is referring to. We can do this by calling the `legend()` and passing the column names for each of the three columns containing cmopany stock prices. By default, matplotlib will attempt to place the legend where it believes it will best fit, however there are numerous other placement options which are described in the matplotlib documentation.

```
plt.legend(price.columns[1:])
```

Finally, we'll label the y axis and add a title to the plot by calling the following functions:

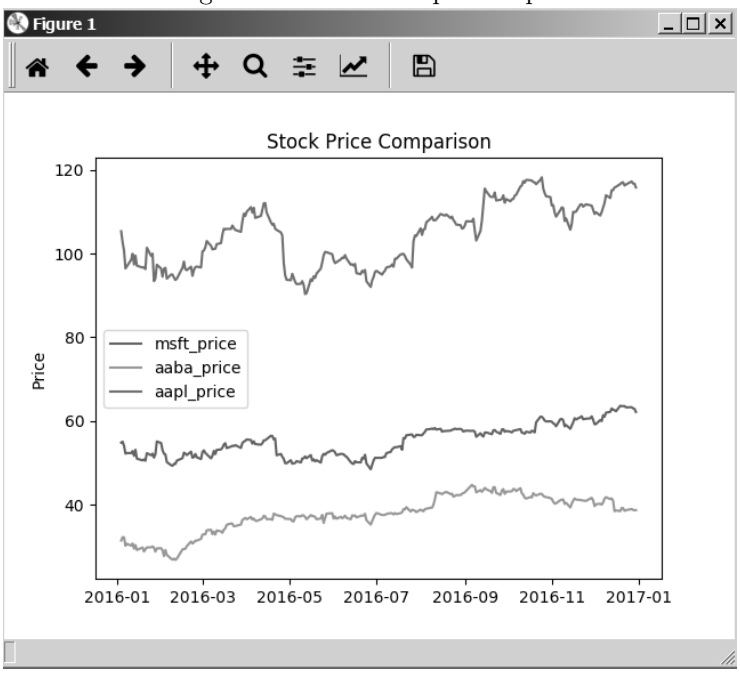
```
plt.ylabel("Price")
plt.title("Stock Price Comparison")
```

we have now created a basic line chart using matplotlib. The full source code that we created for preparing the chart as well as the related output are presented in Figures 8.3 and 8.4, respectively.

Figure 8.3: Basic line plot source code

```
1   import numpy as np
2   import matplotlib, matplotlib.pyplot as plt
3   import pandas as pd
4
5   #import and set up data frame
6   table = pd.read_csv("stockprice.csv")
7   table["date"] = pd.to_datetime(table["date"])
8
9   #create new data frame with only date and price information
10  price = table.drop(table.columns[[2, 4, 6]], axis = 1)
11
12  #plot each line
13  for column in price.columns[1:]:
14  plt.plot(price.date.values,price[column].values)
15
16  #add legend and formatting
17  plt.legend(price.columns[1:])
18  plt.ylabel("Price")
19  plt.title("Stock Price Comparison")
20
21  plt.show()
```

Figure 8.4: Basic line plot output



8.3 Bar plots

We can also use matplotlib to create bar plots. In this example, we'll create a simple figure that plots the dollar value of each stock's change in price year over year. We'll begin by using the same `price` data frame that we used in the line plot example.

To do this, we'll have to calculate the change of price for each stock and we'll store these values in a new list called `changes` as follows:

```
changes.append(price.loc[0,"msft_price"] - price.loc[251,
"msft_price"])
```

This commands uses Panda's `loc()` method to find values at the row and column index specified. Note that the ordering of the `price` data frame is such that the stock price at the end of the year is located at row index 0, while the stock price at the end of the year is located at row index 251.[2] As a result, this expression subtracts the stock price's beginning of the year price from the end of the year price and appends the result to the `changes` list.

[2]This can be verified by calling `print(price.head))` and `print(price.tails())`.

