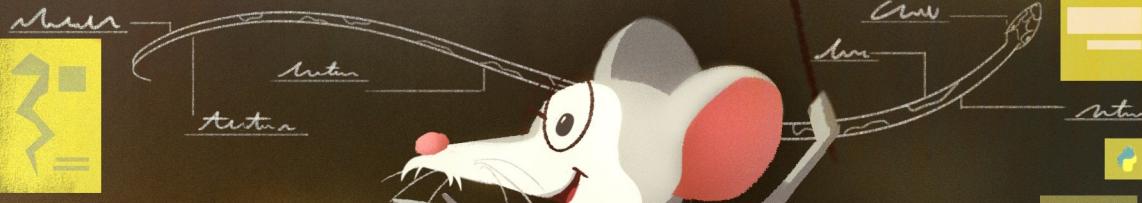


MICHAEL DRISCOLL

PYTHON 101

SECOND EDITION



Sold to

msaravanan@live.com



Python 101

2nd Edition

Michael Driscoll

This book is for sale at <http://leanpub.com/py101>

This version was published on 2020-08-29



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Michael Driscoll

Contents

About the Technical Reviewers	1
Ethan Furman	1
Martin Breuss	1
Acknowledgments	2
Introduction	3
Part I - The Basics	3
Part II - Intermediate Materials	3
Part III - Tutorials	4
Part IV - Python Packaging and Distribution	4
Target Audience	5
About the Author	5
Conventions	5
Requirements	6
Book Source Code	6
Reader Feedback	6
Errata	6
Part I - The Python Language	7
Chapter 1 - Installing Python	8
Installing on Windows	9
Installing on Mac	15
Installing on Linux	21
Android / iOS	22
Other Operating Systems	22
Other Python Variants	22
Wrapping Up	22
Chapter 2 - Python Editors	24
What About the REPL?	25
Getting Started with IDLE	28
Getting Started with PyCharm Community Edition	36
Getting Started with Wing Personal	40

CONTENTS

Getting Started with Visual Studio Code	42
Wrapping Up	45
Chapter 3 - Documenting Your Code	46
What are Comments?	46
Commenting Out	47
Multiline Comments	48
Learning About docstrings	48
Python's Style Guide: PEP8	49
Tools that can help	49
Wrapping Up	50
Review Questions	50
Chapter 4 - Working with Strings	51
Creating Strings	51
String Methods	52
String Formatting	54
Formatting Strings Using %s (printf-style)	55
Formatting Strings Using .format()	56
Formatting Strings with f-strings	59
String Concatenation	60
String Slicing	61
Wrapping Up	62
Review Questions	62
Chapter 5 - Numeric Types	64
Integers	64
Floats	65
Complex Numbers	66
Numeric Operations	66
Augmented Assignment	67
Wrapping Up	68
Review Questions	68
Chapter 6 - Learning About Lists	69
Creating Lists	69
List Methods	70
List Slicing	77
Copying a List	78
Wrapping Up	79
Review Questions	79
Chapter 7 - Learning About Tuples	81
Creating Tuples	81

CONTENTS

Working With Tuples	82
Concatenating Tuples	83
Special Case Tuples	83
Wrapping Up	84
Review Questions	84
Chapter 8 - Learning About Dictionaries	86
Creating Dictionaries	86
Accessing Dictionaries	87
Dictionary Methods	88
Modifying Your Dictionary	92
Deleting Items From Your Dictionary	92
Wrapping Up	93
Review Questions	93
Chapter 9 - Learning About Sets	95
Creating a Set	95
Accessing Set Members	96
Changing Items	97
Adding Items	97
Removing Items	98
Clearing or Deleting a Set	99
Set Operations	100
Wrapping Up	101
Review Questions	102
Chapter 10 - Boolean Operations and None	103
The bool() Function	103
What About None?	104
Wrapping Up	105
Review Questions	105
Chapter 11 - Conditional Statements	106
Comparison Operators	106
Creating a Simple Conditional	107
Branching Conditional Statements	108
Nesting Conditionals	109
Logical Operators	110
Special Operators	112
Wrapping Up	115
Review Questions	115
Chapter 12 - Learning About Loops	116
Creating a for Loop	116

CONTENTS

Looping Over a String	118
Looping Over a Dictionary	118
Extracting Multiple Values in a Tuple While Looping	119
Using enumerate with Loops	120
Creating a while Loop	120
Breaking Out of a Loop	121
Using continue	122
Loops and the else Statement	123
Nesting Loops	124
Wrapping Up	125
Review Questions	126
Chapter 13 - Python Comprehensions	127
List Comprehensions	127
Nested List Comprehensions	129
Dictionary Comprehensions	130
Set Comprehensions	130
Wrapping Up	131
Review Questions	131
Chapter 14 - Exception Handling	132
The Most Common Exceptions	132
Handling Exceptions	133
Raising Exceptions	135
Examining the Exception Object	135
Using the finally Statement	136
Using the else Statement	137
Wrapping Up	138
Review Questions	138
Chapter 15 - Working with Files	140
The open() Function	140
Reading Files	142
Reading Binary Files	143
Writing Files	143
Seeking Within a File	144
Appending to Files	145
Catching File Exceptions	145
Wrapping Up	146
Review Questions	146
Chapter 16 - Importing	147
Using import	147
Using from to Import Specific Bits & Pieces	150

CONTENTS

Using <code>as</code> to assign a new name	150
Importing Everything	151
Wrapping Up	152
Review Questions	152
Chapter 17 - Functions	153
Creating a Function	153
Calling a Function	154
Passing Arguments	154
Type Hinting Your Arguments	155
Passing Keyword Arguments	156
Required and Default Arguments	157
What are <code>*args</code> and <code>**kwargs</code> ?	158
Positional-only Parameters	160
Scope	162
Wrapping Up	164
Review Questions	164
Chapter 18 - Classes	166
Class Creation	167
Figuring Out <code>self</code>	168
Public and Private Methods / Attributes	171
Subclass Creation	172
Polymorphism	174
Making the Class Nicer	174
Wrapping Up	177
Review Questions	177
Part II - Beyond the Basics	178
Chapter 19 - Introspection	179
Using the <code>type()</code> Function	179
Using the <code>dir()</code> Function	180
Getting <code>help()</code>	180
Other Built-in Introspection Tools	182
Wrapping Up	185
Review Questions	185
Chapter 20 - Installing Packages with pip	186
Installing a Package	186
Exploring Command Line Options	188
Installing with <code>requirements.txt</code>	189
Upgrading a Package	190
Checking What's Installed	190

CONTENTS

Uninstalling Packages	191
Alternatives to pip	191
Wrapping Up	192
Review Questions	192
Chapter 21 - Python Virtual Environments	193
Python's venv Library	193
The virtualenv Package	194
Other Tools	195
Wrapping Up	196
Review Questions	196
Chapter 22 - Type Checking in Python	197
Pros and Cons of Type Hinting	197
Built-in Type Hinting / Variable Annotation	198
Collection Type Hinting	199
Hinting Values That Could be None	200
Type Hinting Functions	200
What To Do When Things Get Complicated	201
Classes	202
Decorators	202
Aliasing	203
Other Type Hints	204
Type Comments	204
Static Type Checking	205
Wrapping Up	206
Review Questions	207
Chapter 23 - Creating Multiple Threads	208
Pros of Using Threads	208
Cons of Using Threads	209
Creating Threads	209
Subclassing Thread	211
Writing Multiple Files with Threads	213
Wrapping Up	216
Review Questions	216
Chapter 24 - Creating Multiple Processes	217
Pros of Using Processes	217
Cons of Using Processes	218
Creating Processes with multiprocessing	218
Subclassing Process	219
Creating a Process Pool	221
Wrapping Up	222

CONTENTS

Review Questions	222
Chapter 25 - Launching Subprocesses with Python	223
The subprocess.run() Function	223
The subprocess.Popen() Class	225
The subprocess.Popen.communicate() Function	226
Reading and Writing with stdin and stdout	227
Wrapping Up	228
Review Questions	228
Chapter 26 - Debugging Your Code with pdb	229
Starting pdb in the REPL	230
Starting pdb on the Command Line	231
Stepping Through Code	232
Adding Breakpoints in pdb	233
Creating a Breakpoint with set_trace()	234
Using the built-in breakpoint() Function	235
Getting Help	236
Wrapping Up	237
Review Questions	237
Chapter 27 - Learning About Decorators	238
Creating a Function	238
Creating a Decorator	239
Applying a Decorator with @	241
Creating a Decorator for Logging	242
Stacking Decorators	244
Passing Arguments to Decorators	246
Using a Class as a Decorator	248
Python's Built-in Decorators	249
Python Properties	251
Wrapping Up	254
Review Questions	254
Chapter 28 - Assignment Expressions	255
Using Assignment Expressions	255
What You Cannot Do With Assignment Expressions	257
Wrapping Up	257
Review Questions	257
Chapter 29 - Profiling Your Code	258
Learning How to Profile with cProfile	258
Profiling a Python Script with cProfile	259
Working with Profile Data Using pstats	262

CONTENTS

Other Profilers	266
Wrapping Up	267
Review Questions	267
Chapter 30 - An Introduction to Testing	268
Using doctest in the Terminal	268
Using doctest in Your Code	271
Using doctest From a Separate File	272
Using unittest For Test Driven Development	274
Wrapping Up	282
Review Questions	283
Chapter 31 - Learning About the Jupyter Notebook	284
Installing The Jupyter Notebook	284
Creating a Notebook	285
Adding Content	289
Adding an Extension	296
Exporting Notebooks to Other Formats	297
Wrapping Up	298
Review Questions	298
Part III - Practical Python	299
Chapter 32 - How to Create a Command-line Application with argparse	300
Parsing Arguments	300
Creating Helpful Messages	303
Adding Aliases	306
Using Mutually Exclusive Arguments	307
Creating a Simple Search Utility	309
Wrapping Up	312
Review Questions	312
Chapter 33 - How to Parse XML	313
Parsing XML with ElementTree	313
Creating XML with ElementTree	316
Editing XML with ElementTree	318
Manipulating XML with lxml	319
Wrapping Up	323
Review Questions	323
Chapter 34 - How to Parse JSON	324
Encoding a JSON String	325
Saving JSON to Disk	325
Decoding a JSON String	326

CONTENTS

Loading JSON from Disk	327
Validating JSON with <code>json.tool</code>	327
Wrapping Up	328
Review Questions	328
Chapter 35 - How to Scrape a Website	329
Rules for Web Scraping	329
Preparing to Scrape a Website	330
Scraping a Website	332
Downloading a File	335
Wrapping Up	336
Review Questions	336
Chapter 36 - How to Work with CSV files	338
Reading a CSV File	338
Reading a CSV File with <code>DictReader</code>	340
Writing a CSV File	341
Writing a CSV File with <code>DictWriter</code>	343
Wrapping Up	344
Review Questions	345
Chapter 37 - How to Work with a Database Using <code>sqlite3</code>	346
Creating a SQLite Database	346
Adding Data to Your Database	348
Searching Your Database	350
Editing Data in Your Database	352
Deleting Data From Your Database	354
Wrapping Up	354
Review Questions	355
Chapter 38 - Working with an Excel Document in Python	356
Python Excel Packages	356
Getting Sheets from a Workbook	357
Reading Cell Data	358
Iterating Over Rows and Columns	360
Writing Excel Spreadsheets	362
Adding and Removing Sheets	363
Adding and Deleting Rows and Columns	365
Wrapping Up	367
Review Questions	367
Chapter 39 - How to Generate a PDF	368
Installing ReportLab	368
Creating a Simple PDF with the Canvas	369

CONTENTS

Creating Drawings and Adding Images Using the Canvas	371
Creating Multi-page Documents with PLATYPUS	375
Creating a Table	378
Wrapping Up	381
Review Questions	382
Chapter 40 - How to Create Graphs	383
Installing Matplotlib	383
Creating a Simple Line Chart with PyPlot	383
Creating a Bar Chart	385
Creating a Pie Chart	389
Adding Labels	392
Adding Titles to Plots	394
Creating a Legend	396
Showing Multiple Figures	398
Wrapping Up	403
Review Questions	403
Chapter 41 - How to Work with Images in Python	404
Installing Pillow	404
Opening Images	405
Cropping Images	409
Using Filters	411
Adding Borders	414
Resizing Images	419
Wrapping Up	423
Review Questions	424
Chapter 42 - How to Create a Graphical User Interface	425
Installing wxPython	426
Learning About Event Loops	426
How to Create Widgets	427
How to Lay Out Your Application	431
How to Add Events	435
How to Create an Application	436
Wrapping Up	444
Review Questions	445
Part IV - Distributing Your Code	446
Chapter 43 - How to Create a Python Package	447
Creating a Module	447
Creating a Package	449
Packaging a Project for PyPI	451

CONTENTS

Creating Project Files	451
Creating setup.py	452
Generating a Python Wheel	453
Uploading to PyPI	454
Wrapping Up	455
Review Questions	456
Chapter 44 - How to Create an Exe for Windows	457
Installing PyInstaller	457
Creating an Executable for a Command-Line Application	458
Creating an Executable for a GUI	460
Wrapping Up	463
Review Questions	463
Chapter 45 - How to Create an Installer for Windows	464
Installing Inno Setup	464
Creating an Installer	465
Testing Your Installer	478
Wrapping Up	478
Review Questions	478
Chapter 46 - How to Create an “exe” for Mac	479
Installing PyInstaller	479
Creating an Executable with PyInstaller	479
Wrapping Up	482
Review Questions	483
Afterword	484
Appendix A - Version Control	485
Version Control Systems	485
Distributed vs Centralized Versioning	486
Common Terminology	486
Python IDE Version Control Support	488
Wrapping Up	489
Appendix B - Version Control with Git	490
Installing Git	490
Configuring Git	491
Creating a Project	492
Ignoring Files	493
Initializing a Repository	493
Checking the Project Status	493
Adding Files to a Repository	494

CONTENTS

Committing Files	495
Viewing the Log	495
Changing a File	496
Reverting a File	497
Checking Out Previous Commits	498
Pushing to Github	500
Wrapping Up	500
Review Question Answer Key	501
Chapter 3 - Documenting Your Code	501
Chapter 4 - Working with Strings	501
Chapter 5 - Numeric Types	503
Chapter 6 - Learning About Lists	504
Chapter 7 - Learning About Tuples	505
Chapter 8 - Learning About Dictionaries	506
Chapter 9 - Learning About Sets	507
Chapter 10 - Boolean Operations and None	508
Chapter 11 - Conditional Statements	509
Chapter 12 - Learning About Loops	510
Chapter 13 - Python Comprehensions	511
Chapter 14 - Exception Handling	512
Chapter 15 - Working with Files	512
Chapter 16 - Importing	513
Chapter 17 - Functions	514
Chapter 18 - Classes	515
Chapter 19 - Introspection	516
Chapter 20 - Installing Packages with pip	516
Chapter 21 - Python Virtual Environments	517
Chapter 22 - Type Checking in Python	518
Chapter 23 - Creating Multiple Threads	518
Chapter 24 - Creating Multiple Processes	519
Chapter 25 - Launching Subprocesses with Python	519
Chapter 26 - Debugging Your Code	520
Chapter 27 - Learning About Decorators	521
Chapter 28 - Assignment Expressions	521
Chapter 29 - Profiling Your Code	522
Chapter 30 - An Introduction to Testing	522
Chapter 31 - Learning About the Jupyter Notebook	523
Chapter 32 - How to Create a Command Line Application with argparse	523
Chapter 33 - How to Parse XML	524
Chapter 34 - How to Parse JSON	524
Chapter 35 - How to Scrape a Website	525
Chapter 36 - How to Work with CSV files	525

CONTENTS

Chapter 37 - How to Work with a Database Using <code>sqlite</code>	526
Chapter 38 - Working with an Excel Document in Python	527
Chapter 39 - How to Generate a PDF	527
Chapter 40 - How to Create Graphs	528
Chapter 41 - How to Work with Images in Python	528
Chapter 42 - How to Create a Graphical User Interface	529
Chapter 43 - How to Create a Python Package	529
Chapter 44 - How to Create an Exe for Windows	530
Chapter 45 - How to Create an Installer for Windows	531
Chapter 46 - How to Create an “exe” for Mac	531

About the Technical Reviewers

Ethan Furman

Ethan, a largely self-taught programmer, discovered Python around the turn of the century, but wasn't able to explore it for nearly a decade. When he finally did, he fell in love with its simple syntax, lack of boiler-plate, and the ease with which one can express one's ideas in code. After writing a dbf library to aid in switching his company's code over to Python, he authored PEP 409, wrote the Enum implementation for PEP 435, and authored PEP 461. He was invited to be a core developer after PEP 435, which he happily accepted.

He thanks his mother for his love of language, stories, and the written word.

Martin Breuss

With a background in education, Martin started to learn programming mostly by himself through online resources after finishing university. Since then, he's worked as a curriculum developer, programming mentor, code reviewer, and Python bootcamp instructor. Quality education, combined with figuring out how to have fun while effectively learning unfamiliar topics, has always been a big interest in his life. Currently, he creates content for Real Python as well as online and in-person courses for CodingNomads, a community for people learning to code.

If you are just starting out with computer programming, keep in mind that learning something new takes time and effort, and that it will be easier if you have fun while doing it. Some learning strategies that have been helpful for Martin are (1) focusing on projects that interest him personally and (2) actively participating in communities of motivated learners.

Acknowledgments

Writing is a time consuming process. For Python 101's 2nd Edition, I wanted to get some great technical reviewers so that the book would be better than ever. I want to thank Martin Breuss for his many insightful comments and useful suggestions. I was also honored to have Python core developer Ethan Furman as a technical reviewer and editor. His time spent ensuring correct code and improving content is appreciated.

I also want to thank the many people who have supported me through Kickstarter and my blog. I have many amazing readers and friends, such as Steve Barnes who gave me some great feedback on the first few chapters of this book and Michal who helped with some of the chapters in part III. There are so many of you who have helped me learn new things about Python and been a wonderful community.

Thank you!

Mike

Introduction

Welcome to the 2nd Edition of **Python 101**! The original Python 101 came out in the summer of 2014 and was written with Python 3.5 in mind. The 2nd Edition of this book has been completely updated and rearranged for the latest version of Python, which at the time of writing is 3.8.

Some publishers / authors will only do minor updates when creating a new edition of a book. That is not how I roll. I have personally gone through the entire book and updated every single chapter. I have removed content that was either no longer relevant or could lead to confusion to readers. I have also added several new chapters to the book that cover such things as using version control and setting up projects.

Many programming books will only teach you the basics of the language. With **Python 101**, the goal is to help you not only learn the basics of the language but to go beyond the basics and dig into some intermediate level material. The reason for this is that you usually need to know more than the basics to create something valuable.

Because of this, the book will be split up into the following four parts:

- Part one will cover Python's basics
- Part two will be intermediate material
- Part three will be a series of small tutorials
- Part four will cover Python packaging and distribution

Note that not all sections will be the same length.

Let's go ahead and talk about each of these sections in turn!

Part I - The Basics

This is the heart of the book. In this section you will learn all the basics that you need to know to start using Python effectively. Each chapter will teach you something new and they are ordered in such a way that they will build on each other. If you already know Python well, then you can skip this section and move on to **Part II**.

Part II - Intermediate Materials

Now that you know how Python works, you can dive into more intermediate level material. In this section, you will learn about the following topics:

- Virtual environments
- Type hinting
- Threads and Processes
- Debugging
- Decorators
- Code profiling
- Basic testing

These topics cover some intermediate level Python and also help you learn some key software development skills, like knowing how to debug your code, add basic unit tests, and use version control.

Part III - Tutorials

This part of the book is where you will put it all together. You will learn how to use Python with some real world scripts. These scripts will be basic, but they will demonstrate the power of Python and what you can do with it.

Here is what will be covered:

- How to Create a Command Line Application
- How to Parse XML
- How to Parse JSON
- How to Scrape a Website
- How to Work with CSV Files
- How to Work with a SQLite Database
- How to Create an Excel Document
- How to Generate a PDF

Part IV - Python Packaging and Distribution

Now that you know how to write programs, you will probably want to know how to share them with your friends. In this section, you will learn how to transform your code into something that other developers or users can use.

Specifically you will learn how to:

- Create a Cross Platform Python Package
- Create an Exe for Windows
- Create an Installer for Windows
- Create an “exe” for Mac

By the end of this section, you should be able to confidently distribute your code all on your own!

Target Audience

This book is written for people that have used other programming languages or taken some computer science or related classes. While this book won't handhold you through all the terminology, it will help you learn how to use Python effectively. It also covers some intermediate level topics that most beginner books do not.

About the Author

Michael Driscoll has been programming with Python for more than a decade. He is active in multiple Python communities and is a contributor for Real Python. Mike has also been blogging about Python at <http://www.blog.pythonlibrary.org/> for many years and has written several books about Python:

- Python 101 (1st Edition)
- Python 201: Intermediate Python
- wxPython Recipes
- Python Interviews
- ReportLab: PDF Publishing with Python
- Jupyter Notebook 101
- Creating GUI Applications with wxPython

Conventions

All technical books have their own conventions for how things are presented. In this book, new topics will be in **bold**. When referring to Python related keywords or code in a sentence, they will be in monospace.

