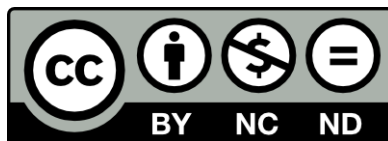


# **Introduction to Accounting Data Analytics Using Python**

Stephen Perreault, PhD, CPA

First Edition

© 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.

First edition: August 2017

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is data analytics? . . . . .	1
1.2	Sources of data . . . . .	1
1.3	About this book . . . . .	1
1.4	Intended audience . . . . .	1
1.5	Software used . . . . .	1
<b>2</b>	<b>Version control</b>	<b>3</b>
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	Discussion Questions . . . . .	20
2.14	Exercises . . . . .	21
<b>3</b>	<b>Basic Python concepts</b>	<b>23</b>
3.1	Why Python? . . . . .	23
3.2	Setting up a development environment . . . . .	24
3.3	Your first program: “Hello World!” . . . . .	25
3.4	An example development workflow . . . . .	28
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	Discussion questions . . . . .	42
3.19	Exercises . . . . .	43
<b>4</b>	<b>Intermediate Python concepts</b>	<b>47</b>
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 . . . . .	62
4.10	Discussion questions . . . . .	62
4.11	Exercises . . . . .	63
<b>5</b>	<b>Data scraping</b>	<b>67</b>
5.1	Introducing Beautiful Soup . . . . .	67

*To Kathleen, Lyla, Willy, and Matilda.*



# Chapter 1

## Introduction

**1.1 What is data analytics?**

**1.2 Sources of data**

**1.3 About this book**

**1.4 Intended audience**

**1.5 Software used**

Assumes you are using a Windows-based PC. Mac versions in the future.





## 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.<sup>1</sup> 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.

---

<sup>1</sup>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.

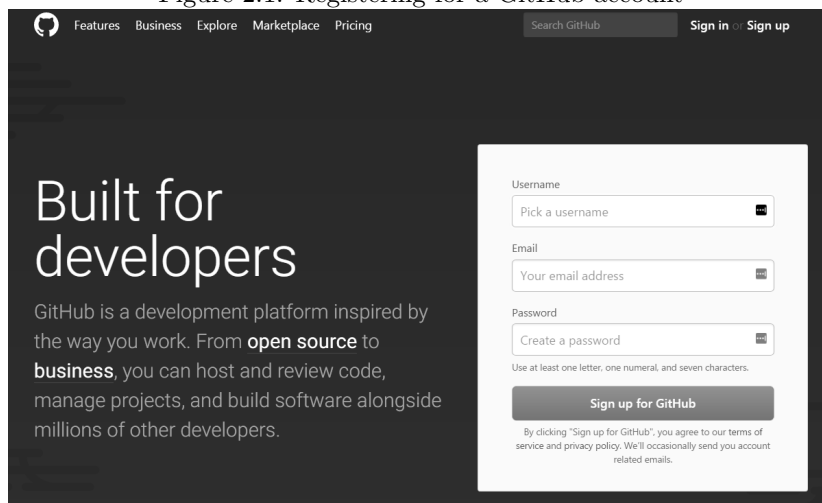
Unfortunately, Git has an reputation as being difficult for beginners to use

(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!<sup>2</sup> 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



The screenshot shows the GitHub website's registration page. The header includes the GitHub logo, navigation links (Features, Business, Explore, Marketplace, Pricing), a search bar, and 'Sign in' and 'Sign up' links. The main content area on the left says 'Built for developers' and describes GitHub as a development platform. On the right, a white registration form is displayed with the following fields: 'Username' with a placeholder 'Pick a username', 'Email' with a placeholder 'Your email address', and 'Password' with a placeholder 'Create a password'. Below the password field is a note: 'Use at least one letter, one numeral, and seven characters.' A 'Sign up for GitHub' button is at the bottom of the form. Below the button is a small disclaimer: 'By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We'll occasionally send you account related emails.'

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

---

<sup>2</sup>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 are unnecessary for the purposes of this book.