We can the call the matpolotlib function bar() to create the bar chart. The chart taxes the x coordinates of the bars as well as the height of the bars as default parameters. Because we want the three bars to be evenly spaced, we'll pass the list `[1,2,3]` as the x coordinates (any evenly spaced list of three integers would be also work). We'll also pass the `changes` list we just created to provide the height for each of the bars.

```
plt.bar(y_pos, changes)
```

We want to update the x-axis ticks to plot the name of the stock underneath each bar. To do this need to create a list that contains the names of each of the three stocks (these should be presented in the same order as the contents of the `changes` list). The x-axis ticks can be updated using the `xticks()` function, passing both the x coordinates of the labels and the label text. Finally, we'll give the y-axis a label and title the plot.

```
stocks = ["msft", "aaba", "aapl"]
plt.xticks([1,2,3], stocks)
plt.ylabel("Price increase ($)")
plt.title("2016 Stock Price Increases")
```

Calling `plt.show()` should display a plot similar to that in Figure 8.5 The full source code for this plot is presented in Figure 8.6.

Figure 8.5: Basic bar plot output

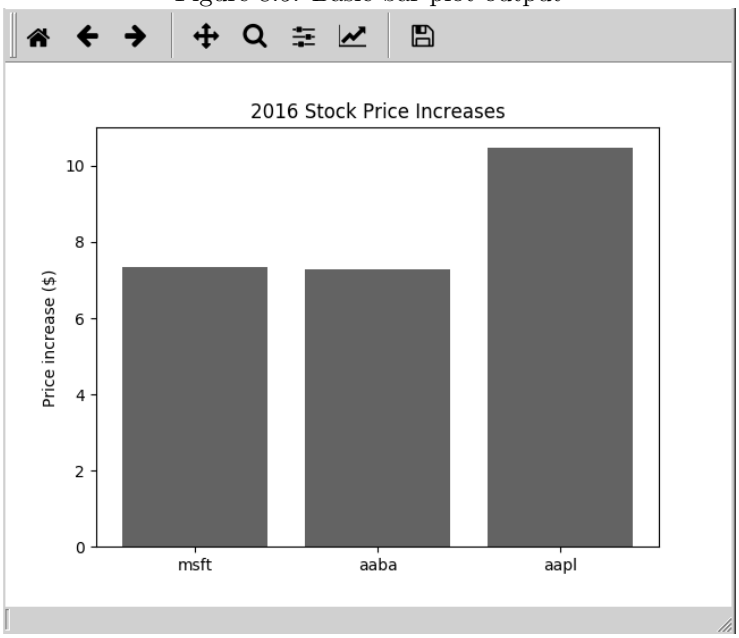


Figure 8.6: Basic bar plot source code

```
1   import numpy as np
2   import matplotlib, matplotlib.pyplot as plt
3   import pandas as pd
4
5   #import and set up data frame
6   table = pd.read_csv("stockprice.csv")
7   table["date"] = pd.to_datetime(table["date"])
8
9   #create new data frame with only date and price information
10  price = table.drop(table.columns[[2, 4, 6]], axis = 1)
11
12  #create new data frame with YOY stock price changes
13  changes = [];
14  changes.append(price.loc[0,"msft_price"] - price.loc[251,
        "msft_price"])
15  changes.append(price.loc[0,"aaba_price"] - price.loc[251,
        "aaba_price"])
16  changes.append(price.loc[0,"aapl_price"] - price.loc[251,
        "aapl_price"])
17
18  #create bar chart by passing x index and bar heights
19  plt.bar([1,2,3], changes)
20
21  #add labels and title
22  stocks = ["msft", "aaba", "aapl"]
23  plt.xticks([1,2,3], stocks)
24  plt.ylabel("Price increase ($)")
25  plt.title("2016 Stock Price Increases")
26
27  plt.show()
```

8.4 Pie chart

Pie charts are useful plots to use when trying to compare parts of a whole. In this example, we'll use a pie chart to plot proportion of `msft`, `aaba`, and `aapl` held within a portfolio. To begin, we'll create simple lists that hold the names of the stock we are plotting as well as some hypothetical holding percentages:

```
prop = [25, 35, 40]
stocks = ["msft", "aaba", "aapl"]
```