Code blocks will look like this:

```
1 def greeter(name: str) -> None:  
2     print(f'Hello {name}')  
3  
4 greeter('Mike')
```

There will also be blocks of code that represent Python's interactive interpreter, also known as a REPL:

```
1 >>> name = 'Mike'  
2 >>> print(f'My name is {name}')  
3 My name is Mike
```

This demonstrates how the interpreter should behave.

Requirements

You will need the Python language to follow along in this book. See chapter 1 for installation details or go get the official Python distribution for free at:

- <http://python.org/download/>

If you need anything beyond what comes with Python, the chapter will tell you how to install it.

Book Source Code

The book's source code can be found on Github:

- <https://github.com/driscollis/python101code>

Reader Feedback

If you enjoyed the book or have any other kind of feedback, I would love to hear from you. You can contact me at the following:

- comments@pythonlibrary.org

Errata

I try my best not to publish errors in my writings, but it happens from time to time. If you happen to see an error in this book, feel free to let me know by emailing me at the following:

* errata@pythonlibrary.org

Now let's get started!

Part I - The Python Language

The first section of this book is dedicated to helping you learn the Python programming language. You will learn all the basics that you need to know to understand Python's syntax. You may not know how to put it all together when you finish this section, but that is the purpose of the following sections.

In this part of the book, you will find:

- Chapter 1 - Installing Python
- Chapter 2 - Python Editors
- Chapter 3 - Documenting Your Code
- Chapter 4 - Working with Strings
- Chapter 5 - Numeric Types
- Chapter 6 - Learning About Lists
- Chapter 7 - Learning About Tuples
- Chapter 8 - Learning About Dictionaries
- Chapter 9 - Learning About Sets
- Chapter 10 - Boolean Operations and None
- Chapter 11 - Conditional Statements
- Chapter 12 - Learning About Loops
- Chapter 13 - Python Comprehensions
- Chapter 14 - Exception Handling
- Chapter 15 - Working with Files
- Chapter 16 - Importing
- Chapter 17 - Functions
- Chapter 18 - Classes

Once you have finished this section, you will be able to understand the basics of Python and its syntax.

Let's get started now!

Chapter 1 - Installing Python

Depending on which operating system you are using, you may need to install the Python programming language. This chapter will cover the primary ways to install Python.

First of all, there are several different versions of Python, which are called “distributions”. A distribution is a word used to describe a collection of software. A Python distribution will include the core Python language at a minimum and sometimes include extra 3rd party libraries.

The official version is called Python or CPython and you can get it from the following:

- <https://www.python.org/>

Another popular distribution of Python is called **Anaconda** and comes from the Anaconda company. This variation of Python is focused on data science and includes many additional 3rd party packages in addition to Python’s standard library. You can read more about it here:

- <https://www.anaconda.com/>

Anaconda is designed to use a command-line tool called **conda** for installing additional packages whereas Python uses **pip**, although you can also use **pip** with Anaconda. Also note that the Anaconda download is much larger than the official Python one is because it has so many extra packages included with it.

If you are on a Windows PC and don’t have administrator privileges on your machine, then you might want to check out **WinPython**, which can be run from a USB:

- <https://winpython.github.io/>

There are many other Python distributions to choose from. You can see a bunch more here:

- <https://wiki.python.org/moin/PythonDistributions>

This book is focused on Python 3. The current version at the time of writing is Python 3.8. It is recommended that you use the official Python distribution rather than Anaconda, although the examples in this book should work for both. Any examples that use a specific feature only found in 3.8 or newer will be noted as such.

There are 32-bit and 64-bit distributions of Python. If you are unsure what your computer uses, you should opt for the 32-bit version if that is available. Newer Macs no longer support 32-bit, so in that case you only have one choice.

Installing on Windows

The <https://www.python.org/> website has a download section where you can download an installer for Python.

After the installer is downloaded, double-click it and go through the installation wizard. Here is the first screen you should see:



Fig. 1-1: Installing Python 3.8 via the installer

There is a checkbox in the wizard for adding Python to the path. If you don't have an older version of Python already installed or if you want to use the latest as your default Python, then I recommend that you check that checkbox. It is unchecked by default, as shown in the image above.

If you install Python to your path, it will allow you to run Python from the command line (cmd.exe) or Powershell by just typing `python`. If you install Python using the Windows Store, it will automatically add Python to your path.

The next page of the installation wizard allows you to enable or disable optional features:

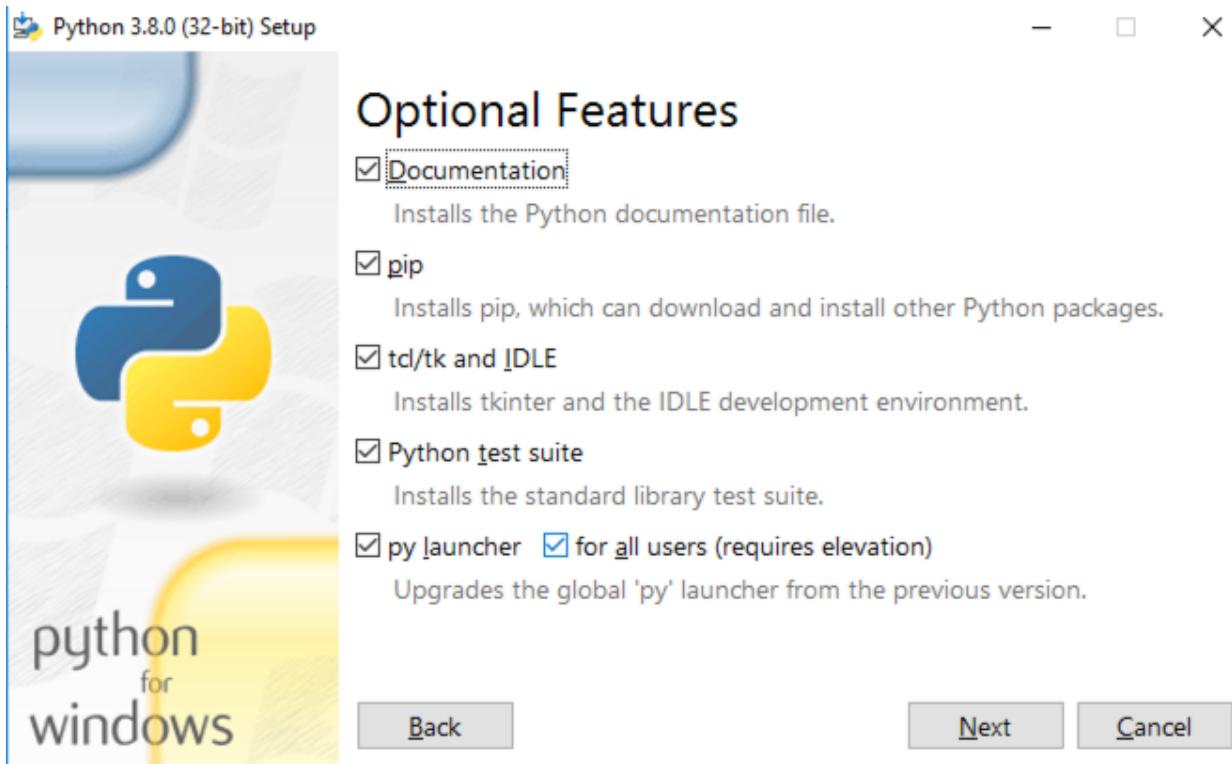


Fig. 1-2: Enabling Optional Python features

You can leave the defaults enabled, but if you are short on space, you should untick the “Python Test Suite” option. The next page of the wizard will allow you to enable Advanced Options:

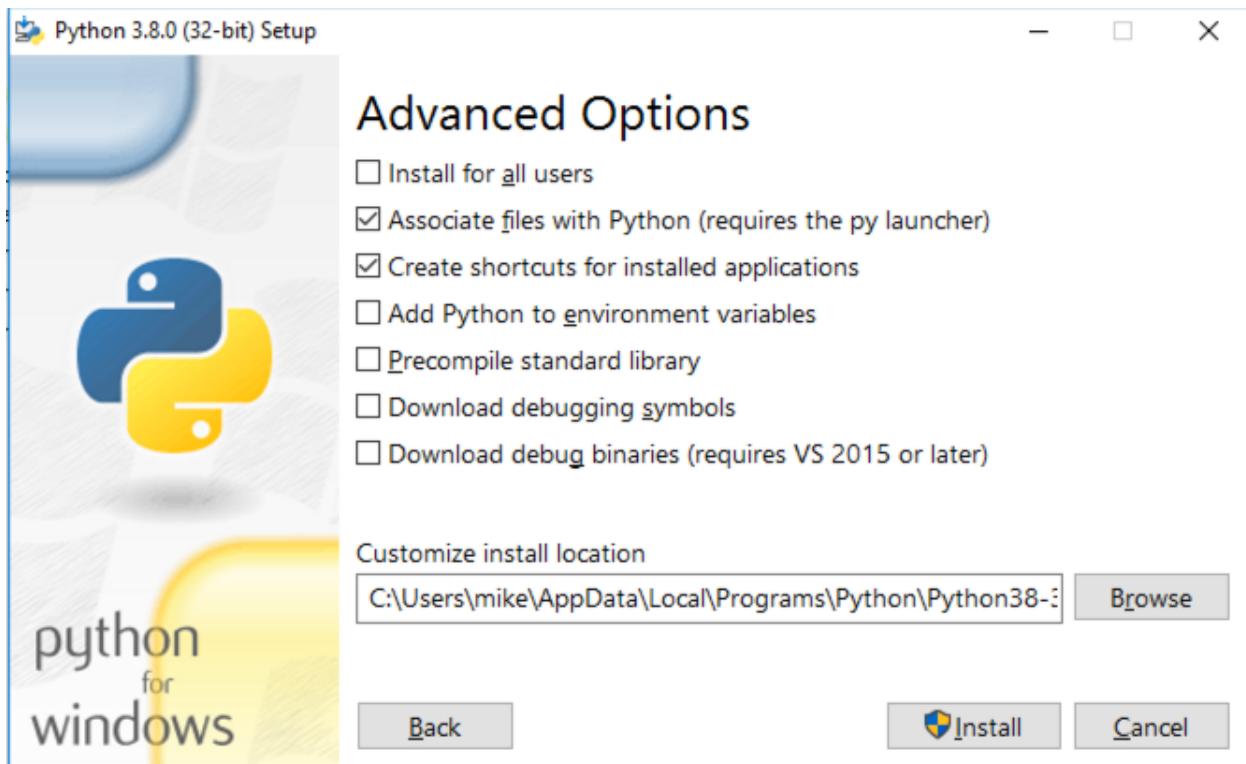


Fig. 1-3: Advanced Options

Here you can install Python for all users of the machine. You can also associate Python files with Python, create shortcuts, add Python to your environment and more. Most of the time, the defaults will be fine. However it's a good idea to go ahead and check the “Precompile standard library” as that can make Python run better on your machine.

When you press **Next** here, you will probably get a warning from Windows’s User Access Control:

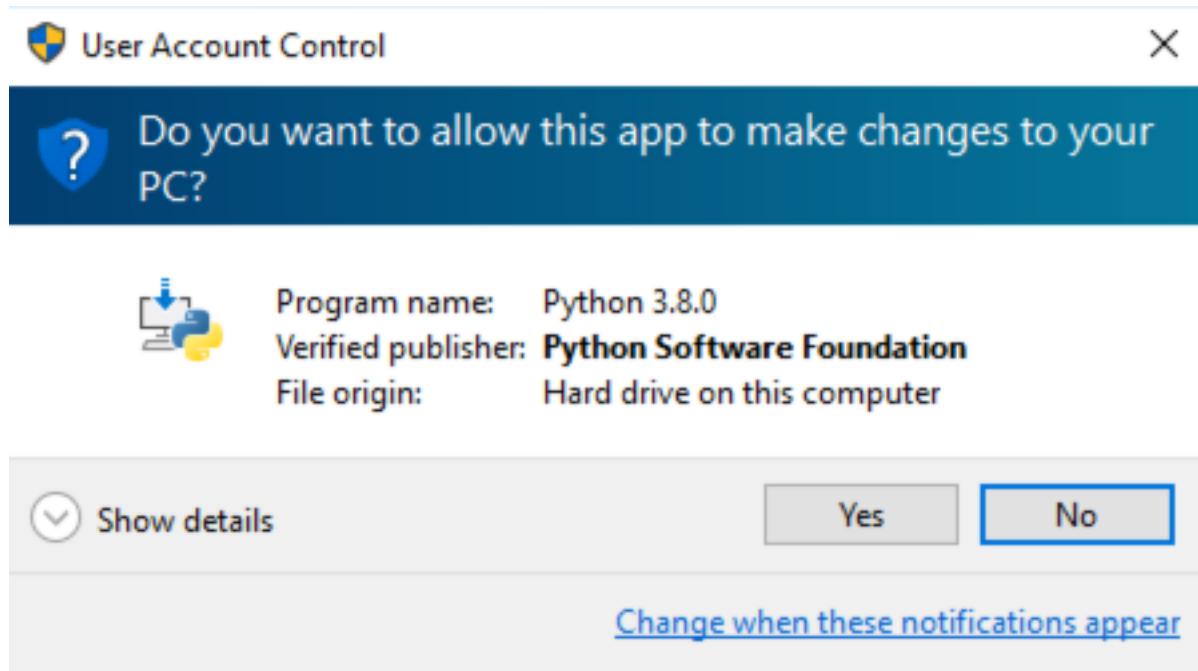


Fig. 1-4: User Access Control Warning

This is a verification step that asks if you really want to proceed with installing Python. Go ahead and press Yes. Now Python is being installed:

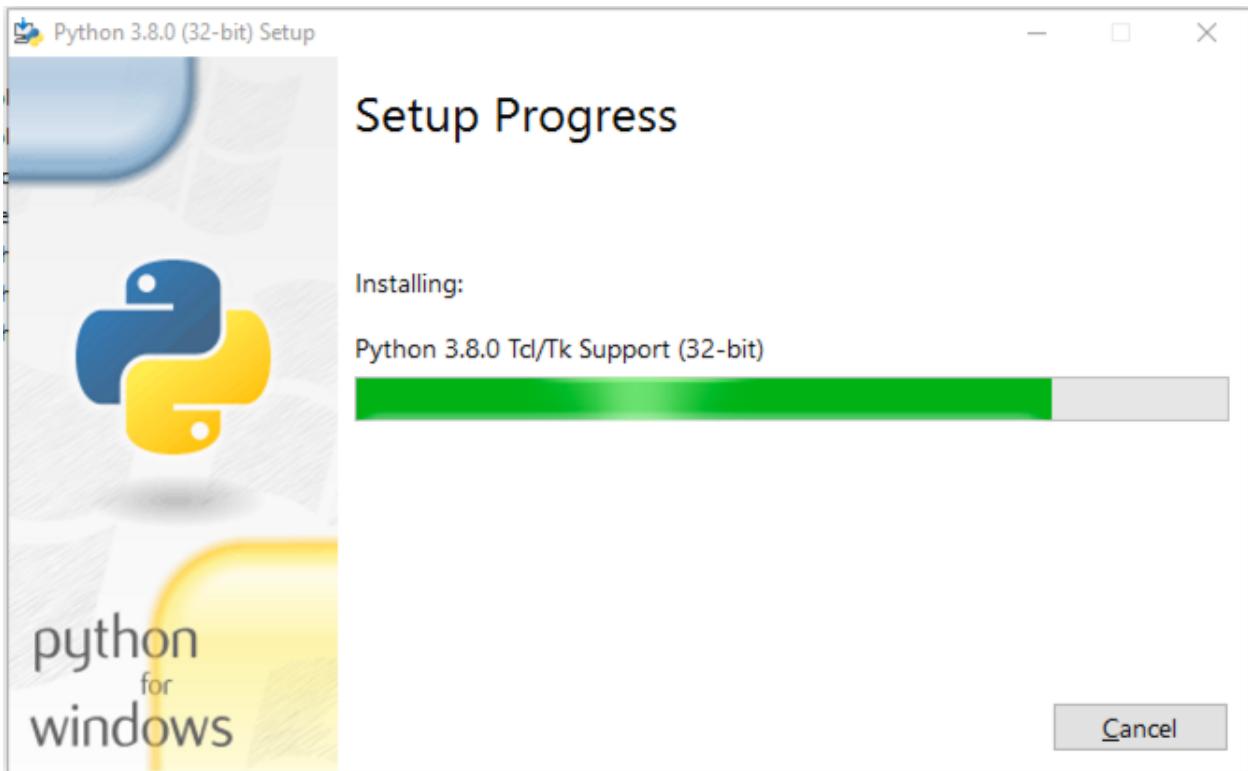


Fig. 1-5: Installing Python

Once the install finishes, you will see this final dialog:

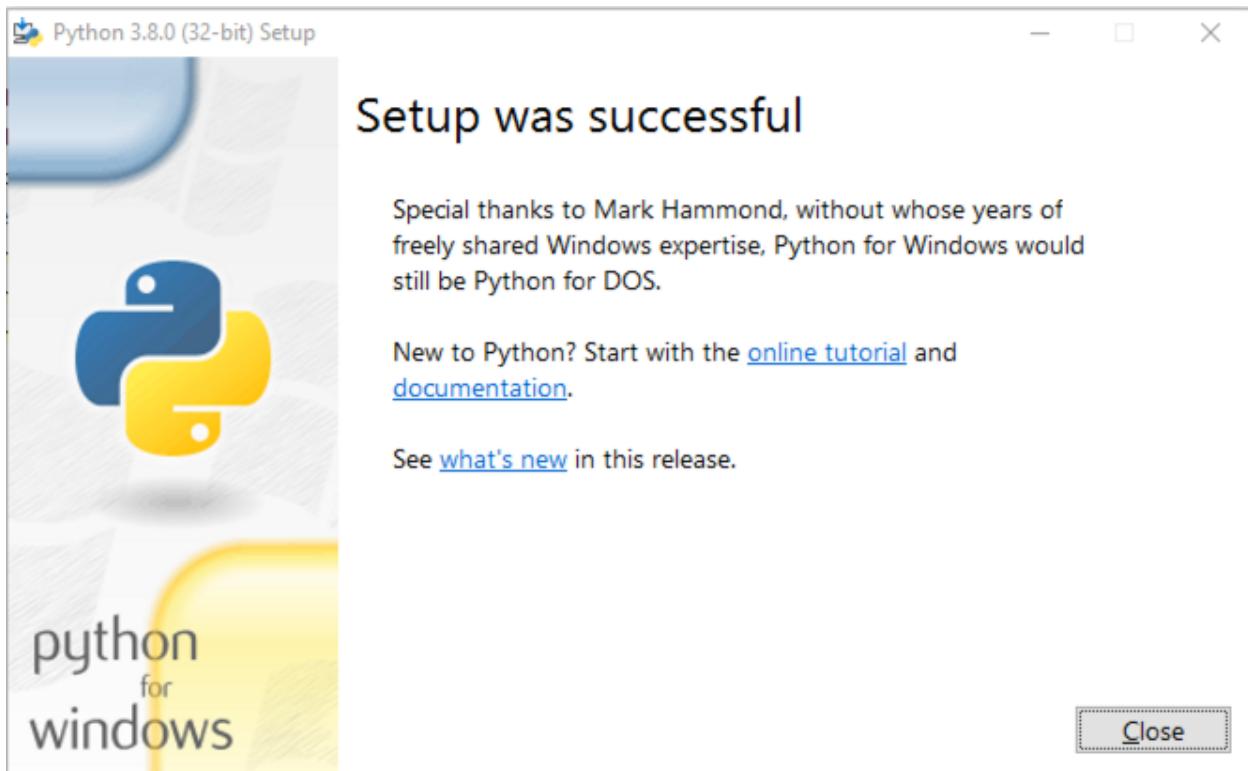


Fig. 1-6: Setup Successful

You now have Python installed on your Windows machine! Try running Python by opening cmd.exe:

A screenshot of a Command Prompt window titled "Command Prompt - python". The window shows the following text:

```
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\mike>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The window has standard Windows-style minimize, maximize, and close buttons at the top right.

Fig. 1-7: Running Python in cmd.exe

You should see something like the above. To exit, you can press **CTRL+D** on Linux and Mac, **CTRL+Z** on Windows, or type `exit()` on any platform and press **Enter**.

Installing on Mac

Macs usually come with Python pre-installed. However, if you want the latest version of Python, then you may need to download Python.

The <https://www.python.org/> website has a download section where you can also download a Python installer for Mac. Once downloaded, you will need to double-click it to install Python. You may need to tell your Mac to allow you to install Python as the system might bring up a dialog box that warns you about installing programs downloaded from the internet.

Note that the App Store does not have Python in it, so using the Python website is the way to go.

Let's take a moment to learn how to install Python on a Mac. Here's the first thing you will see when you double-click the downloaded pkg file:

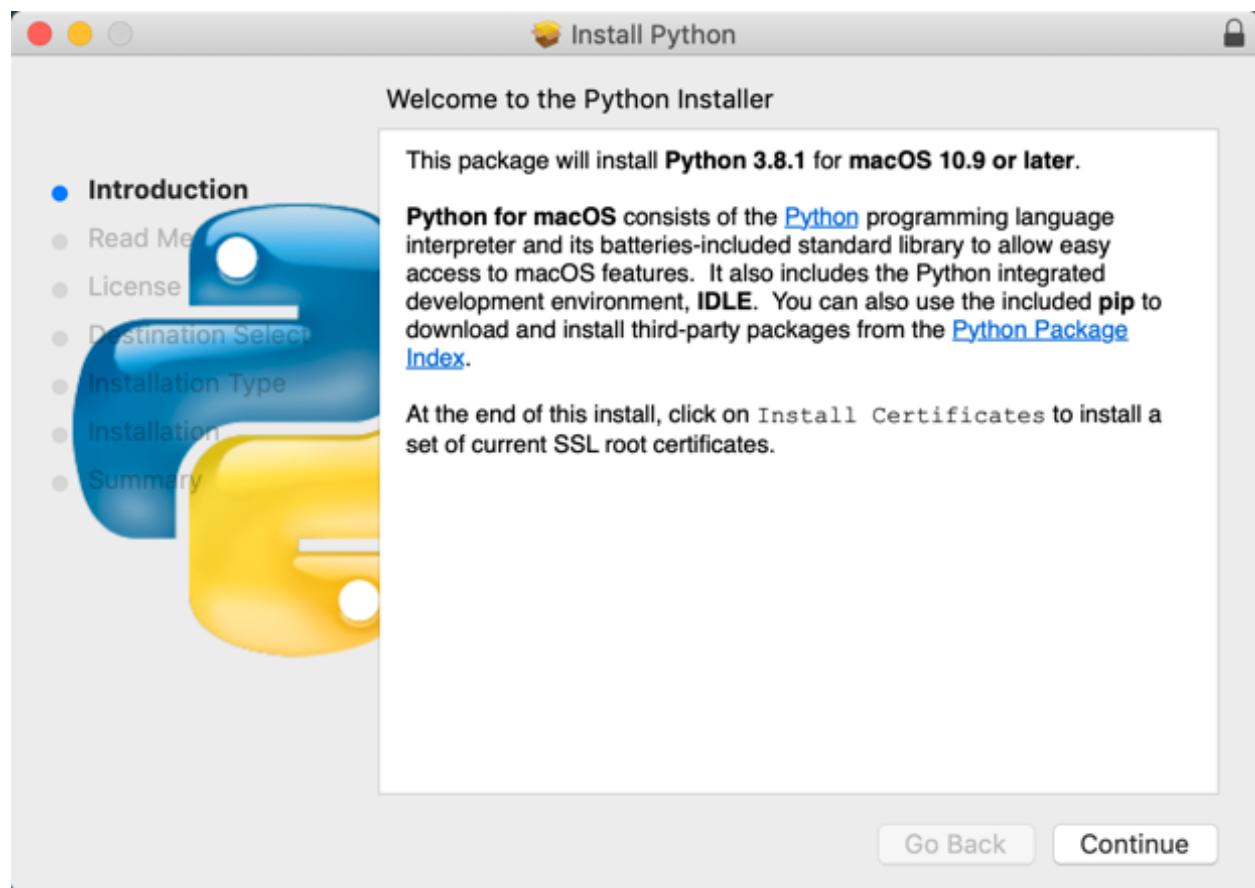


Fig. 1-8: Installing Python on Mac OSX

This screen basically tells you what you are about to install and that you will need to install some SSL certificates as well. Go ahead and press **Continue**:

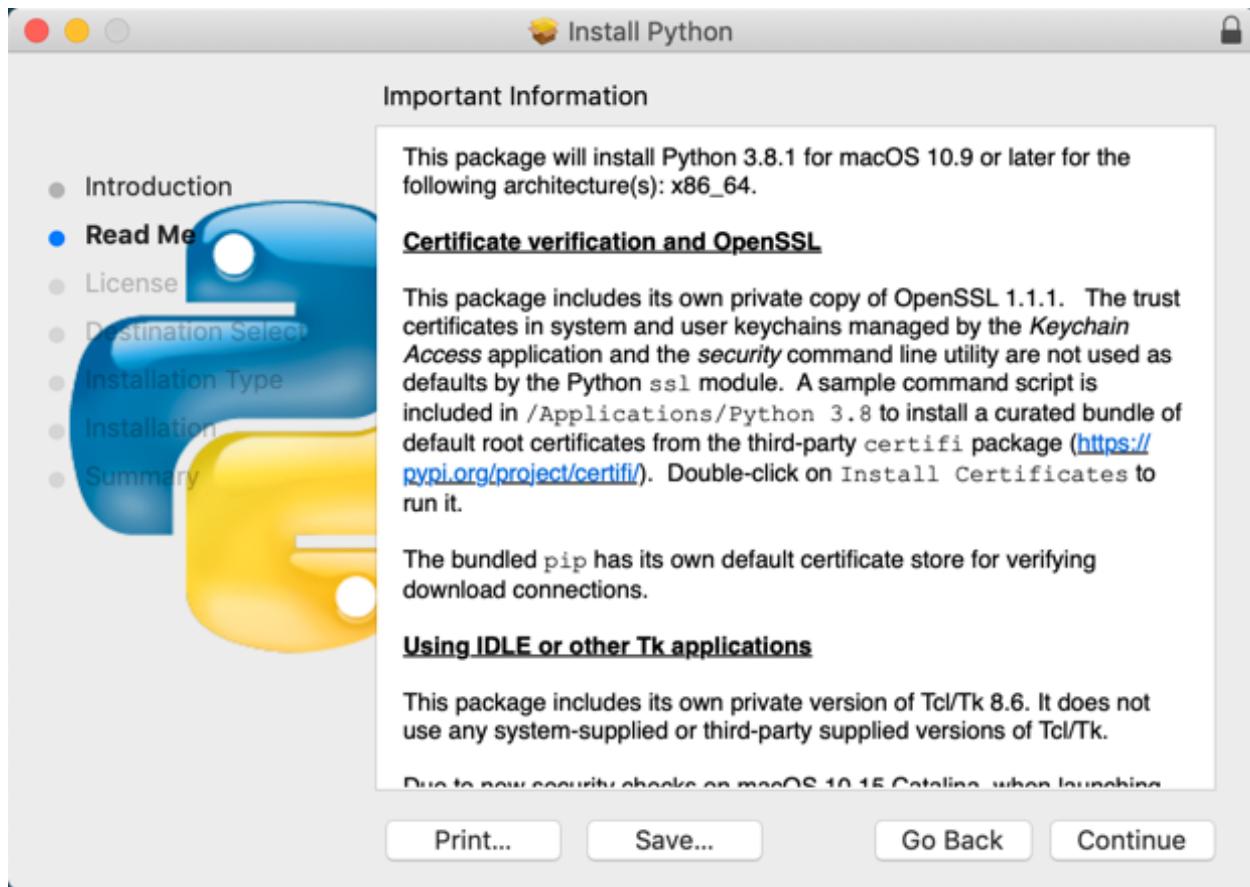


Fig. 1-9: Installing Python on Mac OSX (Read More)

This page gives you more information about the SSL certificate as well as general information about using IDLE and Tkinter on a Mac. IDLE is Python's built-in code editor. You will learn more about that in the next chapter. Tkinter is a Python library that you can use to create cross-platform graphical user interfaces.

Tkinter is the library that is used to create IDLE, the Python editor that comes with Python.

If you are interested, read through the information on this page. Otherwise, go ahead and **Continue**:



Fig. 1-10: Installing Python on Mac OSX (License Agreement)

This is Python's license agreement page. It also has a little bit of history about Python on it. Feel free to check it out or skip it and press **Continue**:



Fig. 1-11: Installing Python on Mac OSX (Install Destination)

This page allows you to choose which disk on your computer you want to install Python. I usually use the default, but if you want you can change it to some other location.

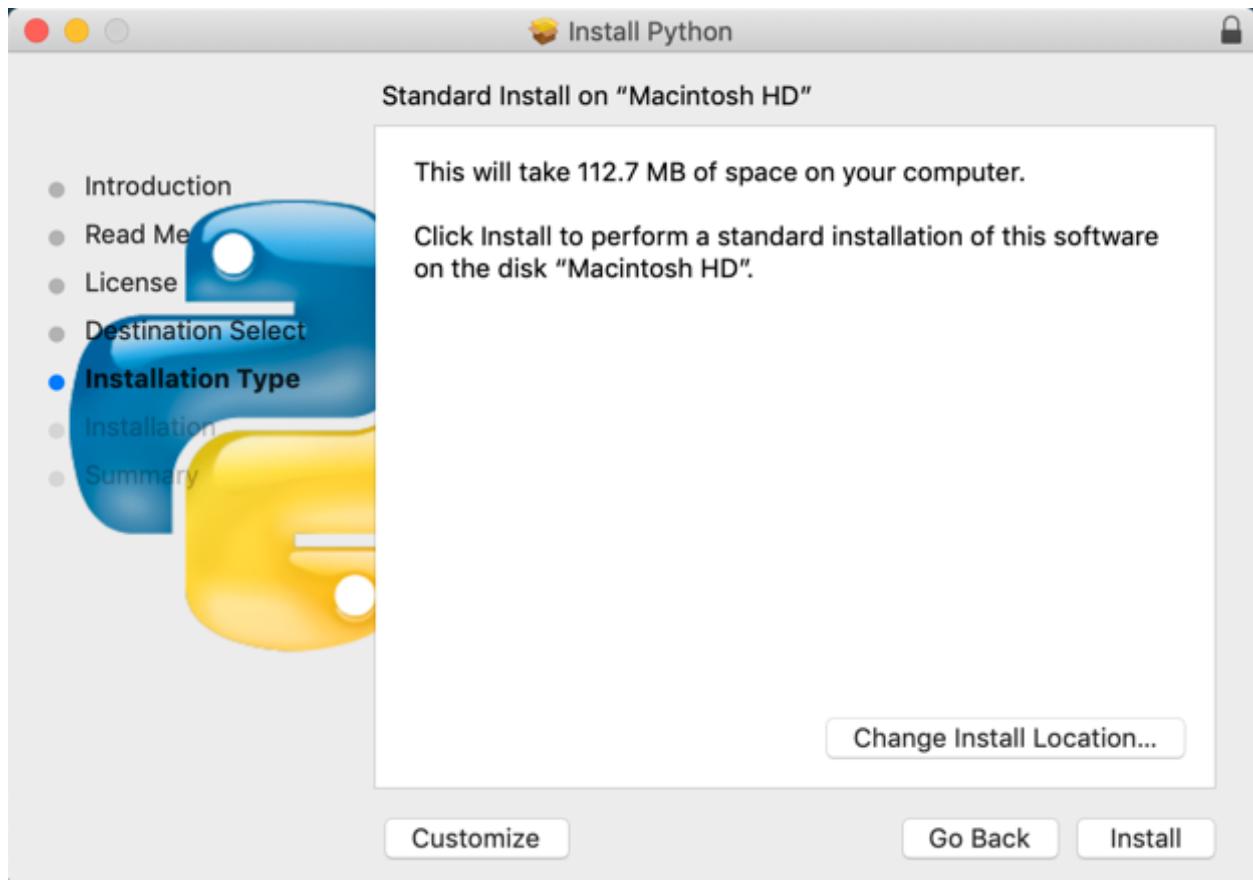


Fig. 1-12: Installing Python on Mac OSX (Standard Install)

This page allows you to choose *which folder* to install Python to, in contrast to the previous page which lets you pick *which disk* to install to.

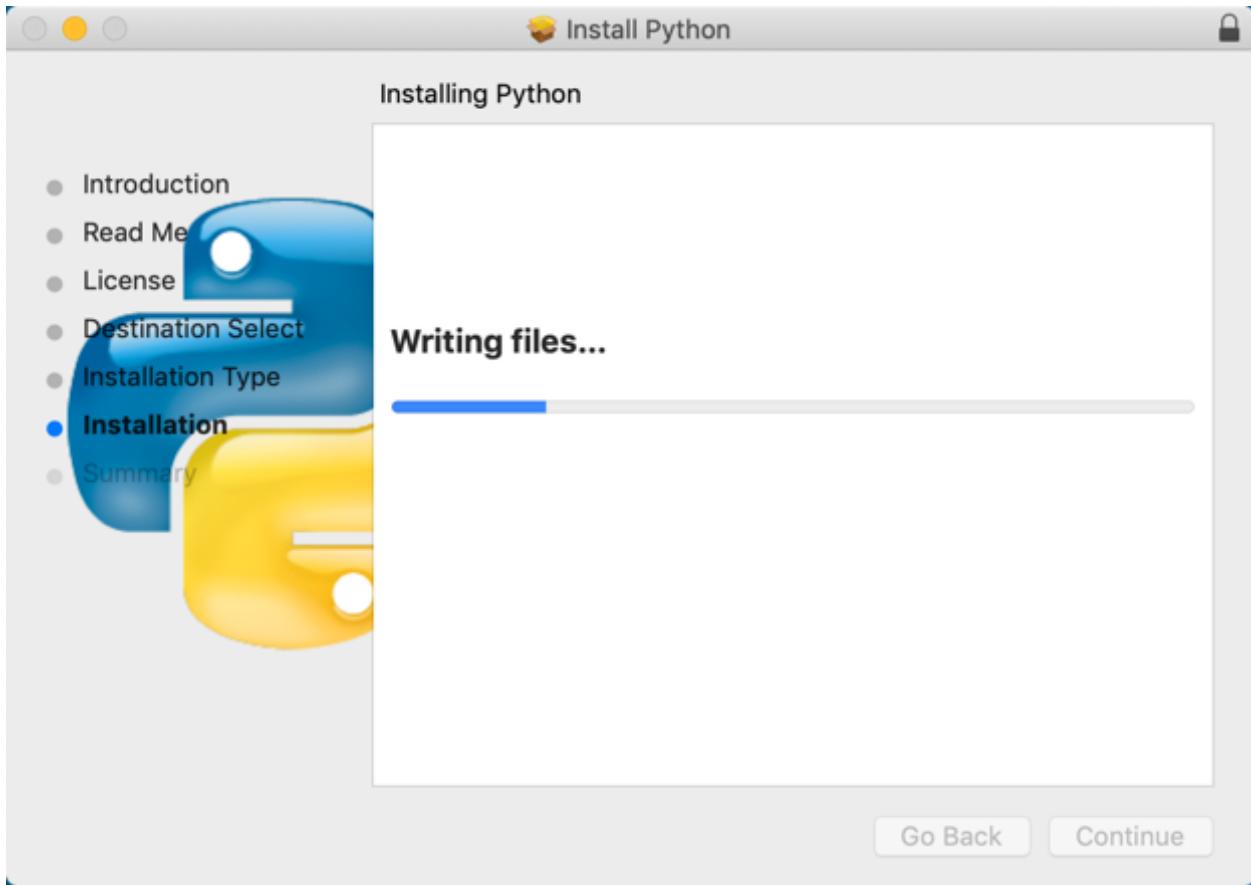


Fig. 1-13: Installing Python on Mac OSX (Actual Installation)

This page shows the installation as it happens. You can wait and watch Python get installed, or go get something to drink. Either way, Python will be installed before too long.



Fig. 1-14: Installing Python on Mac OSX (Finished)

Here is the last page of the installation wizard.

Installing on Linux

Linux also usually comes with Python pre-installed, although it will likely not be the latest version of Python. The Python website has source downloads and directions that you can use to build the latest Python for your Linux distribution.

The source download is Python source code with some build scripts included that will allow you to build and install Python on Linux.

For full instructions, you should read the dev guide:

- <https://devguide.python.org/setup/>

Sometimes you can also install a pre-built copy of the latest Python using your package manager. One good source for Python on Linux is the “deadsnakes” PPA. You will need to use Google to find it, but that makes it much easier to install another version of Python on Linux.

Android / iOS

You can also run Python on Android and iOS via downloadable applications. **Pydroid** is a popular application for Android while **Pythonista** is one of the popular choices for iOS. Trying to write code on a phone can be really problematic due to the on-screen keyboards. If you must go this route, you may want to use a tablet.

Other Operating Systems

Python can run on Raspberry Pi as well. If you do not have a computer, this is one of the most cost effective ways to get Python as a Raspberry Pi can cost as little as \$10. Of course, you will need to hook it up to a monitor, keyboard and mouse. Most of the time, a Raspberry Pi will also need an SD card for storage. But they are a very feasible development environment.

Other Python Variants

In addition to Anaconda, Python has several other variants that are worth mentioning:

- Jython - an implementation of Python written in Java that allows you to use Java code in Python
- IronPython - an implementation of Python written in .NET
- PyPy - written in RPython, PyPy has a just-in-time (JIT) compiler that makes it much faster than regular Python

The main reason for trying out these other implementations of Python is for speed or flexibility. For example, if you are already familiar with .NET or Java, then you might find IronPython or Jython a bit easier to jump into. Another reason to use Jython or IronPython is because you have pre-existing code in Java or .NET that you still need to use with Python.

In the case of PyPy, I usually recommend it if you have a slow Python program and you need a simple way to speed it up. Try running it with PyPy and you might be surprised at the performance improvement. Note that none of these variants are completely compatible with all of Python's 3rd party packages, so if your program uses one it may not work with these variants.

Wrapping Up

Most of the time, installing Python is straight-forward and easy to do. It can get tricky when you need to have multiple versions of Python installed on your machine at the same time, but if you are just starting out I think you'll find the installation process pretty painless.

Now that you have Python installed, you can congratulate yourself. You have started on a new endeavor and you have just taken the first step! However, before you try running Python, you may want to read the next chapter where you will learn about additional tools that will help you get the most out of your Python adventure!

Chapter 2 - Python Editors

The Python programming language comes with its own built-in Integrated Development Environment (IDE) called IDLE. The name, IDLE, supposedly came from the actor, Eric Idle, who was a part of the Monty Python troupe, which is what Python itself is named after.

IDLE comes with Python on Windows and some Linux variants. You may need to install IDLE separately on your particular flavor of Linux or on Mac if you plan to use the Python that came with the operating system. You should check out the Python website for full instructions on how to do so as each operating system is different.

Here are some of the reasons that Integrated Development Environments are useful:

- They provide syntax highlighting which helps prevent coding errors
- Autocomplete of variable names and built-in names
- Breakpoints and debugging.

On that last point, breakpoints tell the debugger where to pause execution. Debugging is the process of going through your code step-by-step to figure out how it works or to fix an issue with your code.

IDLE itself has other attributes that are useful, such as access to Python documentation, easy access to the source code via the Class Browser, and much more. However, IDLE is not the only way to code in Python. There are many useful IDEs out there. You can also use a text editor if you prefer. Notepad, SublimeText, Vim, etc., are examples of text editors. Text editors do not have all the features that a full-fledged IDE has, but tend to have the advantage of being simpler to use.

Here is a shortlist of IDEs that you can use to program in Python:

- PyCharm
- Wing Python IDE
- VS Code (also called Visual Studio Code)
- Spyder
- Eclipse with PyDev

PyCharm and WingIDE both have free and paid versions of their programs. The paid versions have many more features, but if you are just starting out, their free offerings are quite nice. VS Code and Spyder are free. VS Code can also be used for coding in many other languages. Note that to use VS Code effectively with Python, you will need to install a Python extension. You can also use the PyDev plugin for Eclipse to program in Python.

Other popular editors for Python include SublimeText, vim, emacs, and even Notepad++. These editors may not be 100% up-to-date on the syntax of the language, but you can use them for multiple programming languages.

But let's back up a bit and talk about Python's basic console, also known as the REPL, which stands for Read Evaluate Print Loop.

What About the REPL?

REPL or READ, EVAL, PRINT, LOOP is basically Python's interpreter. Python allows you to type code into an interpreter which will run your code live and let you learn about the language. You can access the interpreter, or REPL, by running Python in your terminal (if you are on Mac or Linux) or command console (if you are on Windows).

On Windows, you can go to the Start menu and search for cmd or "Command Prompt" to open the console or terminal:

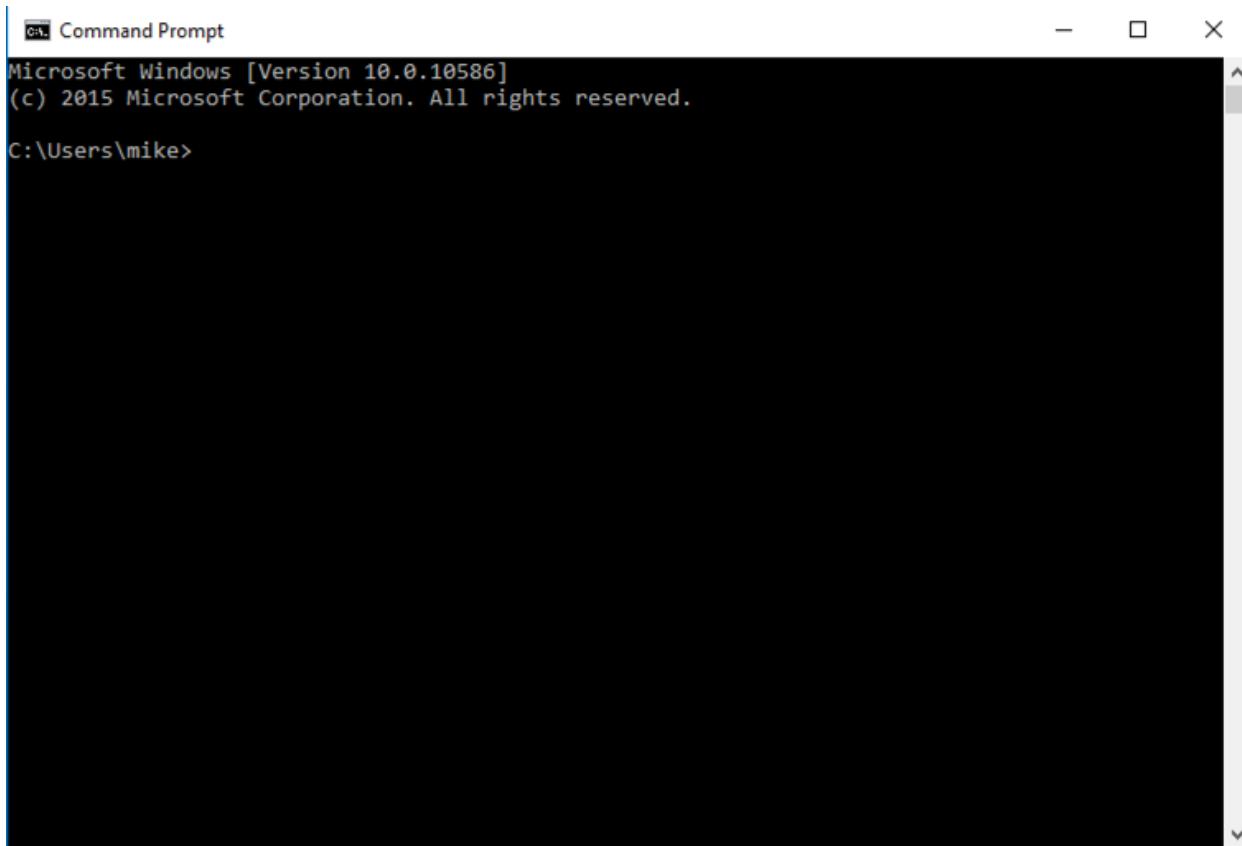
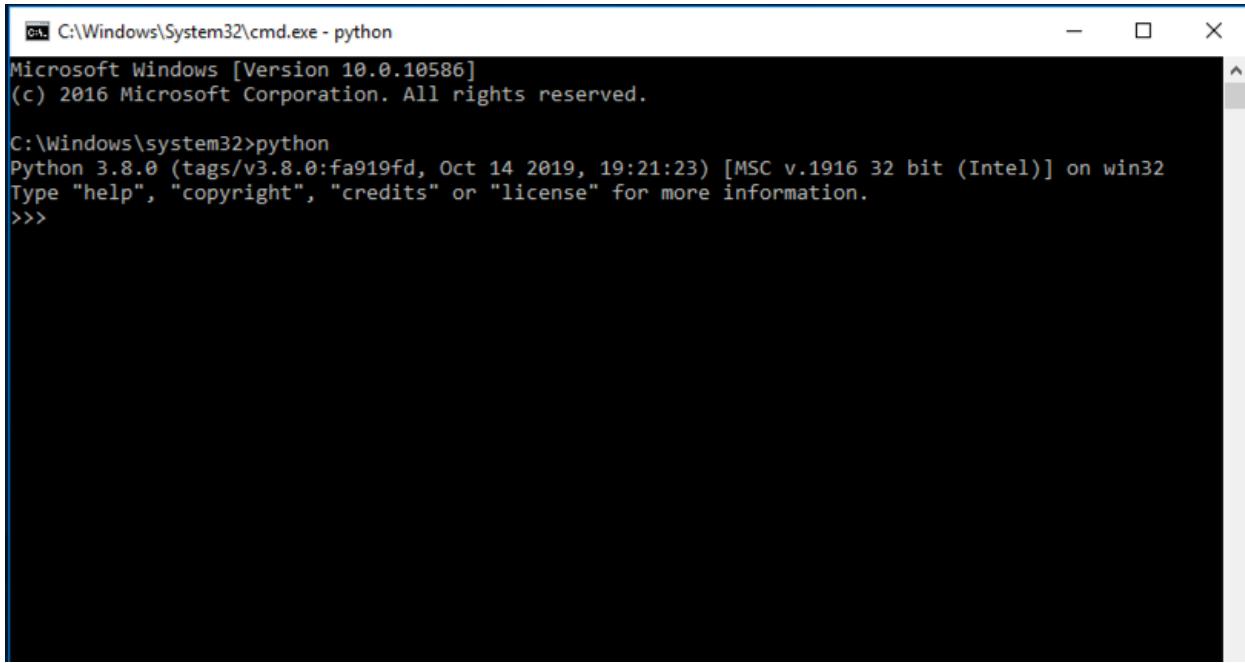


Fig. 2-1: Running the Command Prompt in Windows

Once you have the terminal open you can try typing `python`. You should see something like this:



A screenshot of a Windows command prompt window titled "cmd C:\Windows\System32\cmd.exe - python". The window displays the following text:

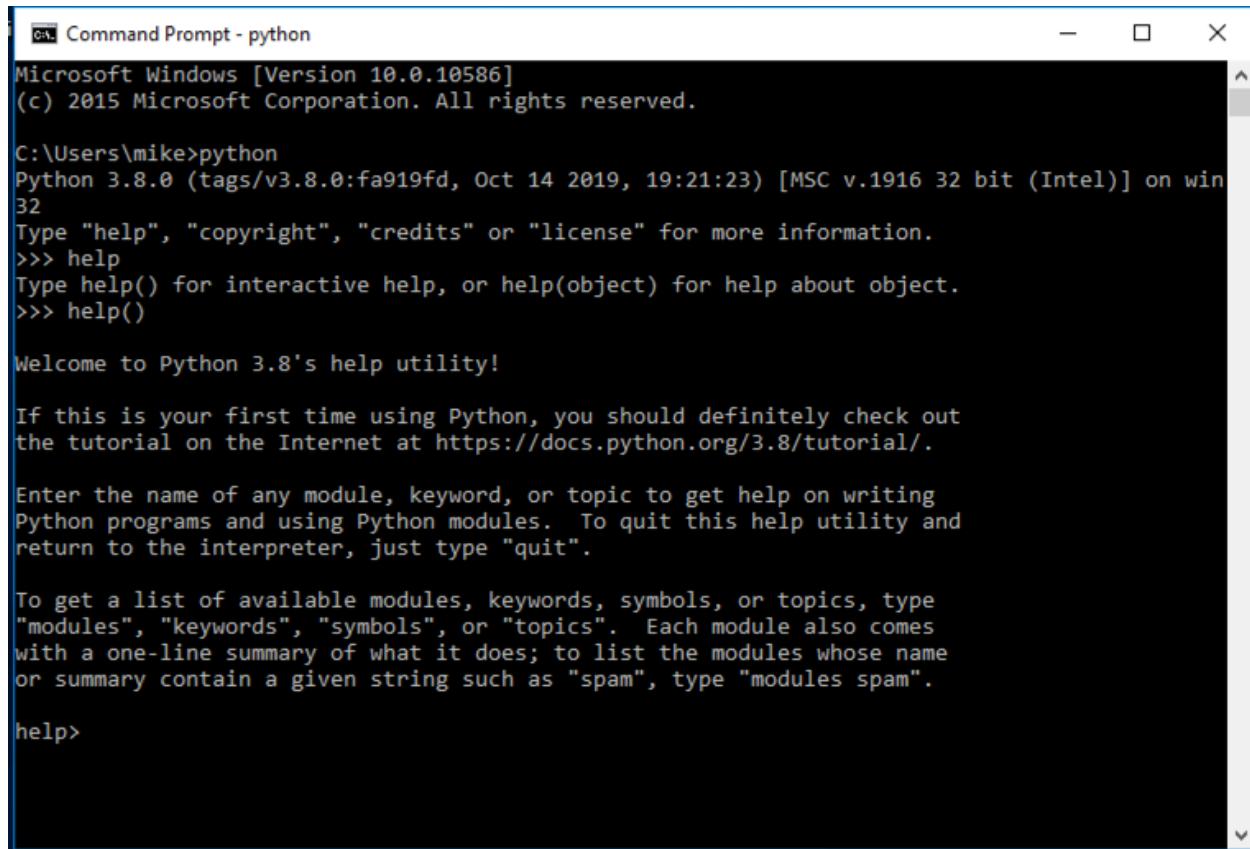
```
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Fig. 2-2: Running Python in cmd.exe