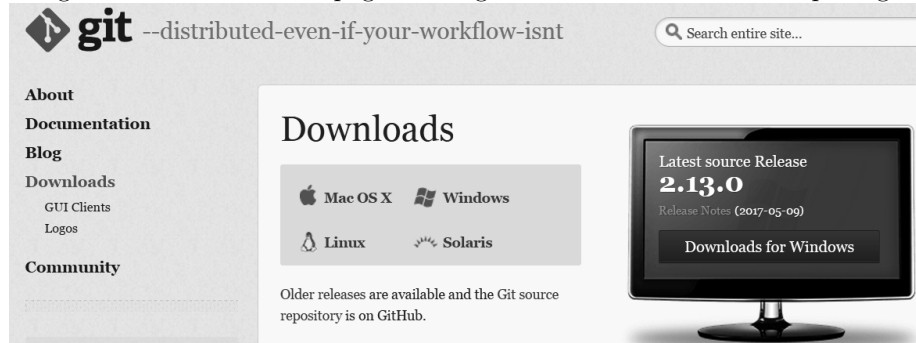
I encourage you to personalize your GitHub profile by uploading a recent picture of yourself. You may also wish to include other relevant background information or links to a webpage (if you have one). However, remember that the information which you include in your profile is publicly viewable - only 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.<sup>3</sup> 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

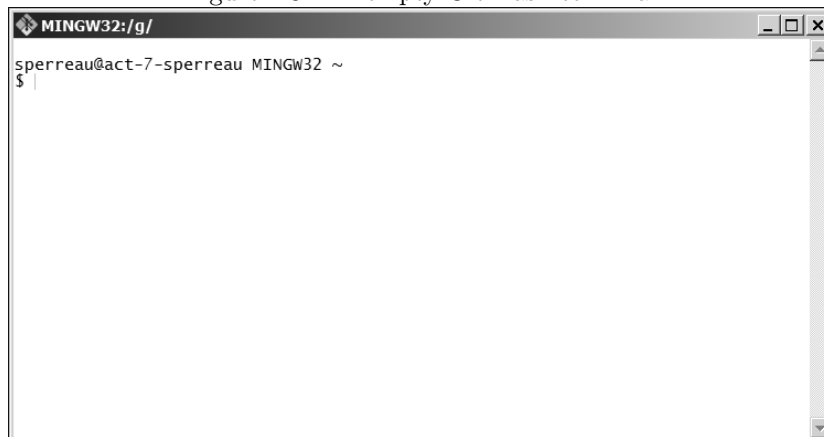


Launching "Git Bash" for the first time will display an empty terminal window

<sup>3</sup>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

The screenshot shows the GitHub 'Create new repository' page. At the top, the 'Owner' is 'steveperreault' and the 'Repository name' is 'my\_first\_repository'. Below this, a hint says 'Great repository names are short and memorable. Need inspiration? How about turbo-octo-funicular.' The 'Description (optional)' field contains 'My first ever repository. Very exciting!'. There are two radio buttons for visibility: 'Public' (selected) and 'Private'. Below these are checkboxes for 'Initialize this repository with a README' (unchecked) and 'Add .gitignore: None' and 'Add a license: None'. At the bottom is a 'Create repository' button.

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.<sup>4</sup>

<sup>4</sup>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:

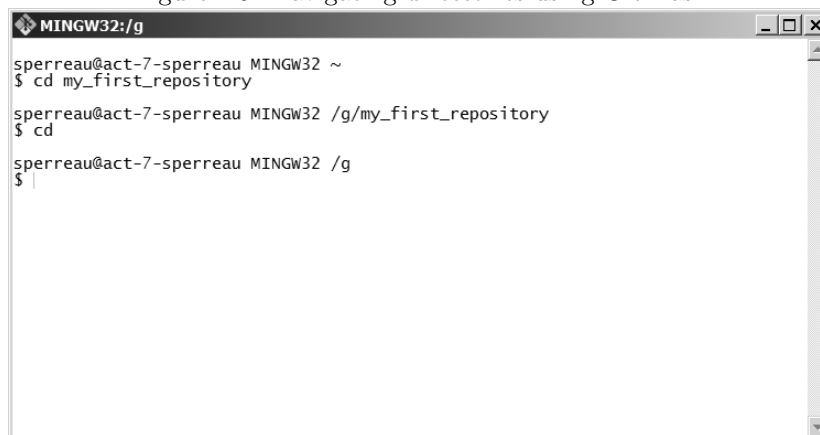
```
cd my_first_repository
```

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:<sup>5</sup>

```
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!

---

<sup>5</sup>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 it is prefixed by the term `git`.

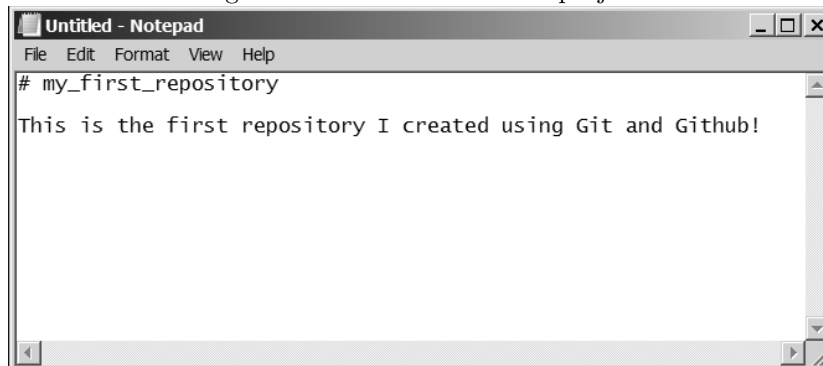
## 2.6 Adding files to your repository

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.<sup>6</sup>

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

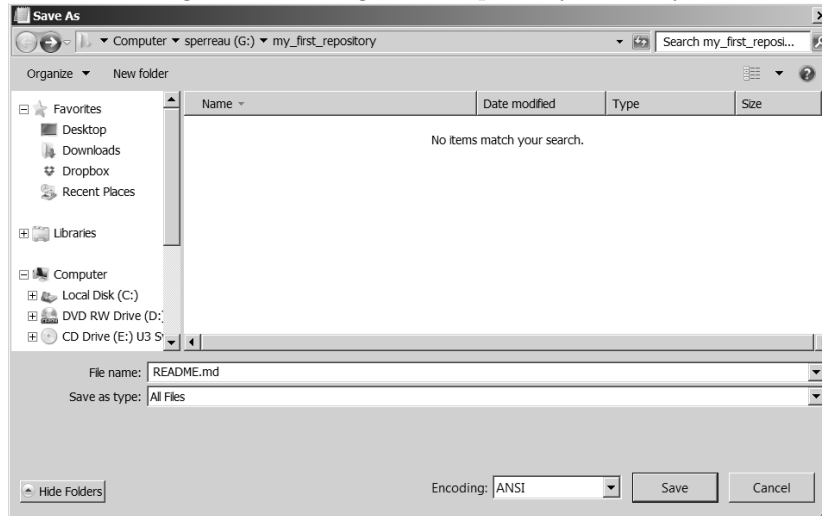


When your text file looks sufficient, it's time to save it to the directory that we created our repository in. In Notepad, click “Save as” from the “File” dropdown menu. Then navigate to the repository directory. Type `README.md` as the “File Name” and click the “Save” button. See Figure 2.7.

<sup>6</sup>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 "committing 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

A screenshot of a terminal window titled "MINGW32:/g/my\_first\_repository". The terminal shows the following commands and output:

```
sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$ git add README.md

sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md

sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$
```

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 committing 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 committed.

## 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:

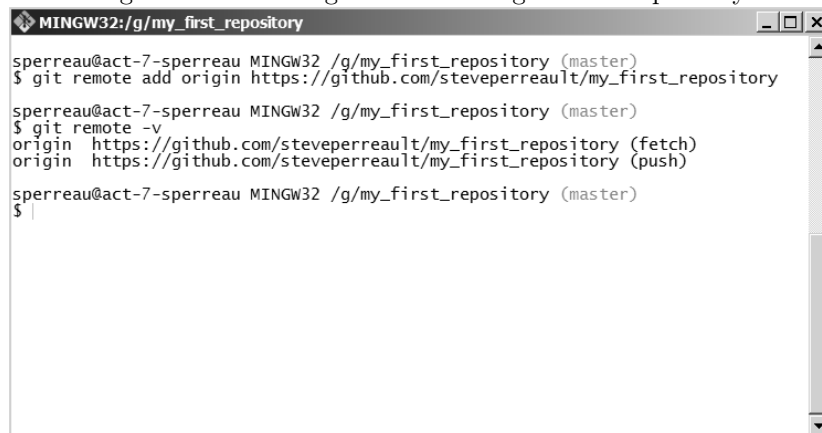
```
git remote add origin https://github.com.name/repo_name
```

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 by 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

A screenshot of a terminal window titled 'MINGW32: /g/my\_first\_repository'. The terminal shows the following commands and output:

```
sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$ git remote add origin https://github.com/steveperreault/my_first_repository

sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$ git remote -v
origin https://github.com/steveperreault/my_first_repository (fetch)
origin https://github.com/steveperreault/my_first_repository (push)

sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$
```

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.

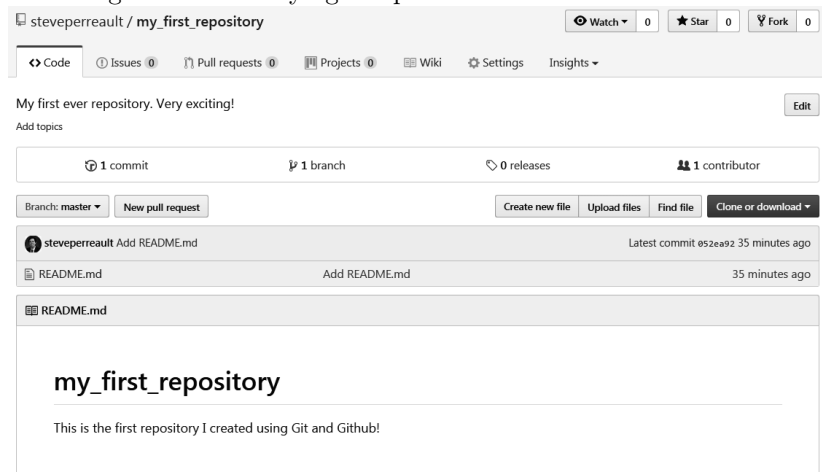
Now that we've verified that our connections are configured appropriately, we can actually push our changes up to the GitHub remote. The following command tells GitHub to push the master branch of the repo (the only branch we

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`.

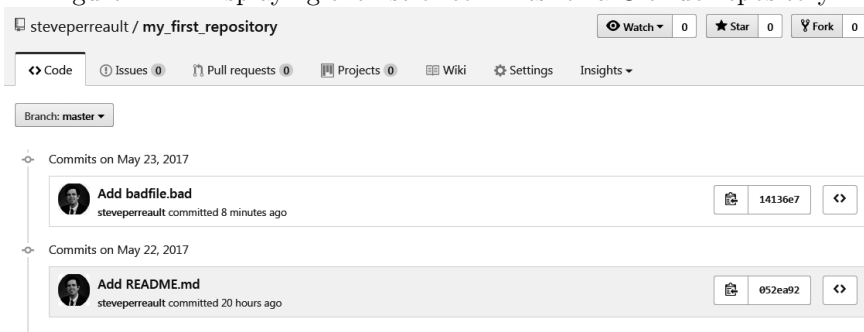
Now, using the methods discussed in the previous two sections, perform the following steps:

- Add `badfile.bad` to the local repository
- Check the local repository status, making sure that `badfile.bad` is marked as an addition to be committed
- 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 visible 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 commit 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`.<sup>7</sup> 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



```
MINGW32:/g/my_first_repository
sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$ git log
commit 14136e7fd52679f359754ec8c5a67d781c46ce85 (HEAD -> master, origin/master)
Author: Steve <sperreau@providence.edu>
Date: Tue May 23 11:15:09 2017 -0400

    Add badfile.bad

commit 052ea922acd3ffd9087a3230e8bf1d23e73d6336
Author: Steve <sperreau@providence.edu>
Date: Mon May 22 15:40:28 2017 -0400

    Add README.md

sperreau@act-7-sperreau MINGW32 /g/my_first_repository (master)
$
```

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).<sup>8</sup> 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!

---

<sup>7</sup>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.

<sup>8</sup>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 to 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.<sup>9</sup> 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).

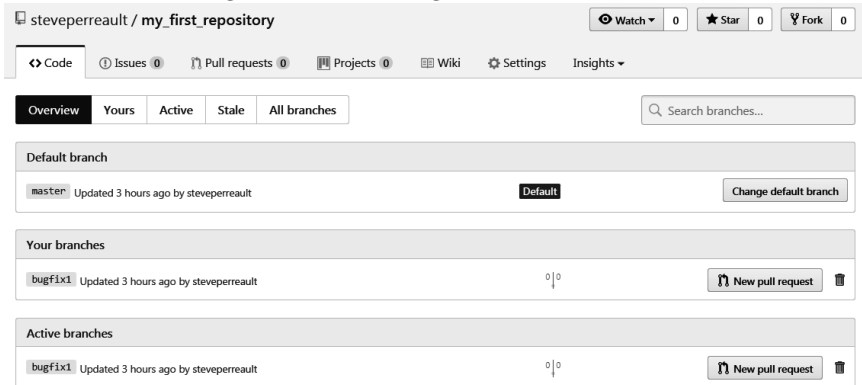
We can now switch back and forth between the two branches using `git checkout` from within the terminal. We can switch between the branches in the remote repository hosted on GitHub as well. Note, however, that since our repository now contains two branches, we will need to be careful to ensure that 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.

---

<sup>9</sup>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 terminal), we will find 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 Discussion Questions

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?

## 2.14 Exercises

1. 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.

2. 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!)

3. 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<sup>1</sup> 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:

---

<sup>1</sup>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 Python interpreter. The interpreter will take the source code we write and carry out the related instructions.<sup>2</sup>
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<sup>®</sup>. 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 Python interpreter (Windows PowerShell comes pre-installed on Windows version 7 and above).

---

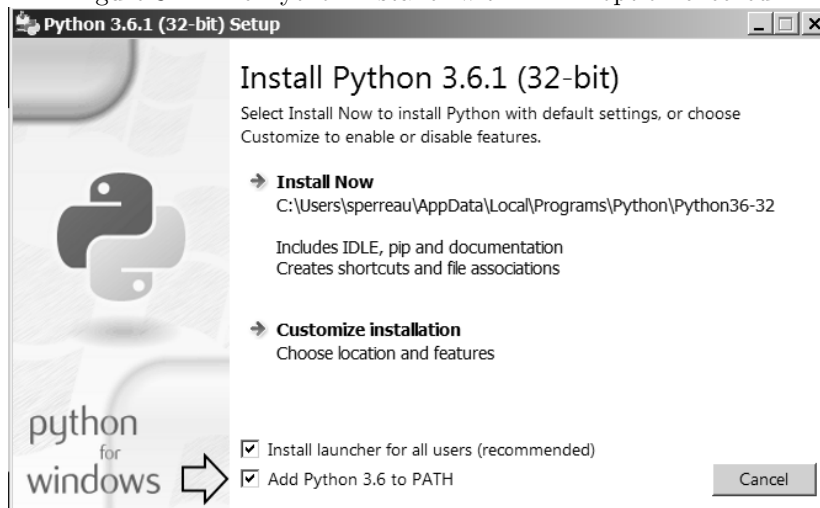
<sup>2</sup>We will be using the latest version of the Python 3.X interpreter which, at the time of this book's writing, is version 3.6.1. Note that the interpreter underwent a fairly controversial upgrade in 2008 which resulted in the version numbering switching from 2.7 to 3.X. The Python development 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 Python interpreter can be downloaded from <https://www.python.org/downloads>. Before downloading, make sure that you don't already have 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 Python interpreter then Python is already installed.

The 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. The full installation will take several minutes to complete.

To verify that 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



### 3.3 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”<sup>3</sup> using the following command:

```
md 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.

---

<sup>3</sup>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"`).



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).”<sup>4</sup> 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 extension `*.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**.

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).

---

<sup>4</sup>If you correctly set the editor syntax formatting to the Python language, this file type should already be selected.

Figure 3.5: Contents of the `hello_world` directory



```
Administrator: Windows PowerShell
PS G:\hello_world> ls

Directory: G:\hello_world

Mode                LastWriteTime         Length Name
----                -
-a---             5/24/2017   4:16 PM           76 hello_world.py

PS G:\hello_world> █
```

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



```
Administrator: Windows PowerShell
PS G:\hello_world> python hello_world.py
Hello world!
I did it! I wrote my first Python program!
PS G:\hello_world> █
```

### 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

---

```
1  print(2+4)
2  # prints the number 6
3  print(2-4)
4  # prints the number -2
5  print (2*4)
6  # prints the number 8
7  print (2/4)
8  # prints the number 0.5
9  print 5 % 2
10 # 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 reserved for identifying comments. Comments can be used to

Figure 3.8: Output for our simple arithmetic program



```
Administrator: Windows PowerShell
PS G:\hello_world> python hello_world.py
6
-2
8
0.5
PS G:\hello_world>
```

explain the purpose of a section of code in plain English or to temporarily 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`.

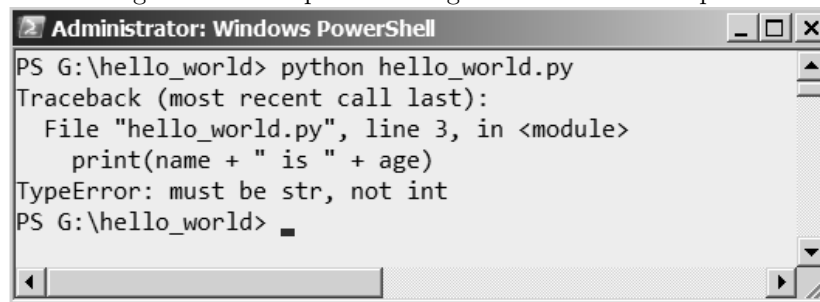
We can also concatenate strings and numeric values together but this does require one additional step. To demonstrate, what do you think the output of the program listed in Figure 3.12 would look like?

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

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command prompt shows the execution of a Python script: `PS G:\hello_world> python hello_world.py`. The output is a traceback indicating a `TypeError: must be str, not int` at line 3 of `hello_world.py`. The error message is: `Traceback (most recent call last):  
 File "hello_world.py", line 3, in <module>  
 print(name + " is " + age)  
TypeError: must be str, not int`. The prompt returns to `PS G:\hello_world>`.

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.<sup>5</sup> 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
```