A very basic pie chart can be created by passing the proportions to the `pie()` function. However, we'll pass some additional parameters to the function in

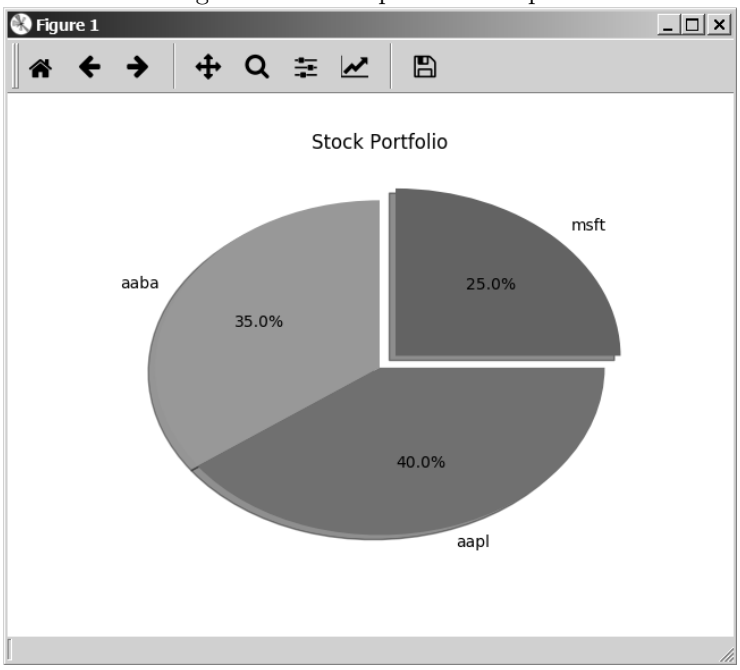
order to make our plot more useful and attractive. First, we'll add labels to each slice based upon the values included in the `stocks` list. We'll then pass the `explode` parameter which will be used to highlight the first slice of the pie via offset (this list can be modified to offset other slices as well). We'll also add a percent sign and round our percents to one decimal place using `autopct` before finally adding shadows to the pie by setting the value of the `shadows` property to `True`. Finally, we'll add a title.

```
plt.pie(prop, labels=stocks, explode = [.1, 0, 0],autopct="%1.1f%%",
shadow=True)

plt.title("Stock Portfolio")
```

Our output for this example, as well as the full source code, are presented in Figures 8.7 and 8.8.

Figure 8.7: Basic pie chart output



8.5 Exploring other plot types

The matplotlib modules offers users access to an enormous amount of different plot types that we have not covered here. These include box plots, scatter plots,

Figure 8.8: Basic pie chart source code

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

#create pie chart data
prop = [25, 35, 40]
stocks = ["msft", "aaba", "aapl"]

#create chart and add title
plt.pie(prop, explode = [.1,0,0], autopct="%1.1f%%", labels=stocks,
    shadow=True)
plt.title("Stock Portfolio")

#show chart
plt.show()
```

step plots, counter plots and many, many more. I encourage you to visit the matplotlib gallery to learn more about how to use these different plots in your projects.[3]

8.6 Summary

This chapter discussed the use of the matplotlib and plotly libraries to create simple plots and visualizations, including line plots, bar plots and pie charts. Basic methods for annotating and customizing these plots was also discussed.

8.7 Exercises

1. **Portfolio project**: Employing the data from Chapter 7 Exercise 1, plot the per capita alcohol consumption for the G7 countries (Canada, France, Germany, Italy, Japan, United States, and United Kingdom) using a bar chart. Stylize the bar chart so it looks visually attractive and is informative.

2. **Portfolio project**: Obtain the 2017 daily changes in stock price for Amazon.com, Ind (symbol: amzn) and Alphabet, Inc. (symbol: goog) from a online stock quoting service such as Google Finance. Plot the daily

[3] https://matplotlib.org/gallery.html

changes using a line plot. Stylize the line plot so it looks visually attractive and is informative.