If this doesn't work and you get an "Unrecognized Command" or some other error, then Python may not be installed or configured correctly. On Windows, you may need to add Python to your system's path or you can just type out the full path to Python in your command console. For example, if you installed Python in C:\Python\Python38, then you can run it using cmd.exe like you did above, but instead of typing python, you would type C:\Python\Python38\python.

If you need to get help in the REPL, you can type `help()`:



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\mike>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> help()

Welcome to Python 3.8's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.8/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

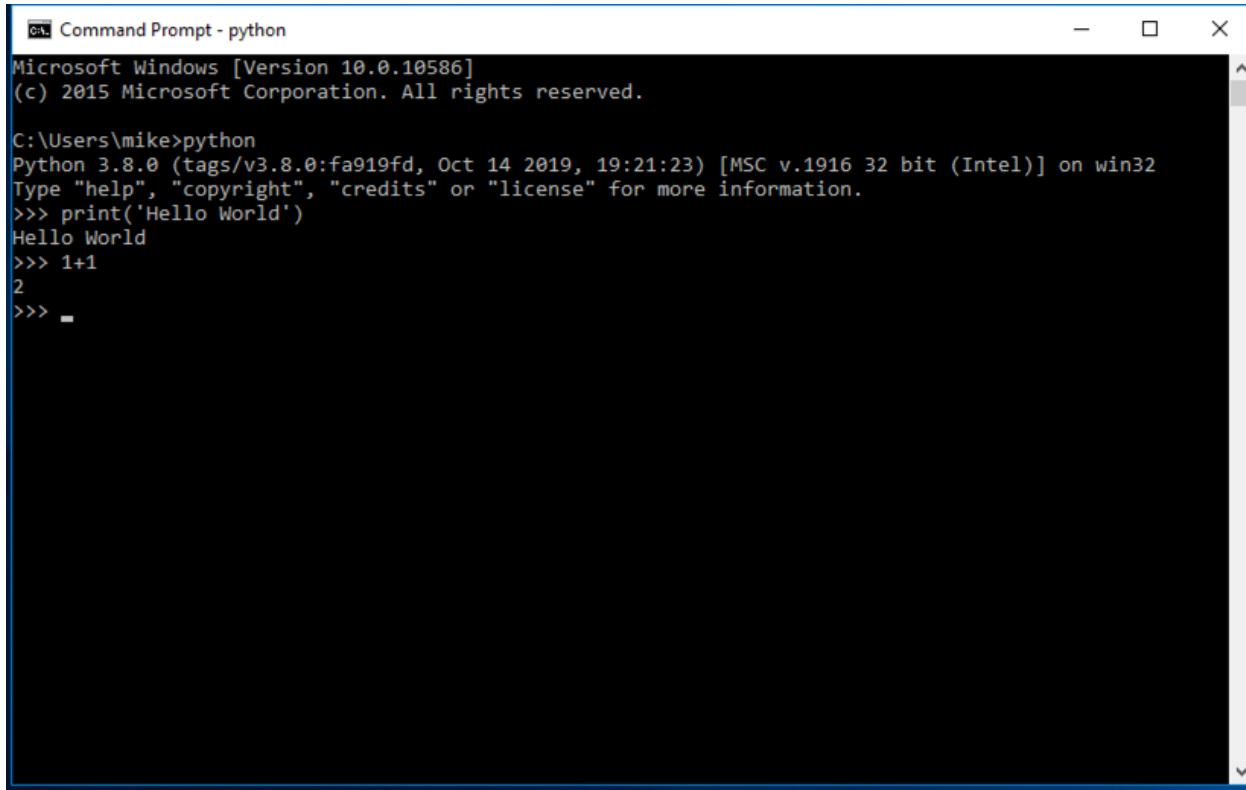
To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Fig. 2-3: REPL Help

You can type live Python code into the REPL and it will be immediately evaluated, which means the code will run as soon as you press enter.

Here's how you would print out "Hello World" and add some numbers in the REPL:



The screenshot shows a Windows Command Prompt window titled "Command Prompt - python". The window displays the following text:

```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\mike>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World')
Hello World
>>> 1+1
2
>>> -
```

Fig. 2-4: REPL Example Code

Python comes with its own code editor called IDLE. Let's learn about that next!

Getting Started with IDLE

IDLE is a good place to start learning Python. Once you have it installed, you can start it up and the initial screen will look like this:

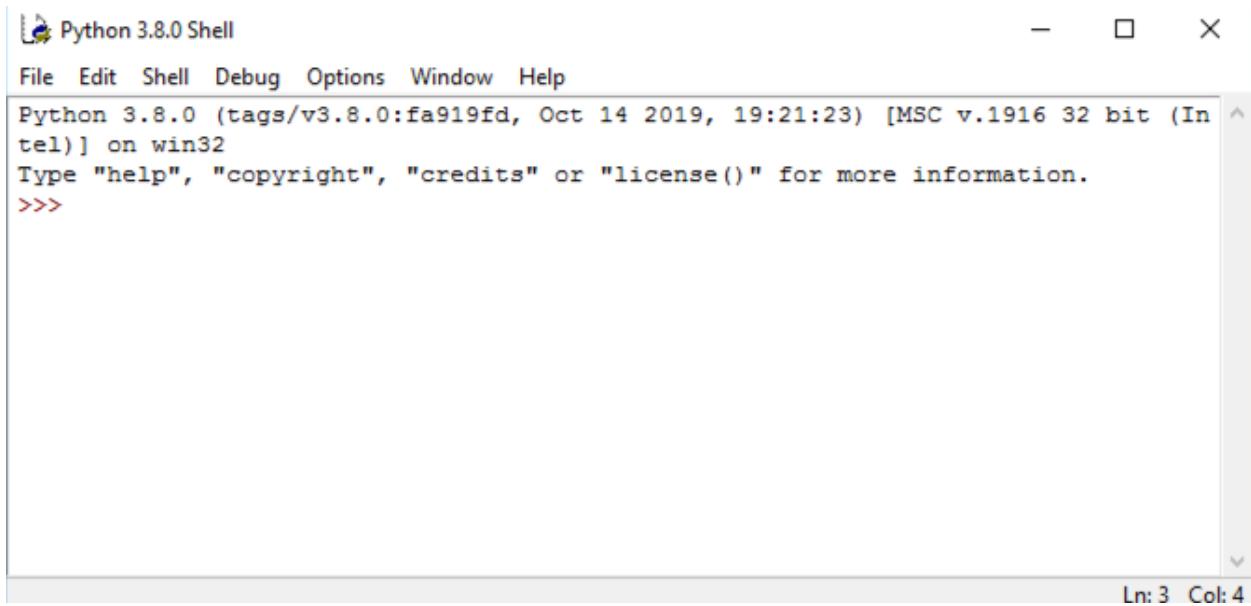


Fig. 2-5: The IDLE Shell

This is a REPL. You can enter code here and it will be evaluated as soon as you press the Return or Enter key.

If you want to actually write a full program, then you will need to open up the editor view by going to **File** → **New**.

You should now have the following dialog on your screen:

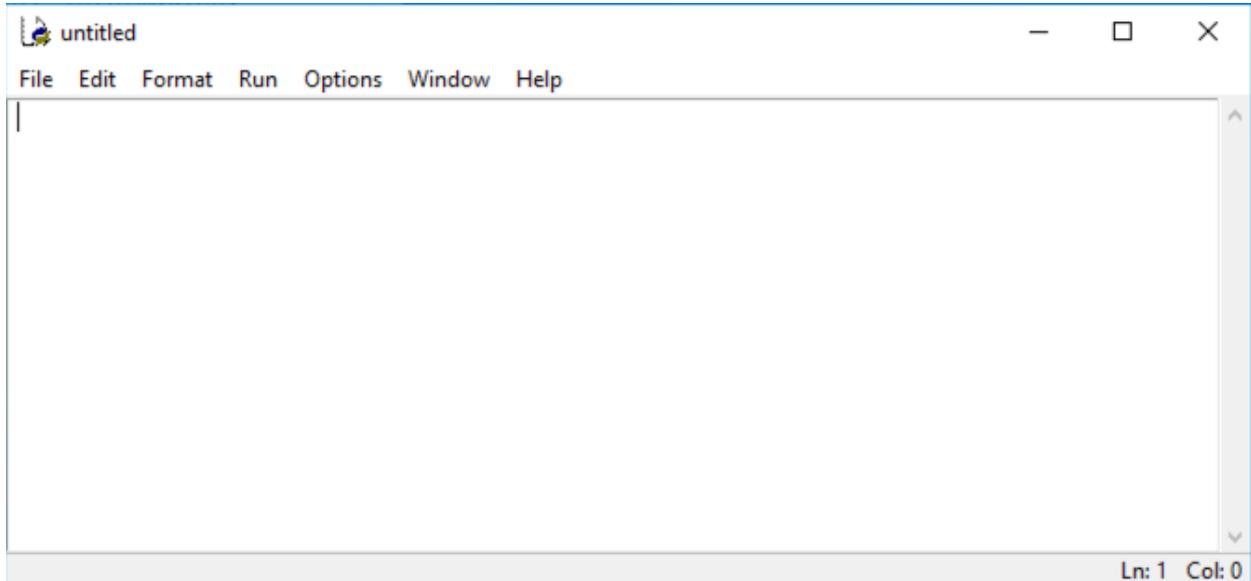


Fig. 2-6: The IDLE Editor

You can enter your code here and save it.

Running Your Code

Let's write a small bit of code in our code editor and then run it. Enter the following code and then save the file by going to **File** → **Save**.

```
1 print('Hello World')
```

To run this code in IDLE, go to the **Run** menu and choose the first option labeled **Run Module**:

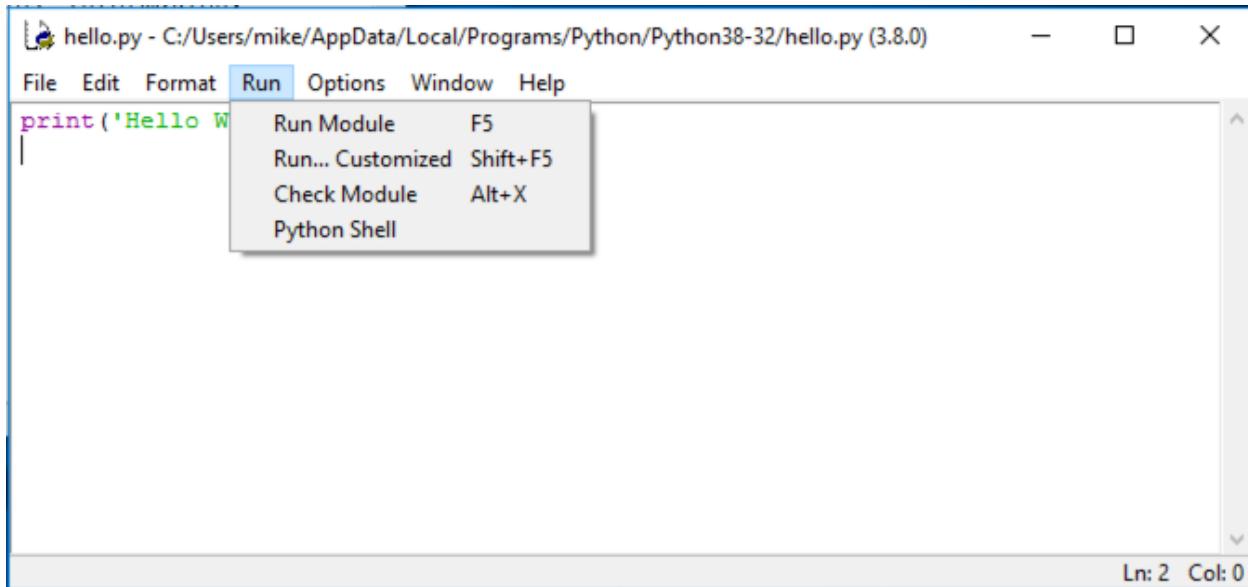
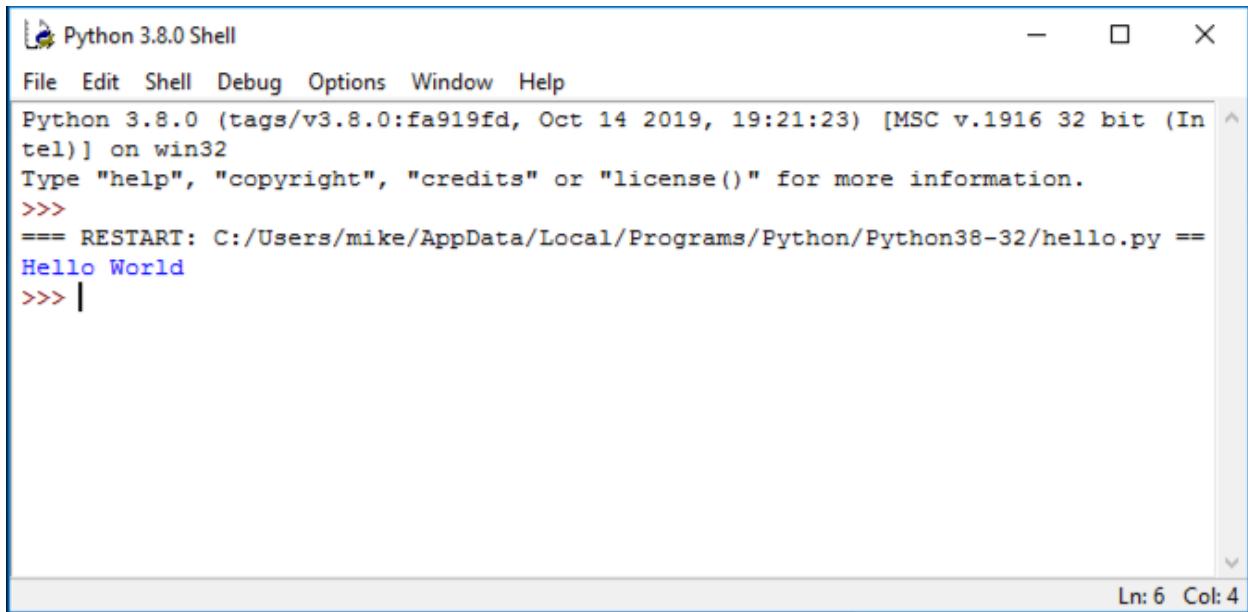


Fig. 2-7: Running Code in IDLE

When you do this, IDLE will switch to the Shell and show you the output of your program, if there is any:



The screenshot shows the Python 3.8.0 Shell window. The title bar reads "Python 3.8.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (In tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:/Users/mike/AppData/Local/Programs/Python/Python38-32/hello.py ==
Hello World
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 6 Col: 4".