---

<sup>5</sup>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 (foo[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.")
```

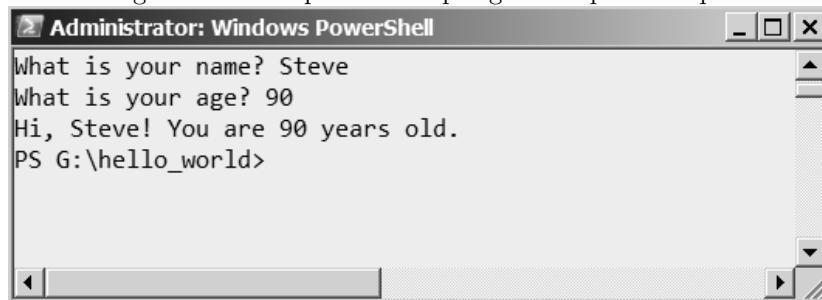
---

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
<code>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>&lt;=</code>	Less than or equal to
<code>&gt;=</code>	Greater than or equal to
<code>==</code>	Equal to
<code>!=</code>	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

---

```
1 print(5>6) # prints false
2 print(6<5) # prints true
3 print(6==5) # prints false
4 print(6!=5) # prints true
5 print("foo"=="foo") # prints true
6 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

---

```
1 age = input("What is your age? ")
2 if(age >= 18):
3     print("Congratulations!")
4     print("You are old enough to vote in the United States!")
5 else:
6     print("Sorry!")
7     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.

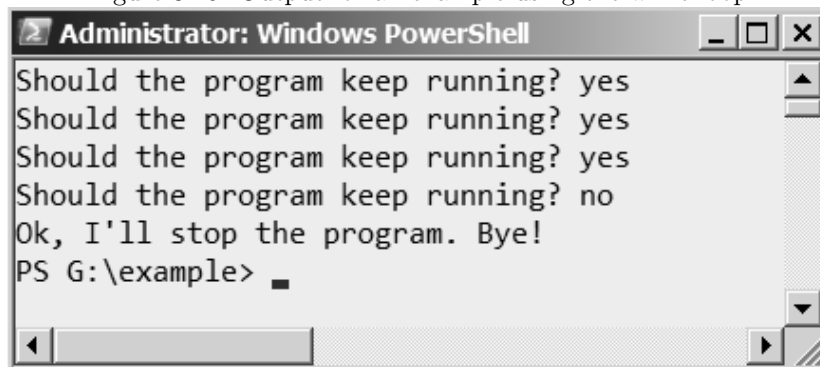
Figure 3.25: An example using the `while` loop.

---

```
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`.<sup>6</sup> 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`

---

<sup>6</sup>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 repeats. 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.

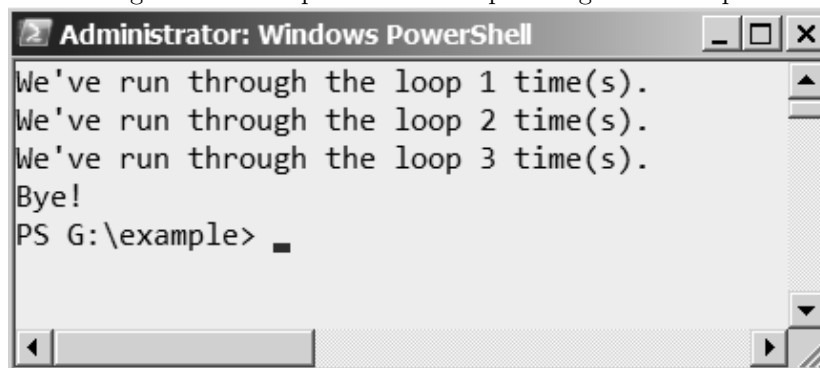
Figure 3.27: An example using the `for` loop.

---

```
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.<sup>7</sup>

## 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 Discussion questions

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`

---

<sup>7</sup>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!")
```
- 

### 3.19 Exercises

1. 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.

2. 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  $\text{weight}/\text{height}^2$ .

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

3. 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**.
4. Modify the program in Figure 3.27 so it prints the message "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()`.