Fig. 2-8: Output of Code in IDLE

You can also use the Run menu's Check Module option to check your code for syntax errors.

Accessing Help / Documentation

Sometimes you need help. Fortunately IDLE has some built-in help about itself and the Python language, too! You can access help about IDLE by going to the Help menu and choosing IDLE Help:

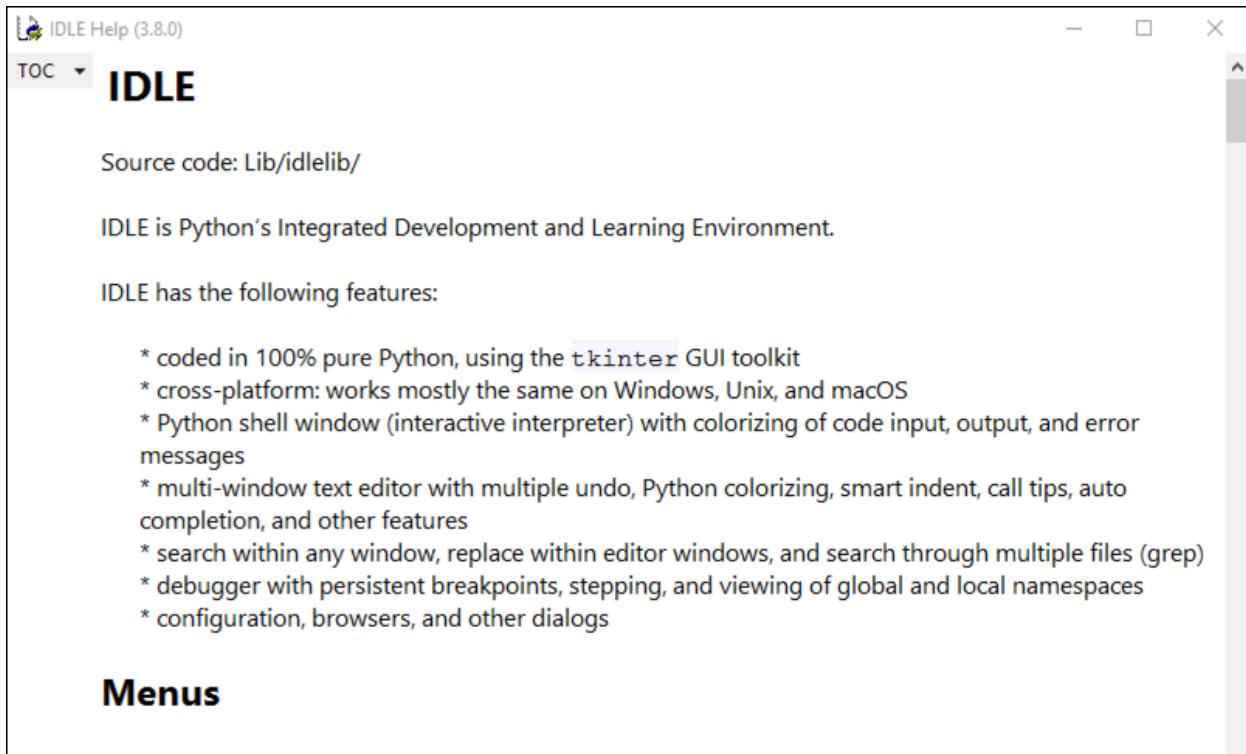


Fig. 2-9: IDLE Help

If you'd rather look up how something works in the Python language, then go to the **Help** menu and choose **Python Docs** or press F1 on your keyboard:

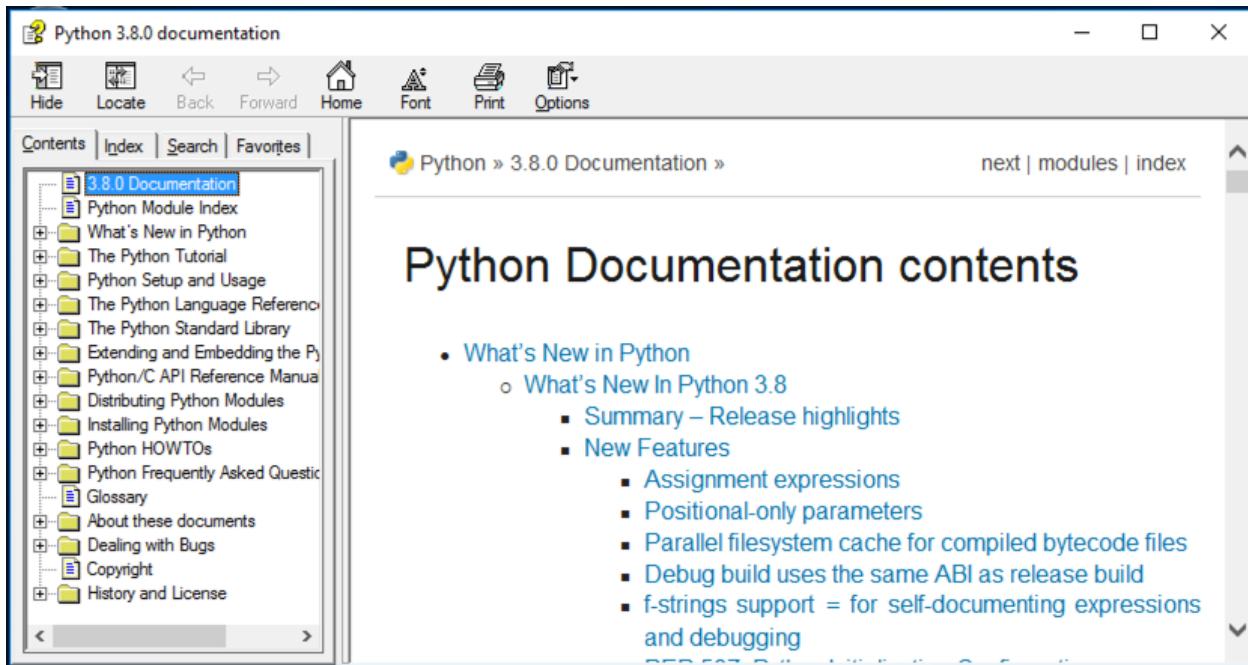


Fig. 2-10: IDLE Python Documentation

This will show you Python's official documentation. Depending on your O/S this may load local help files, or start a browser to show the official on-line help documents.

Restarting the Shell

Let's go back to the Shell screen of IDLE rather than the editor. It has several other functions that are worth going over. The first is that you can restart the shell.

Restarting the shell is useful when you need to start over with a clean slate but don't want to close and reopen the program. To restart the shell, go to the **Shell** menu and choose **Restart Shell**:

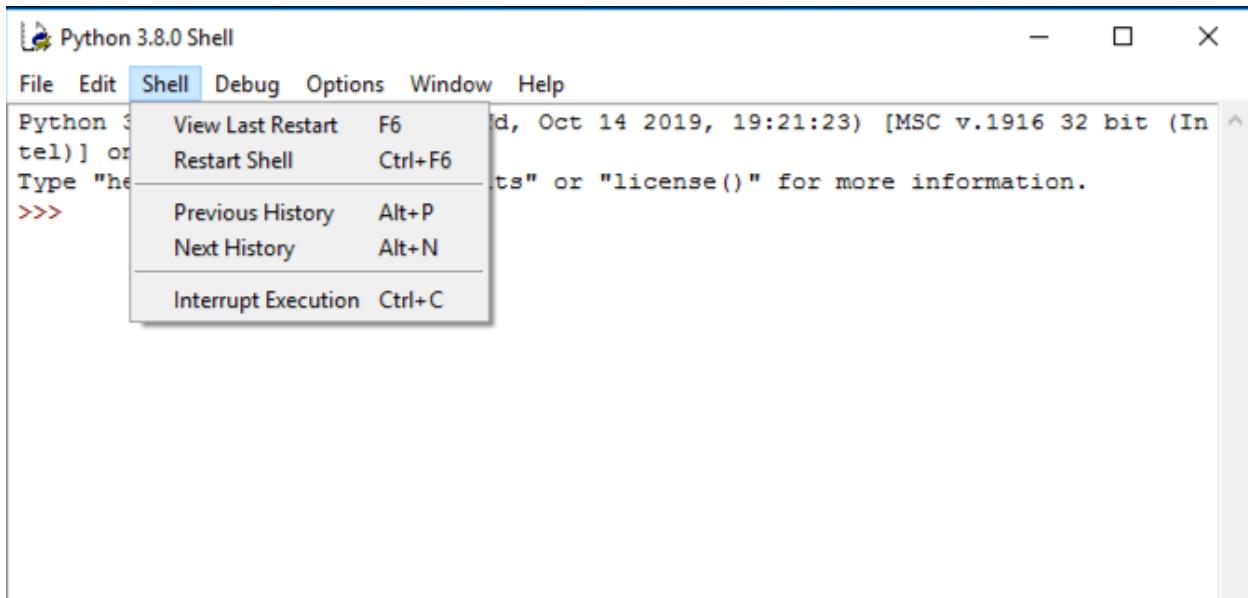


Fig. 2-11: IDLE Restart Menu

If you haven't restarted the shell before, then your screen will look like this:

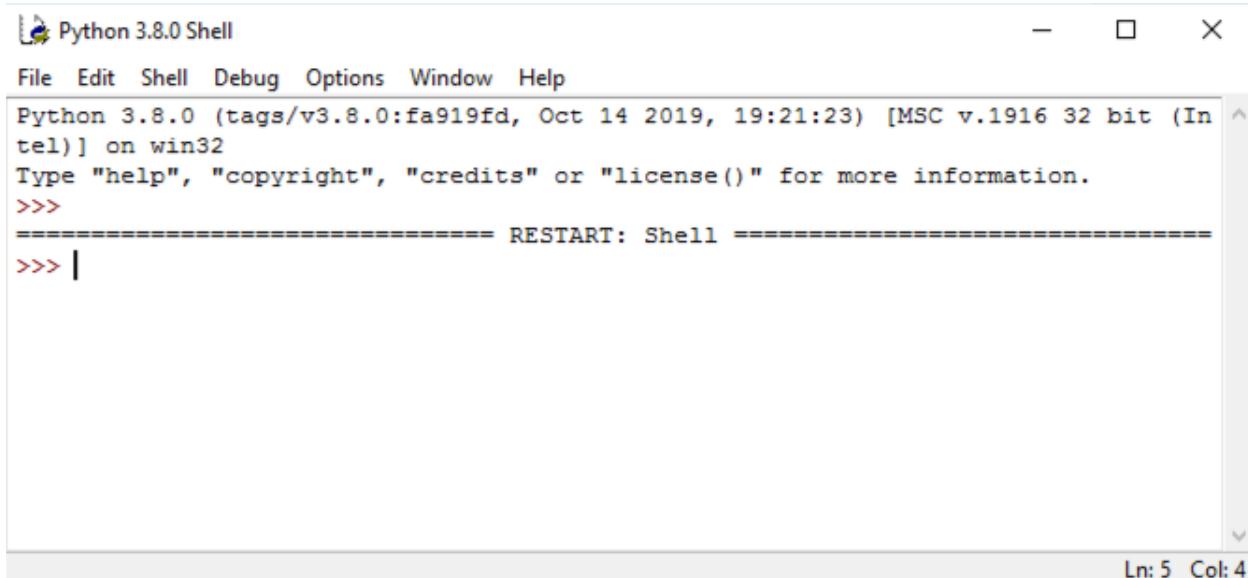


Fig. 2-12: IDLE Restarted

This tells you that your shell has restarted.

Module Browser

IDLE comes with a handy tool called the **Module Browser**. This tool can be found in the **File** menu. When you open it, you will see the following:

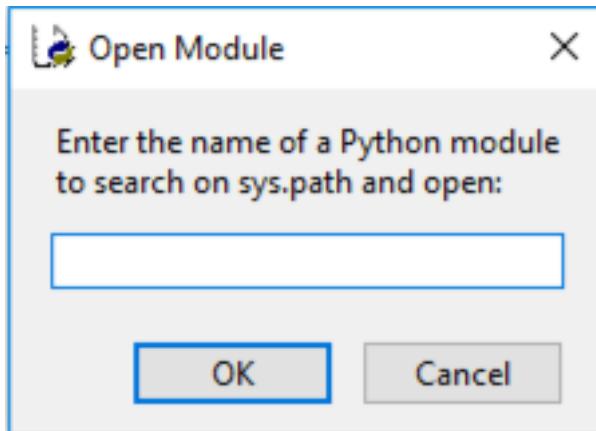


Fig. 2-13: IDLE Opening Module Browser

Modules in Python are code that the Python core development team has created for you. You can use the **Module Browser** to browse the source code of Python itself.

Try entering the following into the dialog above: `os`. Then press OK.

You should now see the following:

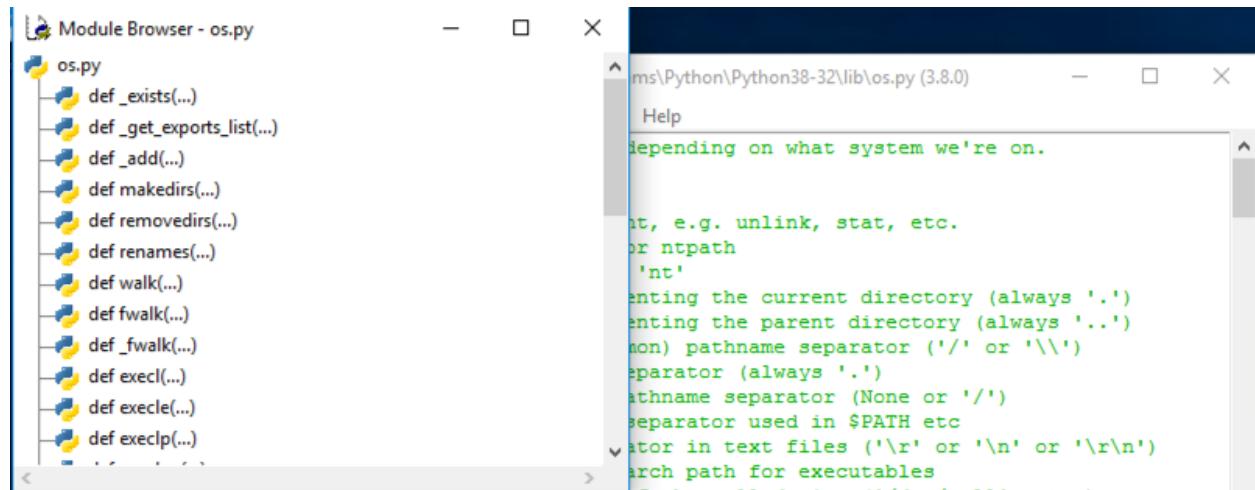


Fig. 2-14: IDLE Opening Module Browser

This allows you to browse the source code for `os.py`. You can double-click anything in the **Module Browser** and it will jump to the beginning of where that code is defined in IDLE's code editor.

Path Browser

Another useful tool that you can use in IDLE is the **Path Browser**. The **Path Browser** allows you to see where Python is installed and also what paths Python uses to import modules from. You will learn more about importing and modules later on in this book.

You can open it by going to **File** and then **Path Browser**:

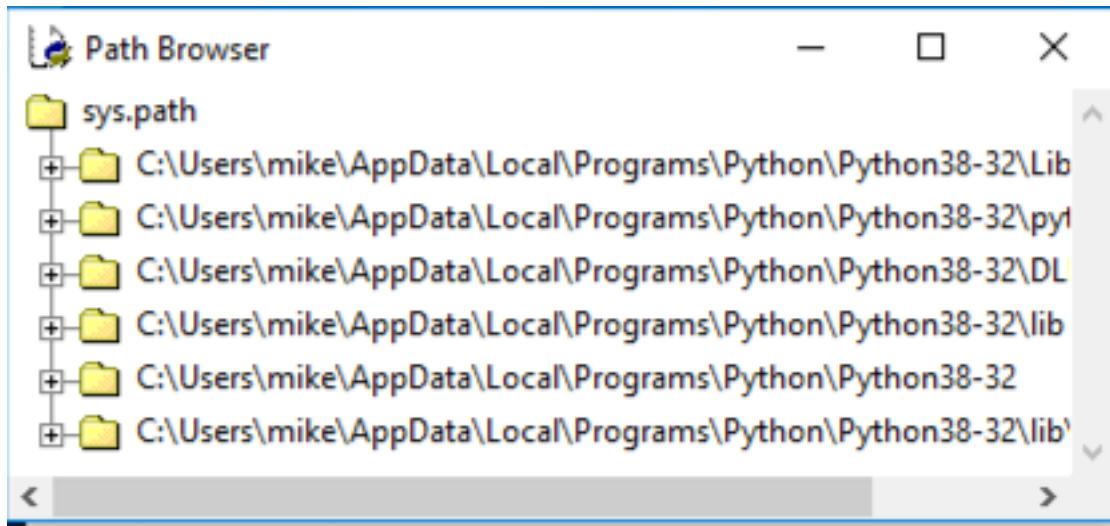


Fig. 2-15: IDLE Path Browser

The **Path Browser** is a good way to diagnose issues with importing modules. It can show you that you might not have Python configured correctly. Or it might show you that you have installed a 3rd party module in the wrong location.

Getting Started with PyCharm Community Edition

PyCharm is a commercial Python IDE from a company called JetBrains. They have a professional version, which costs money, and a community edition, which is free. PyCharm is one of the most popular choices for creating and editing Python programs.

PyCharm Professional has tons of features and a great debugger. However, if you are a beginner, you may find all the functionality in this software to be a bit overwhelming.

To get a copy of PyCharm Community Edition, you can go to the following website:

<https://www.jetbrains.com/pycharm/>

The Community Edition does not have all the features that PyCharm Professional has. But that is okay when you are new to Python. If you would like to try PyCharm, go ahead and download and install the software.

When you run PyCharm it may ask you to import settings. You can ignore that or import settings if you have used PyCharm previously and already have some.

Next, you will probably need to accept their privacy policy / EULA. Depending on the operating system, you may also get asked what theme to apply. The default is Darkula on Windows.

At this point you should see the following Welcome banner:



Fig. 2-16: PyCharm Welcome

PyCharm prefers that you work in a project rather than opening a simple file. Projects are typically collections of related files or scripts. You can set up a new project here or open a pre-existing one.

Once you have gone through that process, your screen should look like this:

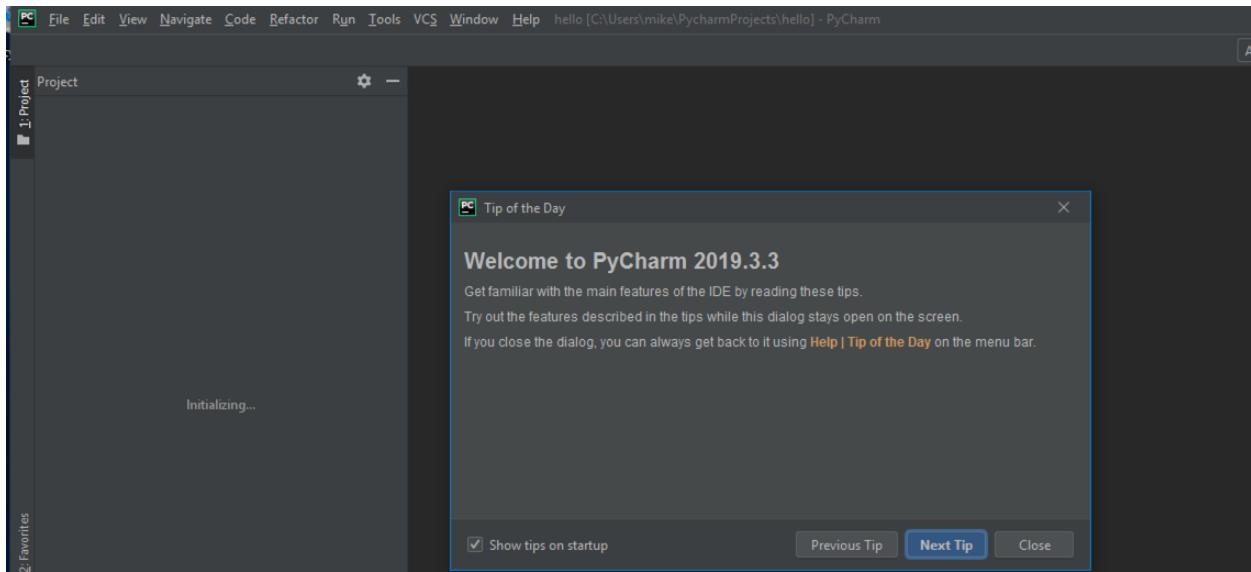


Fig. 2-17: PyCharm Project

Creating a Python Script

To create a new Python script in PyCharm, you can go to **File** and choose **New**. Then pick **Python File** from the choices presented:

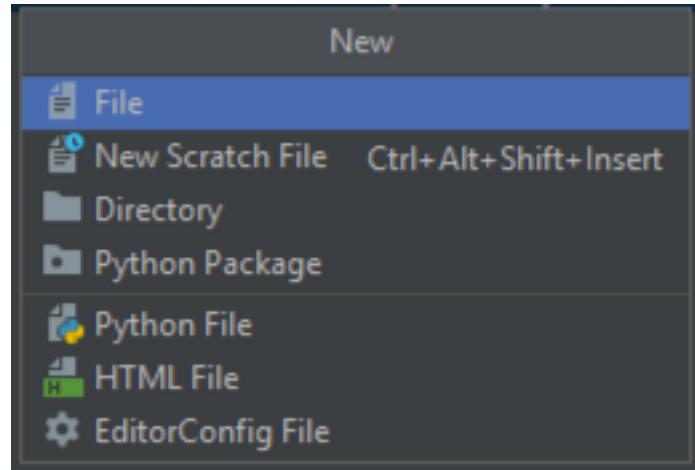


Fig. 2-18: PyCharm New

Give the file a name, such as **hello.py**. Now PyCharm should look like this:

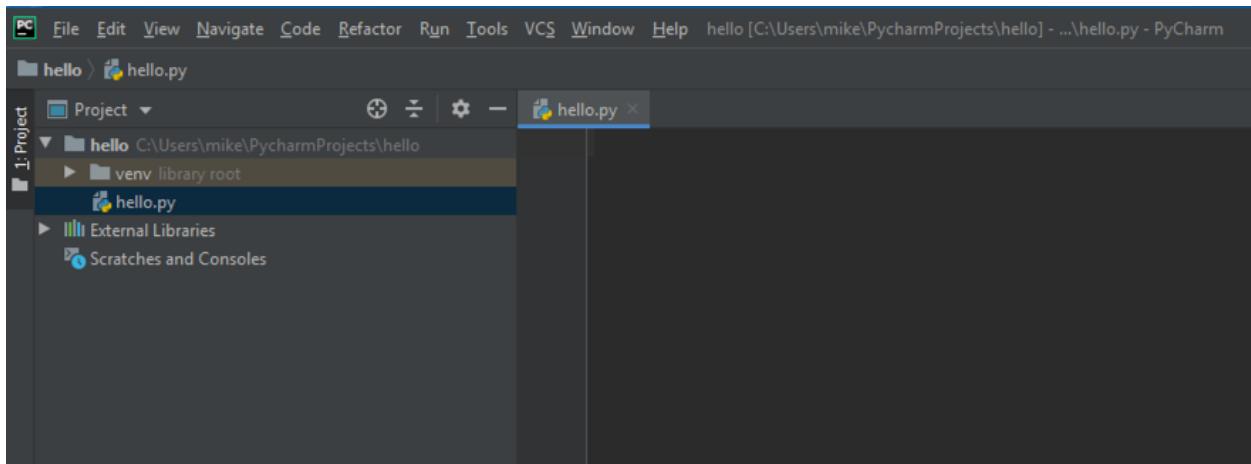


Fig. 2-19: PyCharm Hello World

Running Code in PyCharm

Let's add some code to your file:

```
1 print('Hello PyCharm')
```

To run your code, go to the **Run** menu and choose **Run**. PyCharm might ask you to set up a debug configuration before running it. You can save the defaults and continue.

You should now see the following at the bottom of PyCharm:

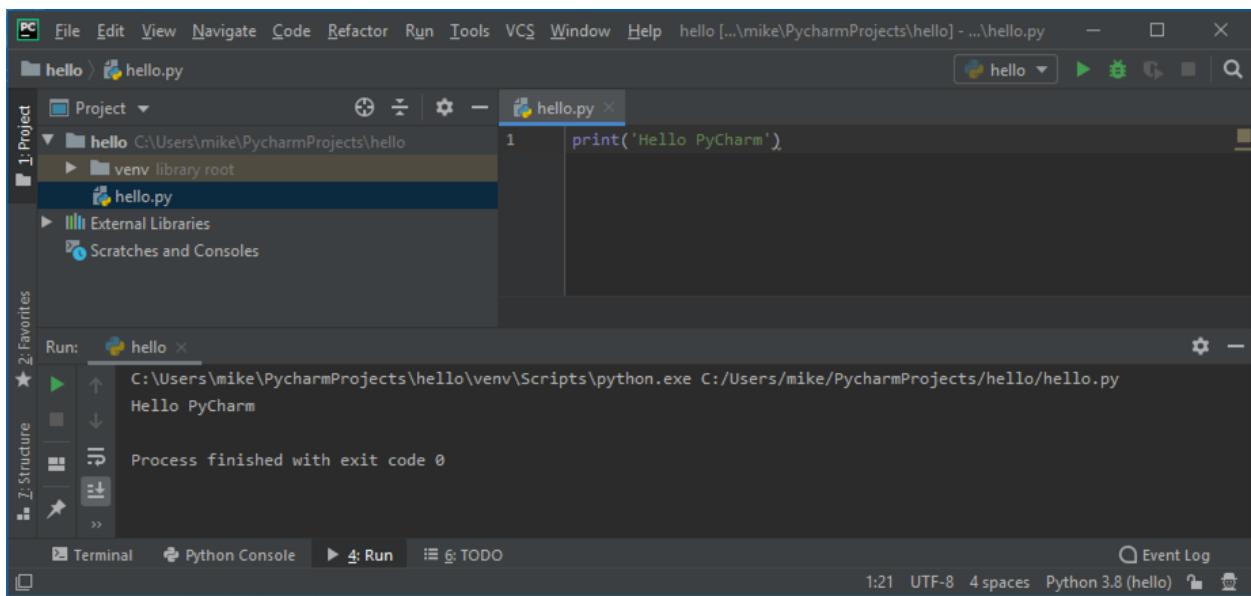


Fig. 2-20: PyCharm Running Code Output

PyCharm Features

PyCharm has tons of features. In fact, it has so many that you could write an entire book on them. For the purposes of this book, you should know that PyCharm will give you suggestions about your code based on PEP8, which is Python's code style guide. You will learn more about that in the next chapter. It will also highlight many other things about your code.

You can usually hover over any code that looks weird to you and a tooltip will appear that will explain the issue or warning.

The debugger that ships with PyCharm is useful for figuring out why your code doesn't work. You can use it to walk through your code line-by-line.

PyCharm's documentation is quite good, so if you get stuck, check their documentation.

Getting Started with Wing Personal

Wingware's Python IDE is written in Python and PyQt. It is my personal favorite IDE for Python. You can get it in Professional (paid), Personal (free) or 101 (really stripped-down version, but also free). Their website explains the differences between the 3 versions.

You can get Wingware here:

<https://wingware.com/>

After you have downloaded and installed the software, go ahead and run it. You will need to accept the License Agreement to load up the IDE.

Once it is fully loaded, you will see something like this:

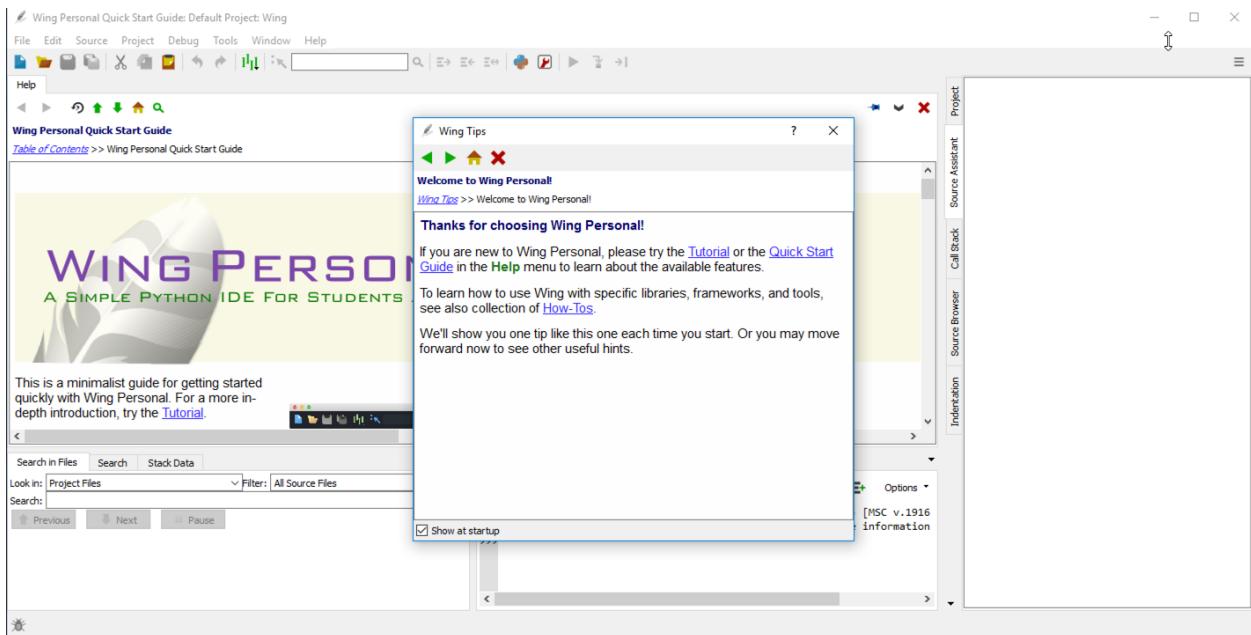


Fig. 2-21: Wingware Python IDE Main Screen

Running Code in Wingware

Let's create some code in Wing. You can open a new file by going to the **File** menu and choosing **New**:

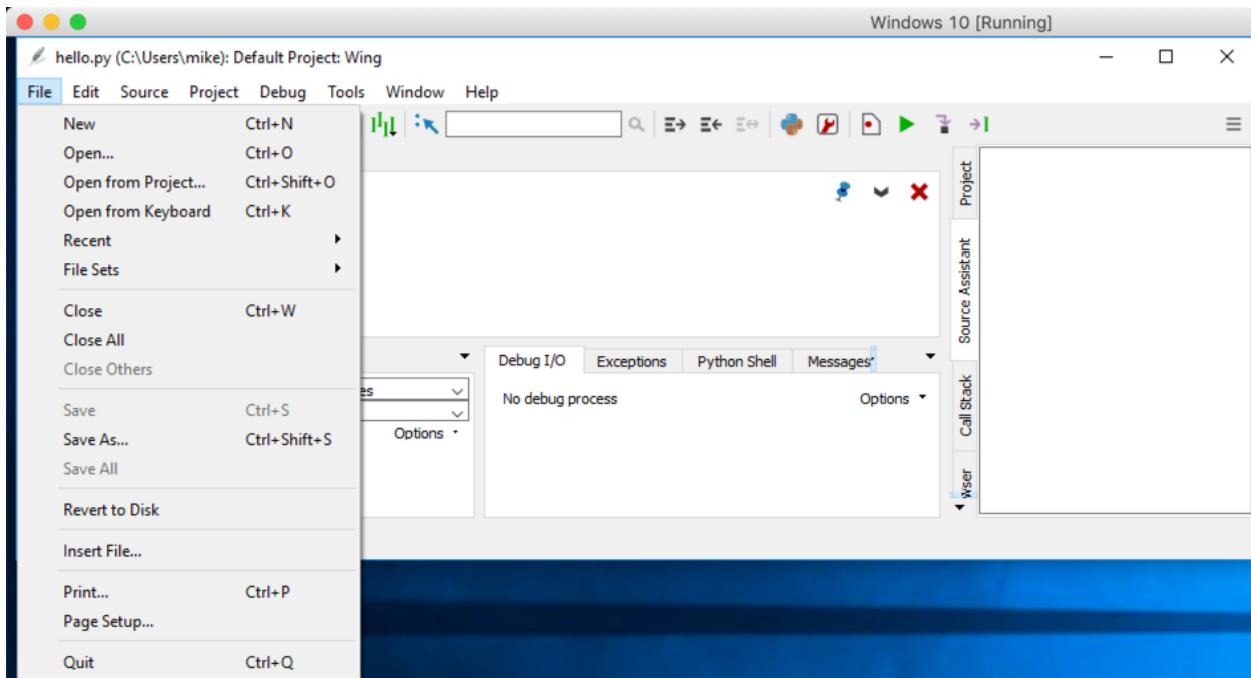


Fig. 2-22: Wingware Python IDE - Adding Code

Now enter the following code:

```
1 print('Hello Wingware')
```

Save the code to disk by going to **File** and then **Save**.

To run this code, you can go to the **Debug** menu, press F5 or click the green “play” button in the toolbar. You will see a debug message dialog:

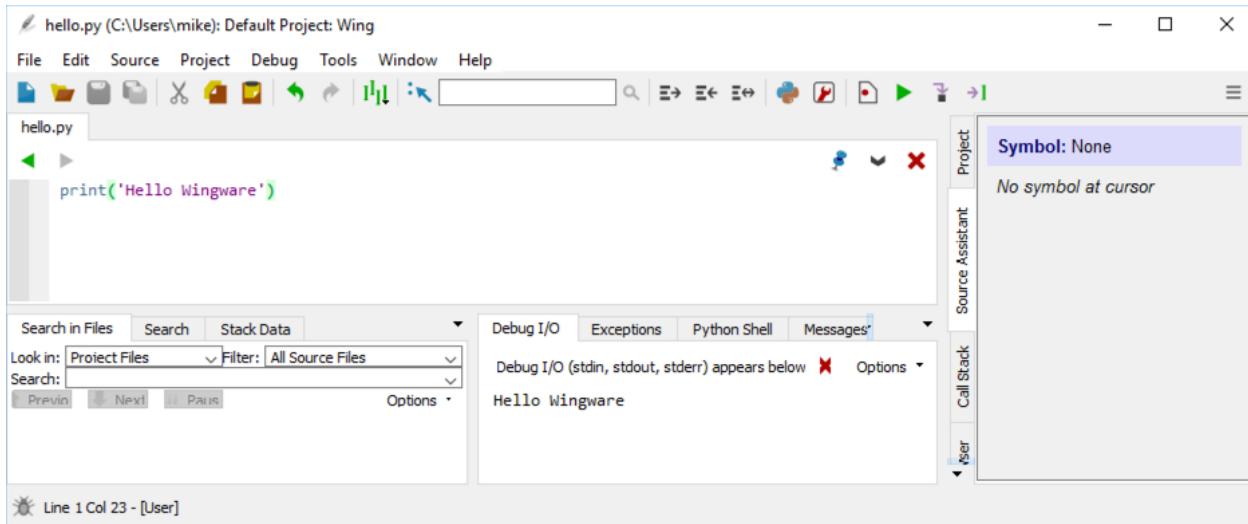


Fig. 2-23: Wingware Python IDE - Code Output

Hit OK and the code will run. You will see the output in the **Debug I/O** tab if there is any.

Note that Wing does not require you to create a project to run a single Python file. You can create projects if you want to though.

Wing Features

Wing has an incredible debugger. However, you cannot use it to its full extent in the free versions of the software. But there is a **Source Assistant** tab in the Personal edition that is very useful. It will show you information about the functions / modules that you have loaded as you use them. This makes learning new modules much easier.

Wing will also show you various issues with your code while you type, although PyCharm seems to do more in this area than Wing does.

Both products have plugins and you can write your own for both IDEs as well.

Getting Started with Visual Studio Code

Visual Studio Code, or VS Code for short, is a general-purpose programming editor. Unlike PyCharm and WingIDE, it is designed to work with lots of languages. PyCharm and WingIDE will let you

write in other languages too, but their primary focus is on Python.

VS Code is made by Microsoft and it is free. You can download it here:

<https://code.visualstudio.com/>

Once you have it downloaded and installed, you will need to install support for Python from the VS Code marketplace.

If you open up VS Code, the screen will look something like this:

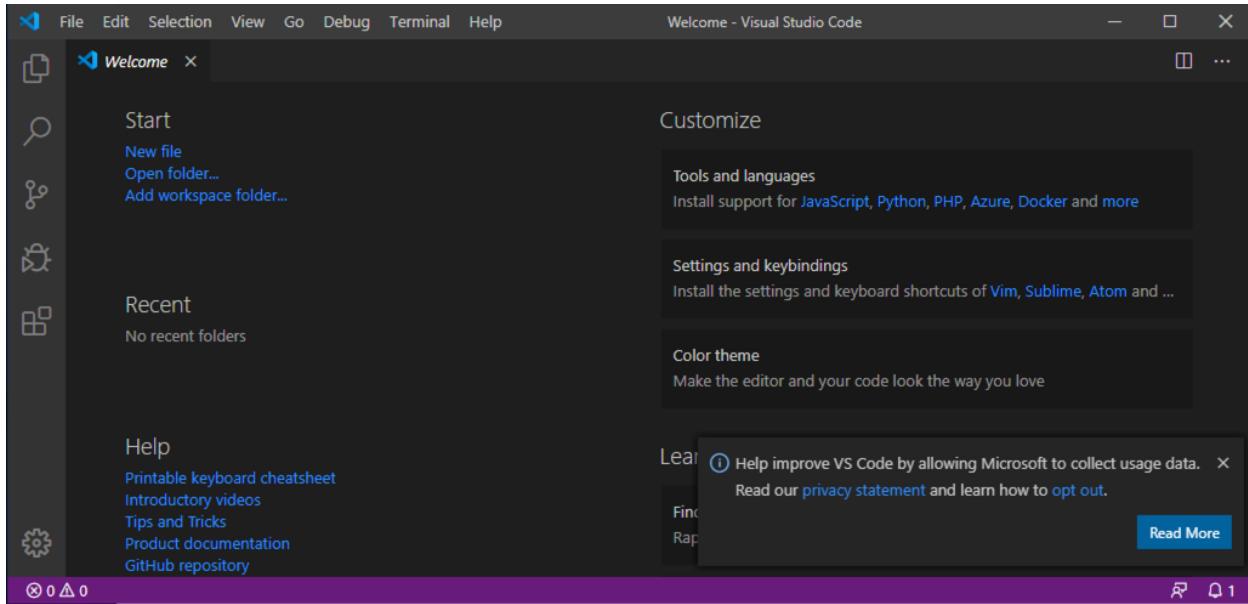


Fig. 2-24: VS Code - Main Screen

Under Customize you can see there is an option for installing Python. If that isn't there, you can click on the **Extensions** button that is on the left and search for Python there:

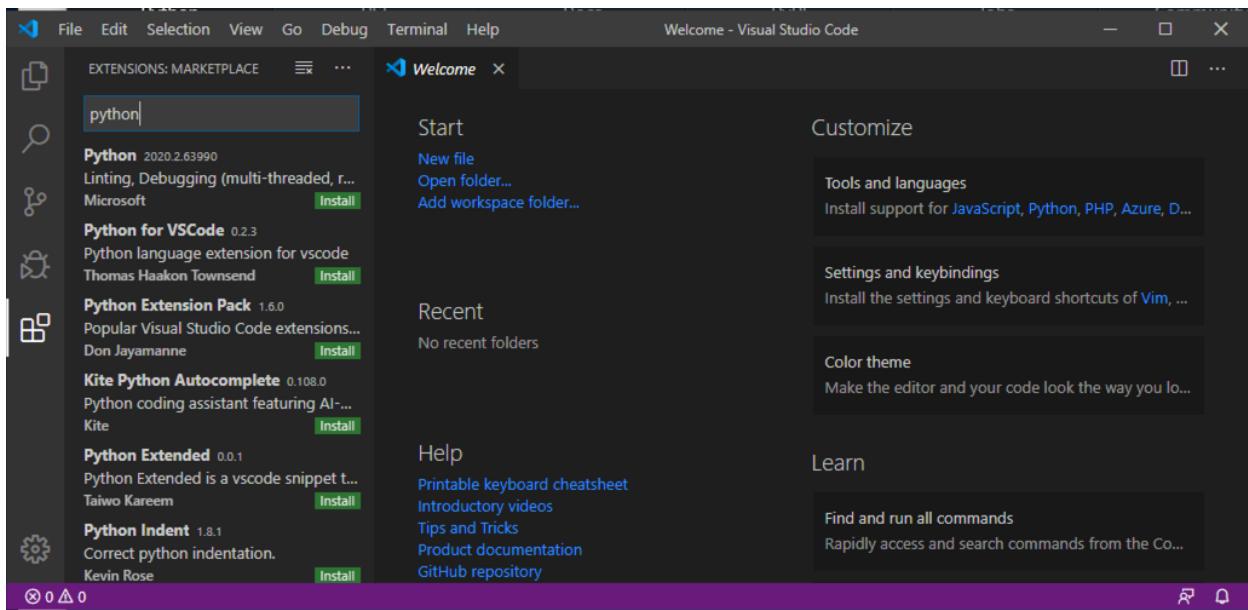


Fig. 2-25: VS Code - Adding the Python Extension

Go ahead and install the Python extension so that VS Code will recognize Python correctly.