5. 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
```

6. 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."
7. **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:

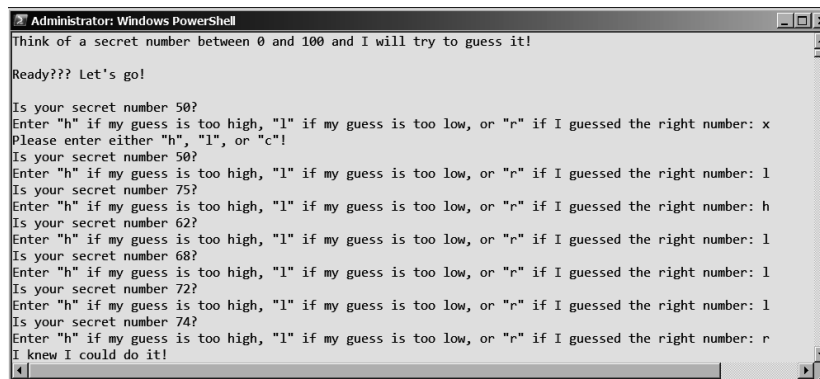
```
1: 1
2: 1
3: 2
4: 3...
```

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

8. **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:



```
Administrator: Windows PowerShell
Think of a secret number between 0 and 100 and I will try to guess it!

Ready??? Let's go!

Is your secret number 50?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: x
Please enter either "h", "l", or "c"!
Is your secret number 50?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: l
Is your secret number 75?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: h
Is your secret number 62?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: l
Is your secret number 68?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: l
Is your secret number 72?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: l
Is your secret number 74?
Enter "h" if my guess is too high, "l" if my guess is too low, or "r" if I guessed the right number: r
I knew I could do it!
```

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 initials:", end=" "), PrintInitials(firstName, lastName)
7  print("Tom Smith's initials:", end=" "), PrintInitials("Tom", "Smith")
```

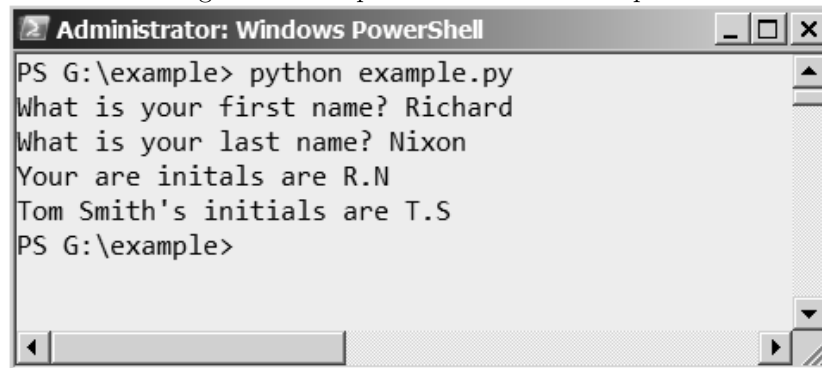
---

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 then tell the function to expect two arguments each time it is called, `firstName` and `lastName`.
- We end the function definition with a colon `:`.
- 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



```

Administrator: Windows PowerShell
PS G:\example> python example.py
What is your first name? Richard
What is your last name? Nixon
Your initials are R.N
Tom Smith's initials are T.S
PS G:\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")`.<sup>1</sup> 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

<sup>1</sup>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 initials 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 occurrence 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)

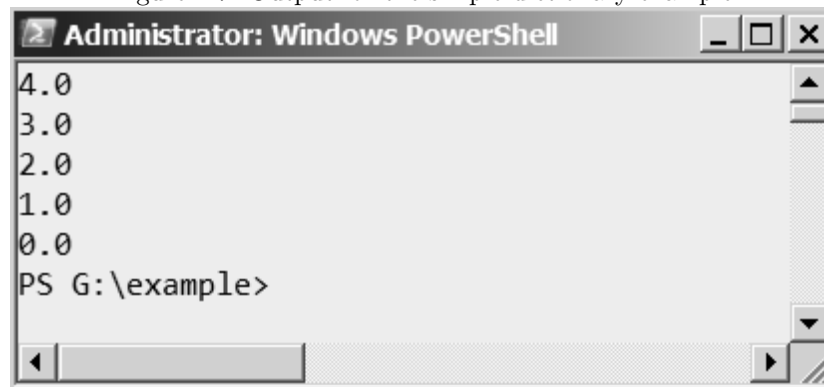
Figure 4.6: A simple dictionary

---

```
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 out 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

---

```
1     employees = {"Name": "Steve", "Job": "Accounting professor", "Age":  
                90}  
2  
3     for k, v in employees.items():  
4         print (k + ": " + str(v))
```