Running Code in VS Code

Open a folder in the **File Explorer** tab and then you can right-click in there to create a new file. Alternatively, you can go to the **File** menu and choose **New File** and do it that way.

Once that is done, you can enter the following code and save it:

```
1 print('Hello VS Code')
```

Then right-click anywhere in the editor and select the **Run Python File in Terminal** selection. This will cause your code to run and you will see the following:

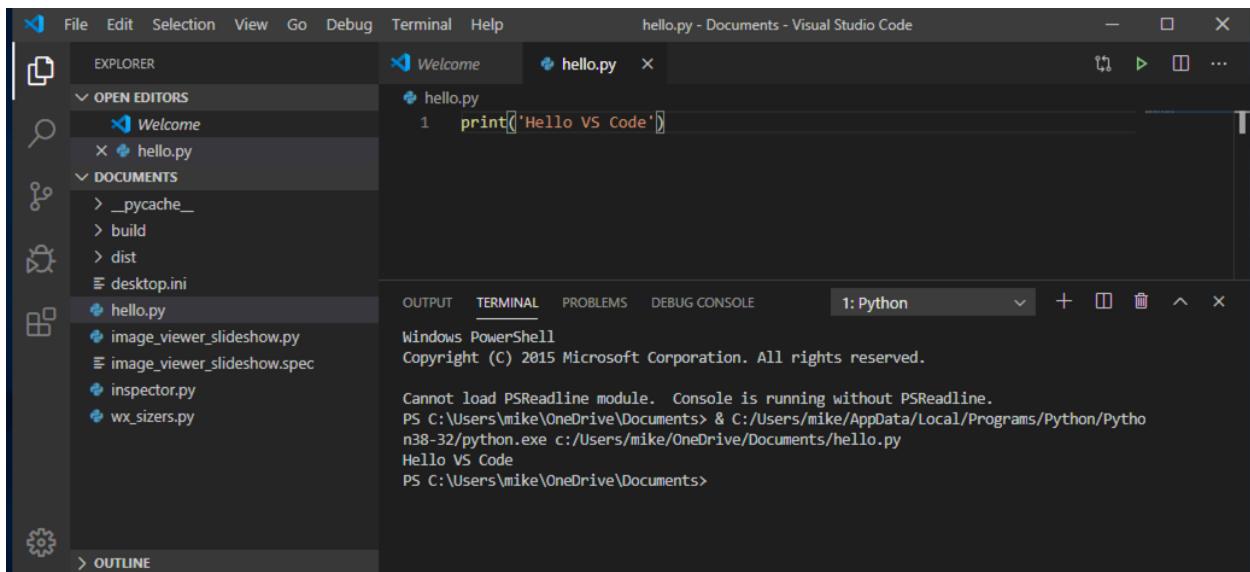


Fig. 2-26: VS Code - Running Code

Note: I didn't have the `PSReadline` module installed when I ran this code which is why you see the error in the console above.

VS Code Features

VS Code can run all kinds of different languages. However, for the purposes of Python, Microsoft has a team of Python developers that are constantly improving this IDE and adding new features. There are tons of extensions that you can install to enhance the editor's functionality.

One of the coolest extensions that you can install is **Live Share**, which lets you do real-time collaboration between developers. It basically shares your coding session with others. Since this IDE is the newest of the bunch and its feature set is changing a lot, you will need to research it on your own time.

Wrapping Up

There are lots of Python code editors to choose from. IDLE is nice in that it comes with Python and is written in Python, so you can actually learn a lot just by looking at its source code. PyCharm and VS Code are very popular right now. Wing IDE used to be more popular than it is today, but I think it is still really great. All of these tools are good, but you should give them a try to see which one works the best for you.

Chapter 3 - Documenting Your Code

Documenting your code early on is quite a bit more important than most new developers realize. Documentation in software development refers to the idea of giving your variables, functions and other identifiers descriptive names. It also refers to adding good comments. When you are immersed in developing your latest creation, it is easy to create variables and functions with non-descriptive names. A month or a year later, when you inevitably come back to your code, you will spend an inordinate amount of time trying to figure out what your code does.

By making your code self-documenting (i.e. using descriptive names) and adding comments when necessary, you will make your code more readable for yourself and for anyone else who may use your code. This will make updating your code and refactoring your code easier too!

In this chapter you will learn about the following topics:

- Comments
- Docstrings
- PEP8 - The Python Style Guide
- Other Tools Useful for Documenting Your Code

Let's get started by learning about comments.

What are Comments?

Comments are code that is for you, not for your computer. What I mean by that is that a comment is basically a note to yourself that explains what is happening in that portion of your code. You use comments to explain why you did something or how a piece of code works. When you are starting out as a new developer, it is good to leave yourself lots of comments to refer back to. But once you learn how to properly name your functions and variables, you will find that you don't need comments as much.

However, comments are still recommended, especially for code that is complex and not easy to understand at first glance. Depending on the company you work for, you may also use comments to document bug fixes. For example, if you are fixing a bug, you might include a comment that mentions which bug you are fixing to help explain why you had to change it.

You can create comments by using the `#` sign followed by some descriptive text.

Here is an example

```
1 # This is a bad comment  
2 x = 10
```

In the code above, the first line demonstrates how to create a simple comment. When Python goes to execute this code, it will see the `#` symbol and ignore all the text that follows it. In effect, Python will skip that line and try to execute the second line.

This comment is marked as a “bad comment”. While it is good for demonstration purposes, it does not describe the code that follows it at all. That is why it is not a good comment. Good comments describe the code that follows. A good comment may describe the purpose for the Python script, the code line or something else. Comments are your code’s documentation. If they don’t provide information, then they should be removed.

You can also create in-line comments:

```
1 x = 10 # 10 is being assigned to x
```

Here you once again assign 10 to the variable `x`, but then you add two spaces and the `#` symbol, which allows you to add a comment about the code. This is useful for when you might need to explain a specific line of code. If you named your variable something descriptive, then you most likely won’t need a comment at all.

Commenting Out

You will hear the term “commenting out code” fairly often. This is the practice of adding the `#` symbol to the beginning of your code. This will effectively disable your code.

For example, you might have this line of code:

```
1 number_of_people = 10
```

If you want to comment it out, you can do the following:

```
1 # number_of_people = 10
```

You comment code out when you are trying out different solutions or when you’re debugging your code, but you don’t want to delete the code. Python will ignore code that is commented out, allowing you to try something else. Most Python code editors (and text editors) provide a way to highlight multiple lines of code and comment out or uncomment out the entire block of code.

Multiline Comments

Some programming languages, such as C++, provide the ability to create multi-line comments. The Python style guide (PEP8) says that the pound sign is preferred. However, you can use triple quoted strings as a multiline comment.

Here's an example:

```
1 >>> '''This is a
2     multiline comment'''
3 >>> """This is also a
4     multiline comment"""
```

When you create triple quoted strings you may be creating a **docstring**.

Let's find out what docstrings are and how you can use them!

Learning About docstrings

Python has the concept of the PEP, or Python Enhancement Proposal. These PEPs are suggestions or new features for the Python language that get discussed and agreed upon by the Python Steering Council.

PEP 257 (<https://www.python.org/dev/peps/pep-0257/>) describes docstring conventions. You can go read that if you'd like the full story. Suffice to say, a docstring is a string literal that should occur as the first statement in a module, function, class or method definition. You don't need to understand all these terms right now. In fact, you'll learn more about them later on in this book.

A docstring is created by using triple double-quotes.

Here is an example:

```
1 """
2 This is a docstring
3 with multiple lines
4 """
```

Docstrings are ignored by Python. They cannot be executed. However, when you use a docstring as the first statement of a module, function, etc, the docstring will become a special attribute that can be accessed via `__doc__`. You will learn more about attributes and docstrings in the chapter about classes.

Docstrings may be used for one-liners or for multi-line strings.

Here is an example of a one-liner:

```
1 """This is a one-liner"""
```

A one-liner docstring is simply a docstring with only one line of text.

Here is an example of a docstring used in a function:

```
1 def my_function():
2     """This is the function's docstring"""
3     pass
```

The code above shows how you can add a docstring to a function. You can learn more about functions in chapter 14. A good docstring describes what the function is supposed to accomplish.

Note: While triple double-quotes are the recommended standard, triple single-quotes, single double-quotes, and single single-quotes all work as well (but single double- and single single-quotes can only contain one line, not multiple lines).

Now let's learn about coding according to Python's style guide.

Python's Style Guide: PEP8

A style guide is a document that describes good programming practices, usually with regard to a single language. Some companies have specific style guides for the company that developers must follow no matter what programming language they are using.

Back in 2001, the Python style guide was created as PEP8 (<https://www.python.org/dev/peps/pep-0008/>). It documents coding conventions for the Python programming language and has been updated several times over the years.

If you plan to use Python a lot, you should really check out the guide. It will help you write better Python code.

Also if you want to contribute to the Python language itself, all your code must conform to the style guidelines or your code will be rejected.

Following a style guide will make your code easier to read and understand. This will help you and anyone else who uses your code in the future.

Remembering all the rules can be hard, though. Fortunately, some intrepid developers have created some utilities that can help!

Tools that can help

There are lots of neat tools that you can use to help you write great code. Here are just a few:

- pycodestyle - <https://pypi.org/project/pycodestyle/> - Checks if your code follows PEP8
- Pylint - <https://www.pylint.org/> - An in-depth static code testing tool that finds common issues with code
- PyFlakes - <https://pypi.org/project/pyflakes/> - Another static code testing tool for Python
- flake8 - <https://pypi.org/project/flake8/> - A wrapper around PyFlakes, pycodestyle and a McCabe script
- Black - <https://black.readthedocs.io/en/stable/> - A code formatter that mostly follows PEP8

You can run these tools against your code to help you find issues with your code. I have found Pylint and PyFlakes / flake8 to be the most useful. Black is helpful if you are working in a team and you want everyone's code to follow the same format. Black can be added to your toolchain to format your code for you.

The more advanced Python IDEs provide some of the checks that Pylint, etc. provide in real-time. For example, PyCharm will automatically check for a lot of the issues that these tools will find. WingIDE and VS Code provide some static code checking as well. You should check out the various IDEs and see which one works the best for you.

Wrapping Up

Python comes with several different ways to document your code. You can use **comments** to explain one or more lines of code. These should be used in moderation and where appropriate. You can also use **docstrings** to document your modules, functions, methods, and classes.

You should also check out Python's style guide that can be found in PEP8. This will help you develop good Python coding practices. There are several other style guides for Python. For example, you might want to look up Google's style guide or possibly NumPy's Python style guide. Sometimes looking at different style guides will help you develop good practices as well.

Finally, you learned about several tools you can use to help you make your code better. If you have the time, I encourage you to check out PyFlakes or Flake8 especially as they can be quite helpful in pointing out common coding issues in your code.

Review Questions

1. How do you create a comment?
2. What do you use a **docstring** for?
3. What is Python's style guide?
4. Why is documenting your code important?

Chapter 4 - Working with Strings

You will be using strings very often when you program. A string is a series of letters surrounded by single, double or triple quotes. Python 3 defines string as a “Text Sequence Type”. You can cast other types to a string using the built-in `str()` function.

In this chapter you will learn how to:

- Creating strings
- String methods
- String formatting
- String concatenation
- String slicing

Let's get started by learning the different ways to create strings!

Creating Strings

Here are some examples of creating strings:

```
1 name = 'Mike'  
2 first_name = 'Mike'  
3 last_name = "Driscoll"  
4 triple = """multi-line  
5 string"""
```

When you use triple quotes, you may use three double quotes at the beginning and end of the string or three single quotes. Also, note that using triple quotes allows you to create multi-line strings. Any whitespace within the string will also be included.

Here is an example of converting an integer to a string:

```
1 >>> number = 5  
2 >>> str(number)  
3 '5'
```

In Python, backslashes can be used to create escape sequences. Here are a couple of examples:

- `\b` - backspace

- \n - line feed
- \r - ASCII carriage return
- \t - tab

There are several others that you can learn about if you read Python's documentation.

You can also use backslashes to escape quotes:

```
1 >>> 'This string has a single quote, \', in the middle'
2 "This string has a single quote, ', in the middle"
```

If you did not have the backslash in the code above, you would receive a `SyntaxError`:

```
1 >>> 'This string has a single quote, ', in the middle'
2 Traceback (most recent call last):
3   Python Shell, prompt 59, line 1
4 invalid syntax: <string>, line 1, pos 38
```

This occurs because the string ends at that second single quote. It is usually better to mix double and single quotes to get around this issue:

```
1 >>> "This string has a single quote, ', in the middle"
2 "This string has a single quote, ', in the middle"
```

In this case, you create the string using double quotes and put a single quote inside of it. This is especially helpful when working with contractions, such as “don’t”, “can’t”, etc.

Now let's move along and see what methods you can use with strings!

String Methods

In Python, everything is an object. You will learn how useful this can be in chapter 18 when you learn about introspection. For now, just know that strings have methods (or functions) that you can call on them.

Here are three examples:

```

1 >>> name = 'mike'
2 >>> name.capitalize()
3 'Mike'
4 >>> name.upper()
5 'MIKE'
6 >>> 'MIke'.lower()
7 'mike'
```

The method names give you a clue as to what they do. For example, `.capitalize()` will change the first letter in the string to a capital letter.

To get a full listing of the methods and attributes that you can access, you can use Python's built-in `dir()` function:

```

1 >>> dir(name)
2 ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
3 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
4 '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
5 '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
6 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
7 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
8 'casfold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
9 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
10 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
11 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
12 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
13 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
14 'translate', 'upper', 'zfill']
```

The first third of the listing are special methods that are sometimes called "dunder methods" (AKA double-underscore methods) or "magic methods". You can ignore these for now as they are used more for intermediate and advanced use-cases. The items in the list above that don't have double-underscores at the beginning are the ones that you will probably use the most.

You will find that the `.strip()` and `.split()` methods are especially useful when parsing or manipulating text.

You can use `.strip()` and its variants, `.rstrip()` and `.lstrip()` to strip off white space from the string, including tab and new line characters. This is especially useful when you are reading in a text file that you need to parse.

In fact, you will often end up stripping end-of-line characters from strings and then using `.split()` on the result to parse out sub-strings.

Let's do a little exercise where you will learn how to parse out the 2nd word in a string.

To start, here's a string:

```
1 >>> my_string = 'This is a string of words'  
2 'This is a string of words'
```

Now to get the parts of a string, you can call `.split()`, like this:

```
1 >>> my_string.split()  
2 ['This', 'is', 'a', 'string', 'of', 'words']
```

The result is a list of strings. Now normally you would assign this result to a variable, but for demonstration purposes, you can skip that part.

Instead, since you now know that the result is a string, you can use list indexing to get the second element:

```
1 >>> 'This is a string of words'.split()[1]  
2 'is'
```

Remember, in Python, lists elements start at 0 (zero), so when you tell it you want element 1 (one), that is the second element in the list.

When doing string parsing for work, I personally have found that you can use the `.strip()` and `.split()` methods pretty effectively to get almost any data that you need. Occasionally you will find that you might also need to use Regular Expressions (regex), but most of the time these two methods are enough.

String Formatting

String formatting or string substitution is where you have a string that you would like to insert into another string. This is especially useful when you need to create a template, such as a form letter. But you will use string substitution a lot for debugging output, printing to standard out and much more.

Standard out (or stdout) is a term used for printing to the terminal. When you run your program from the terminal and you see output from your program, that is because your program “prints” to standard out or standard error (stderr).

Python has three different ways to accomplish string formatting:

- Using the % Method
- Using `.format()`
- Using formatted string literals (f-strings)

This book will focus on f-strings the most and also use `.format()` from time-to-time. But it is good to understand how all three work.

Let's take a few moments to learn more about string formatting.

Formatting Strings Using %s (printf-style)

Using the % method is Python's oldest method of string formatting. It is sometimes referred to as "printf-style string formatting". If you have used C or C++ in the past, then you may already be familiar with this type of string substitution. For brevity, you will learn the basics of using % here.

Note: This type of formatting can be quirky to work with and has been known to lead to common errors such as failing to display Python tuples and dictionaries correctly. Using either of the other two methods is preferred in that case.

The most common use of using the % sign is when you would use %s, which means convert any Python object to a string using str().

Here is an example:

```
1 >>> name = 'Mike'  
2 >>> print('My name is %s' % name)  
3 My name is Mike
```

In this code, you take the variable name and insert it into another string using the special %s syntax. To make it work, you need to use % outside of the string followed by the string or variable that you want to insert.

Here is a second example that shows that you can pass in an int into a string and have it automatically converted for you:

```
1 >>> age = 18  
2 >>> print('You must be at least %s to continue' % age)  
3 You must be at least 18 to continue
```

This sort of thing is especially useful when you need to convert an object but don't know what type it is.

You can also do string formatting with multiple variables. In fact, there are two ways to do this.

Here's the first one:

```
1 >>> name = 'Mike'  
2 >>> age = 18  
3 >>> print('Hello %s. You must be at least %i to continue!' % (name, age))  
4 Hello Mike. You must be at least 18 to continue!
```

In this example, you create two variables and use %s and %i. The %i indicates that you are going to pass an integer. To pass in multiple items, you use the percent sign followed by a tuple of the items to insert.

You can make this clearer by using names, like this:

```
1 >>> print('Hello %(first_name)s. You must be at least %(age)i to continue!'
2     % {'first_name': name, 'age': age})
3 Hello Mike. You must be at least 18 to continue!
```

When the argument on the right side of the % sign is a dictionary (or another mapping type), then the parenthesized formats in the string must refer to the keys in the dictionary. In other words, if you see %(name)s, then the dictionary to the right of the % must have a name key.

If you do not include all the keys that are required, you will receive an error:

```
1 >>> print('Hello %(first_name)s. You must be at least %(age)i to continue!'
2     % {'age': age})
3 Traceback (most recent call last):
4   Python Shell, prompt 23, line 1
5 KeyError: 'first_name'
```

For more information about using the printf-style string formatting, you should see the following link:

<https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>

Now let's move on to using the .format() method.

Formatting Strings Using .format()

Python strings have supported the .format() method for a long time. While this book will focus on using f-strings, you will find that .format() is still quite popular.

For full details on how formatting works, see the following:

<https://docs.python.org/3/library/string.html#formatstrings>

Let's take a look at a few short examples to see how .format() works:

```
1 >>> age = 18
2 >>> name = 'Mike'
3 >>> print('Hello {}. You must be at least {} to continue!'.format(
4     name, age))
5 Hello Mike. You must be at least 18 to continue!
```

This example uses positional arguments. Python looks for two instances of {} and will insert the variables accordingly. If you do not pass in enough arguments, you will receive an error like this:

```
1 >>> print('Hello {}. You must be at least {} to continue!'.format(
2     age))
3 Traceback (most recent call last):
4   Python Shell, prompt 33, line 1
5 IndexError: tuple index out of range
```

This error indicates that you do not have enough items inside the `.format()` call.