---

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

New key-value pairs can be added 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", "a")
```

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

create a new file called "myfile.txt" if the file does not already exist but append any modifications we make if an existing file does exist. 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.

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 prints 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://www.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.<sup>2</sup> 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

---

<sup>2</sup>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 data. 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 variable 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 unpaid) 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.<sup>3</sup>

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 a new instance of a class is created. Every class definition you create should contain a `__init__` method.

---

<sup>3</sup>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 parameter, `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 referred to as “members.”) The prefix `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.

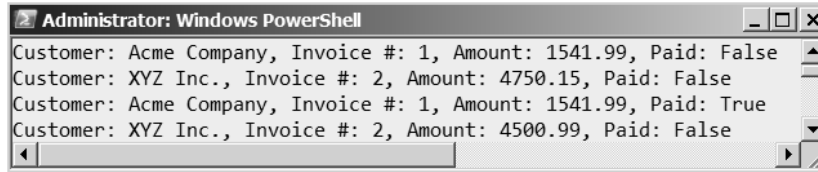
With the class definition complete, we can now create individual instances of the class. Let's review the expression included in line 24 of the program and repeated below:

```
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



```
Administrator: Windows PowerShell
Customer: Acme Company, Invoice #: 1, Amount: 1541.99, Paid: False
Customer: XYZ Inc., Invoice #: 2, Amount: 4750.15, Paid: False
Customer: Acme Company, Invoice #: 1, Amount: 1541.99, Paid: True
Customer: XYZ Inc., Invoice #: 2, Amount: 4500.99, Paid: False
```

## 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 invoice attributes.

We could start from scratch and create a new class to create accounts payable invoice objects. 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 the parent invoice class that relate to accounts payable and accounts receivable classes specifically. These derived classes would inherit the attributes from the parent class. The example in Figure 4.12 demonstrates how the parent class could be defined, while the derived classes are presented in Figure 4.13.

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."

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. Note that both derived classes share all of the same methods as the **Invoice** parent class, with the exception of the `__init__` which has been overridden in each of

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))
```

---

the derived classes. This is due to the fact that initializing instances of each of the derived classes requires the passing of 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`.

Here is some further code and the related output.

## 4.9 Summary

## 4.10 Discussion questions

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]`

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
```

---

- (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()`

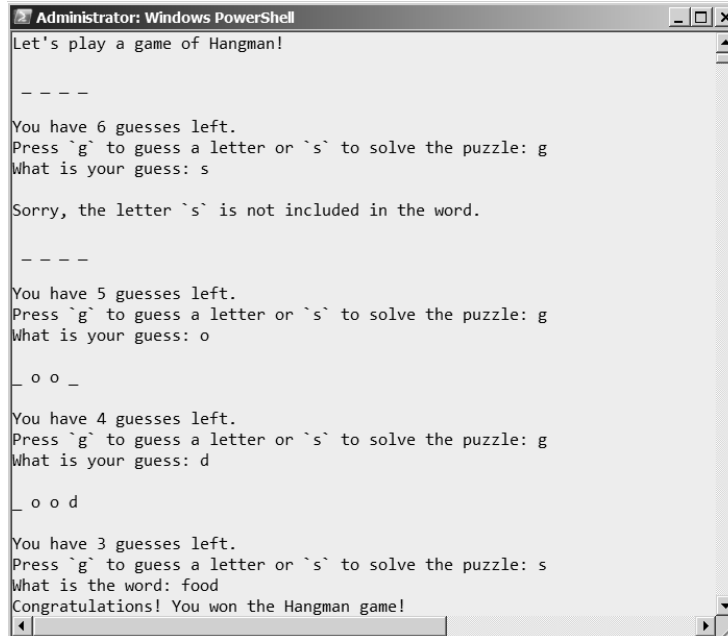
## 4.11 Exercises

1. 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.

2. 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 Fahrenheit is equivalent to approximately 24 degrees Celsius.
3. 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]`.
4. 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]`.
5. Create dictionary that contains a series of keys representing abbreviations of five US states (e.g., NY, MA, CA) whose related values 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.
6. **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.
7. **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 4.13.

Figure 4.15: Sample program output for exercise 7



```
Administrator: Windows PowerShell
Let's play a game of Hangman!

- - - -

You have 6 guesses left.
Press `g` to guess a letter or `s` to solve the puzzle: g
What is your guess: s

Sorry, the letter `s` is not included in the word.

- - - -

You have 5 guesses left.
Press `g` to guess a letter or `s` to solve the puzzle: g
What is your guess: o

_ o o _

You have 4 guesses left.
Press `g` to guess a letter or `s` to solve the puzzle: g
What is your guess: d

_ o o d

You have 3 guesses left.
Press `g` to guess a letter or `s` to solve the puzzle: s
What is the word: food
Congratulations! You won the Hangman game!
```

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 10 to the variable `number`.

8. **Portfolio project:** Modify the example in 4.11 so that it is interactive. 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).





## Chapter 5

# Data scraping

### 5.1 Introducing Beautiful Soup