You can also use named arguments in a similar way to the previous section:

```
1 >>> age = 18
2 >>> name = 'Mike'
3 >>> print('Hello {first_name}. You must be at least {age} to continue!'.format(
4     first_name=name, age=age))
5 Hello Mike. You must be at least 18 to continue!
```

Instead of passing a dictionary to `.format()`, you can pass in the parameters by name. In fact, if you do try to pass in a dictionary, you will receive an error:

```
1 >>> print('Hello {first_name}. You must be at least {age} to continue!'.format(
2     {'first_name': name, 'age': age}))
3 Traceback (most recent call last):
4   Python Shell, prompt 34, line 1
5 KeyError: 'first_name'
```

There is a workaround for this though:

```
1 >>> print('Hello {first_name}. You must be at least {age} to continue!'.format(
2     **{'first_name': name, 'age': age}))
3 Hello Mike. You must be at least 18 to continue!
```

This looks a bit weird, but in Python when you see a double asterisk (**) used like this, it means that you are passing named parameters to the function. So Python is converting the dictionary to `first_name=name, age=age` for you.

You can also repeat a variable multiple times in the string when using `.format()`:

```
1 >>> first_name = 'Mike'
2 >>> print('Hello {first_name}. Why do they call you {first_name}?'.format(
3     first_name=first_name))
4 Hello Mike. Why do they call you Mike?
```

Here you refer to `{first_name}` twice in the string and Python replaces both of them with the `first_name` variable.

If you want, you can also interpolate values using numbers:

```
1 >>> print('Hello {1}. You must be at least {0} to continue!'.format(
2     first_name, age))
3 Hello 18. You must be at least Mike to continue!
```

Because most things in Python start at 0 (zero), in this example you ended up passing the `age` to `{1}` and the `name` to `{0}`.

A common coding style when working with `.format()` is to create a formatted string and save it to a variable to be used later:

```
1 >>> age = 18
2 >>> first_name = 'Mike'
3 >>> greetings = 'Hello {first_name}. You must be at least {age} to continue!'
4 >>> greetings.format(first_name=first_name, age=age)
5 'Hello Mike. You must be at least 18 to continue!'
```

This allows you to reuse `greetings` and pass in updated values for `first_name` and `age` later on in your program.

You can also specify the string width and alignment:

```
1 >>> '{:<20}'.format('left aligned')
2 'left aligned'
3 >>> '{:>20}'.format('right aligned')
4 '      right aligned'
5 >>> '{:^20}'.format('centered')
6 '      centered'
```

Left aligned is the default. The colon `:` tells Python that you are going to apply some kind of formatting. In the first example, you are specifying that the string be left aligned and 20 characters wide. The second example is also 20 characters wide, but it is right aligned. Finally the `^` tells Python to center the string within the 20 characters.

If you want to pass in a variable like in the previous examples, here is how you would do that:

```
1 >>> '{example:^20}'.format(example='centered')
2 '      centered'
```

Note that the `example` must come before the `:` inside of the `{}`.

At this point, you should be pretty familiar with the way `.format()` works.

Let's go ahead and move along to f-strings!

Formatting Strings with f-strings

Formatted string literals or f-strings are strings that have an “f” at the beginning and curly braces inside of them that contain expressions, much like the ones you saw in the previous section. These expressions tell the f-string about any special processing that needs to be done to the inserted string, such as justification, float precision, etc.

The f-string was added in Python 3.6. You can read more about it and how it works by checking out PEP 498 here:

<https://www.python.org/dev/peps/pep-0498/>

Let's go ahead and look at a simple example:

```
1 >>> name = 'Mike'  
2 >>> age = 20  
3 >>> f'Hello {name}. You are {age} years old'  
4 'Hello Mike. You are 20 years old'
```

Here you create the f-string by putting an “f” right before the single, double or triple quote that begins your string. Then inside of the string, you use the curly braces, {}, to insert variables into your string.

However, your curly braces must enclose something. If you create an f-string with empty braces, you will get an error:

```
1 >>> f'Hello {}. You are {} years old'  
2 SyntaxError: f-string: empty expression not allowed
```

The f-string can do things that neither %s nor .format() can do, though. Because of the fact that f-strings are evaluated at runtime, you can put any valid Python expression inside of them.

For example, you could increase the displayed value of the age variable:

```
1 >>> age = 20  
2 >>> f'{age+2}'  
3 '22'
```

Or call a method or function:

```
1 >>> name = 'Mike'  
2 >>> f'{name.lower()}'  
3 'mike'
```

You can also access dictionary values directly inside of an f-string:

```
1 >>> sample_dict = {'name': 'Tom', 'age': 40}
2 >>> f'Hello {sample_dict["name"]}. You are {sample_dict["age"]} years old'
3 'Hello Tom. You are 40 years old'
```

However, backslashes are not allowed in f-string expressions:

```
1 >>> print(f'My name is {name\n}')
2 SyntaxError: f-string expression part cannot include a backslash
```

But you can use backslashes outside of the expression in an f-string:

```
1 >>> name = 'Mike'
2 >>> print(f'My name is {name}\n')
3 My name is Mike
```

One other thing that you can't do is add a comment inside of an expression in an f-string:

```
1 >>> f'My name is {name # name of person}'
2 SyntaxError: f-string expression part cannot include '#'
```

In Python 3.8, f-strings added support for =, which will expand the text of the expression to include the text of the expression plus the equal sign and then the evaluated expression. That sounds kind of complicated, so let's look at an example:

```
1 >>> username = 'jdoe'
2 >>> f'Your {username=}'
3 "Your username='jdoe'"
```

This example demonstrates that the text inside of the expression, `username=` is added to the output followed by the actual value of `username` in quotes.

f-strings are very powerful and extremely useful. They will simplify your code quite a bit if you use them wisely. You should definitely give them a try.

Let's find out what else you can do with strings!

String Concatenation

Strings also allow concatenation, which is a fancy word for joining two strings into one.

To concatenate strings together, you can use the + sign:

```
1 >>> first_string = 'My name is'  
2 >>> second_string = 'Mike'  
3 >>> first_string + second_string  
4 'My name isMike'
```

Oops! It looks like the strings merged in a weird way because you forgot to add a space to the end of the `first_string`. You can change it like this:

```
1 >>> first_string = 'My name is '  
2 >>> second_string = 'Mike'  
3 >>> first_string + second_string  
4 'My name is Mike'
```

Another way to merge strings is to use the `.join()` method. The `.join()` method accepts an iterable, such as a list, of strings and joins them together.

```
1 >>> first_string = 'My name is' # no ending space  
2 >>> second_string = 'Mike'  
3 >>> ''.join([first_string, second_string])  
4 'My name isMike'
```

This will make the strings join right next to each other, just like `+` did. However, you can put something inside of the string that you are using for the join, and it will be inserted between each string in the list:

```
1 >>> ' '.join([first_string, second_string]) # a space is in the join string  
2 'My name is Mike'  
3 >>> '--'.join([first_string, second_string]) # a dash dash is in the join string  
4 'My name is--Mike'
```

More often than not, you can use an f-string rather than concatenation or `.join()` and the code will be easier to follow.

String Slicing

Slicing in strings works in much the same way that it does for Python lists. Let's take the string "Mike". The letter "M" is at position zero and the letter "e" is at position 3.

If you want to grab characters 0-3, you would use this syntax: `my_string[0:4]`

What that means is that you want the substring starting at position zero up to but not including position 4.

Here are a few examples:

```
1 >>> 'this is a string'[0:4]
2 'this'
3 >>> 'this is a string'[:4]
4 'this'
5 >>> 'this is a string'[-4:]
6 'ring'
```

The first example grabs the first four letters from the string and returns them. If you want to, you can drop the zero as that is the default and use `[:4]` instead, which is what example two does.

You can also use negative position values. So `[-4:]` means that you want to start at the end of the string and get the last four letters of the string. You will learn more about slicing in **chapter 6**, which is about the `list` data type.

You should play around with slicing on your own and see what other slices you can come up with.

Wrapping Up

Python strings are powerful and useful. They can be created using single, double, or triple quotes. Strings are objects, so they have methods. You also learned about string concatenation, string slicing, and three different methods of string formatting.

The newest flavor of string formatting is the f-string. It is also the most powerful and the currently preferred method for formatting strings.

Review Questions

1. What are 3 ways to create a string?
2. Run `dir("")`. This lists all the string methods you can use. Which of these methods will capitalize each of the words in a sentence?
3. Change the following example to use f-strings:

```
1 >>> name = 'Mike'
2 >>> age = 21
3 >>> print('Hello %s! You are %i years old.' % (name, age))
4 Hello Mike! You are 21 years old.
```

4. How do you concatenate these two strings together?

```
1 >>> first_string = 'My name is'  
2 >>> second_string = 'Mike'
```

5. Use string slicing to get the substring, “is a”, out of the following string:

```
1 >>> 'this is a string'
```

Chapter 5 - Numeric Types

Python is a little different than some languages in that it only has three built-in numeric types. A built-in data type means that you don't have to do anything to use them other than typing out their name.

The built-in numeric types are:

- `int`
- `float`
- `complex`

Python 2 also had the `long` numeric type, which was an integer able to represent values larger than an `int` could. In Python 3, `int` and `long` were combined so that Python 3 only has `int`. You can create an `int` by simply typing the number or by using `int(2)`, `int(3)`, `int(4)`, and `int("5")` are all integers.

If you are familiar with C++, you probably know that floating-point numbers are defined using the `double` keyword. In Python, you can create a `float` by typing it or by using `float(3.14)`, `float(5.0)`, `float(7.9)`, and `float("8.1")` are all floating point numbers.

A `complex` number has a real and an imaginary part. The real and imaginary parts are accessed using attribute notation: `.real` and `.imag`, respectively. Complex numbers can be created by either typing them or using `complex(2+1j)`, `complex(2-1j)`, `complex(5j)`, `complex(7+2j)`, `complex("7+2j")`, and `complex(7, 2j)` are all complex numbers.

There are two other numeric types that are included with Python in its standard library. They are as follows:

- `decimal` - for holding floating-point numbers that allow the user to define their precision
- `fractions` - rational numbers

You can import these libraries using Python's `import` keyword, which you will learn about in [chapter 16](#). You might also be interested in checking out Python's `round()` keyword or its `math` module.

Let's go ahead and learn a little bit more about how you can create and use numeric types in Python!

Integers

You can create an integer in two ways in Python. The most common way is to assign an integer to a variable:

```
1 my_integer = 3
```

The equals sign (=) is Python's **assignment** operator. It "assigns" the value on the right to the variable name on the left. So in the code above, you are *assigning* the value 3 to the variable `my_integer`.

The other way to create an integer is to use the `int` callable, like this:

```
1 my_integer = int(3)
```

Most of the time, you won't use `int()` to create an integer. In fact, `int()` is usually used for converting a string or other type to an integer. Another term for this is **casting**.

A little known feature about `int()` is that it takes an optional second argument for the base in which the first argument is to be interpreted. In other words, you can tell Python to convert to base2, base8, base16, etc.

Here's an example:

```
1 >>> int('10', 2)
2 2
3 >>> int('101', 2)
4 5
```

The first argument has to be a string while the second argument is the base, which in this case is 2. Now let's move on and learn how to create a **float**!

Floats

A **float** in Python refers to a number that has a decimal point in it. For example, 2.0 is a **float** while 2 is an **int**.

You can create a **float** in Python like this:

```
1 my_float = 2.0
```

This code will assign the number, 2.0, to the variable `my_float`.

You can also create a float like this:

```
1 my_float = float(2.0)
```

Python's `float()` built-in will convert an integer or even a string into a float if it can. Here's an example of converting a string to a float:

```
1 my_float = float("2.0")
```

This code converts the string, “2.0”, to a `float`. You can also cast string or floats to `int` using the `int()` built-in from the previous section.

Note: The `float` numeric type is inexact and may differ across platforms. You shouldn’t use the `float` type when dealing with sensitive numeric types, such as money values, due to rounding issues. Instead it is recommended that you use Python’s `decimal` module.

Complex Numbers

A complex number has a *real* and an *imaginary* part, which are each a floating-point number. Let’s look at an example with a complex number object named `comp` to see how you can access each of these parts by using `comp.real` and `comp.imag` to extract the real and imaginary parts, respectively, from the number:

```
1 >>> comp = 1 + 2j
2 >>> type(comp)
3 <class 'complex'>
4 >>> comp.real
5 1.0
6 >>> comp.imag
7 2.0
```

In the code sample above, you created a complex number. To verify that it is a complex number, you can use Python’s built-in `type` function on the variable. Then you extract the `real` and `imag` parts from the complex number.

You can also use the `complex()` built-in callable to create a complex number:

```
1 >>> complex(10, 12)
2 (10+12j)
```

Here you created a complex number in the interpreter, but you don’t assign the result to a variable.

Numeric Operations

All the numeric types, with the exception of `complex`, support a set of numeric operations.

Here is a list of the operations that you can do:

Operation	Result
a + b	The sum of a and b
a - b	The difference of a and b
a * b	The product of a and b
a / b	The quotient of a and b
a // b	The floored quotient of a and b
a % b	The remainder of a / b
-a	a negated (multiply by -1)
+a	a unchanged
abs(a)	absolute value of a
int(a)	a converted to integer
float(x)	a converted to a floating-point number
complex(re, im)	A complex number with real and imaginary
c.conjugate()	The conjugate of the complex number c
divmod(a, b)	The pair: (a // b, a % b)
pow(a, b)	a to the power of b
a ** b	a to the power of b

You should check out the full documentation for additional details about how numeric types work (scroll down to the *Numeric Types* section):

- <https://docs.python.org/3/library/stdtypes.html>

Augmented Assignment

Python supports doing some types of arithmetic using a concept called **Augmented Assignment**. This idea was first proposed in PEP 203:

- <https://www.python.org/dev/peps/pep-0203/>

The syntax allows you to do various arithmetic operations using the following operators:

`+= -= *= /= %= **= <<= >>= &= ^= |=`

This syntax is a shortcut for doing common arithmetic in Python. With it you can replace the following code:

```

1  >>> x = 1
2  >>> x = x + 2
3  >>> x
4  3

```

with this:

```
1 >>> x = 1
2 >>> x += 2
3 >>> x
4 3
```

This code is the equivalent of the previous example.

Wrapping Up

In this chapter, you learned the basics of Python’s Numeric types. Here you learned a little about how Python handles `int`, `float`, and `complex` number types. You can use these types for working with most operations that involve numbers. However, if you are working with floating-point numbers that need to be precise, you will want to check out Python’s `decimal` module. It is tailor-made for working with that type of number.

Review Questions

1. What 3 numeric types does Python support without importing anything?
2. Which module should you use for money or other precise calculations?
3. Give an example of how to use augmented assignment.

Chapter 6 - Learning About Lists

Lists are a fundamental data type in the Python programming language. A list is a mutable sequence that is typically a collection of homogeneous items. Mutable means that you can change a list after its creation. You will frequently see lists that contain other lists. These are known as nested lists. You will also see lists that contain all manner of other data types, such as dictionaries, tuples, and other objects.

In this chapter, you will learn the following:

- Creating Lists
- List Methods
- List Slicing
- List Copying

Let's find out how you can create a list!

Creating Lists

There are several ways to create a list. You may construct a list in any of the following ways:

- Using a pair of square brackets with nothing inside creates an empty list: []
- Using square brackets with comma-separated items: [1, 2, 3]
- Using a list comprehension (see Chapter 13 for more information): [x for x in iterable]
- Using the list() function: list(iterable)

An iterable is a collection of items that can return its members one at a time; some iterables have an order (i.e. sequences), and some do not. Lists themselves are sequences. Strings are sequences as well. You can think of strings as a sequence of characters.

Let's look at a few examples of creating a list so you can see it in action:

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list
3 [1, 2, 3]
```

This first example is pretty straight-forward. Here you create a list with 3 numbers in it. Then you print it out to verify that it contains what you think it should.

The next way to create a list is by using Python's built-in list() function:

```
1 >>> list_of_strings = list('abc')
2 >>> list_of_strings
3 ['a', 'b', 'c']
```

In this case, you pass a string of three letters to the `list()` function. It automatically iterates over the characters in the string to create a list of three strings, where each string is a single character.

The last example to look at is how to create empty lists:

```
1 >>> empty_list = []
2 >>> empty_list
3 []
4 >>> another_empty_list = list()
5 >>> another_empty_list
6 []
```

The quickest way to create an empty list is by using the square brackets without putting anything inside them. The second easiest way is to call `list()` without any arguments. The nice thing about using `list()` in general is that you can use it to cast a compatible data type to a `list`, as you did with the string “abc” in the example earlier.

List Methods

You haven’t learned about methods yet, but it is important to cover `list` methods now. Don’t worry. You will be learning more about methods throughout this book and by the end you will understand them quite well!

A Python `list` has several methods that you can call. A method allows you to do something to the list.

Here is a listing of the methods you can use with a `list`:

- `append()`
- `clear()`
- `copy()`
- `count()`
- `extend()`
- `index()`
- `insert()`
- `pop()`
- `remove()`
- `reverse()`
- `sort()`

Most of these will be covered in the following sections. Let's talk about the ones that aren't covered in a specific section first.

You can use `count()` to count the number of instances of the object that you passed in.

Here is an example:

```
1 >>> my_list = list('abcc')
2 >>> my_list.count('a')
3 1
4 >>> my_list.count('c')
5 2
```

This is a simple way to count the number of occurrences of an item in a list.

The `index()` method is useful for finding the first instance of an item in a list:

```
1 >>> my_list = list('abcc')
2 >>> my_list.index('c')
3 2
4 >>> my_list.index('a')
5 0
```

Python lists are zero-indexed, so "a" is in position 0, "b" is at position 1, etc.

You can use the `reverse()` method to reverse a list *in-place*:

```
1 >>> my_list = list('abcc')
2 >>> my_list.reverse()
3 >>> my_list
4 ['c', 'c', 'b', 'a']
```

Note that the `reverse()` method returns `None`. What that means is that if you try to assign the reversed list to a new variable, you may end up with something unexpected:

```
1 >>> x = my_list.reverse()
2 >>> print(x)
3 None
```

Here you end up with `None` instead of the reversed list. That is what *in-place* means. The original list is reversed, but the `reverse()` method itself doesn't return anything.

Now let's find out what you can do with the other `list` methods!

Adding to a List

There are three `list` methods that you can use to add to a list. They are as follows:

- `append()`
- `extend()`
- `insert()`

The `append()` method will add an item to the end of a pre-existing `list`:

```
1 >>> my_list = list('abcc')
2 >>> my_list
3 ['a', 'b', 'c', 'c']
4 >>> my_list.append(1)
5 >>> my_list
6 ['a', 'b', 'c', 'c', 1]
```

First you create a list that is made up of four one-character strings. Then you append an integer to the end of the list. Now the list should have 5 items in it with the `1` on the end.

You can use Python's built-in `len()` function to check the number of items in a `list`:

```
1 >>> len(my_list)
2 5
```

So this tells you that you do in fact have five items in the list. But what if you wanted to add an element somewhere other than the end of the list?

You can use `insert()` for that:

```
1 >>> my_list.insert(0, 'first')
2 >>> my_list
3 ['first', 'a', 'b', 'c', 'c', 1]
```

The `insert()` method takes two arguments:

- The position at which to insert
- The item to insert

In the code above, you tell Python that you want to insert the string, "first", into the 0 position, which is the first position in the list.

There are two other ways to add items to a `list`. You can add an iterable to a `list` using `extend()`:

```
1 >>> my_list = [1, 2, 3]
2 >>> other_list = [4, 5, 6]
3 >>> my_list.extend(other_list)
4 >>> my_list
5 [1, 2, 3, 4, 5, 6]
```

Here you create two lists. Then you use `my_list`'s `extend()` method to add the items in `other_list` to `my_list`.

The `extend()` method will iterate over the items in the passed in `list` and add each of them to the `list`.

You can also combine lists using concatenation:

```
1 >>> my_list = [1, 2, 3]
2 >>> other_list = [4, 5, 6]
3 >>> combined = my_list + other_list
4 >>> combined
5 [1, 2, 3, 4, 5, 6]
```

In this case, you create two lists and then combine them using Python's `+` operator. Note that `my_list` and `other_list` have not changed.

You can also use `+=` with Python lists:

```
1 >>> my_list = [1, 2, 3]
2 >>> other_list = [4, 5, 6]
3 >>> my_list += other_list
4 >>> my_list
5 [1, 2, 3, 4, 5, 6]
```

This is a somewhat simpler way to combine the two lists, but it does change the original list in the same way that using the `extend()` method does.

Now let's learn how to access and change elements within a `list`.

Accessing and Changing List Elements

Lists are made to be worked with. You will need to learn how to access individual elements as well as how to change them.

Let's start by learning how to access an item:

```
1 >>> my_list = [7, 8, 9]
2 >>> my_list[0]
3 7
4 >>> my_list[2]
5 9
```

To access an item in a list, you need to use square braces and pass in the index of the item that you wish to access. In the example above, you access the first and third elements.

Lists also support accessing items in reverse by using negative values:

```
1 >>> my_list[-1]
2 9
```

This example demonstrates that when you pass in -1, you get the last item in the list returned. Try using some other values and see if you can get the first item using negative indexing.

If you try to use an index that does not exist in the list, you will get an `IndexError`:

```
1 >>> my_list[-5]
2 Traceback (most recent call last):
3 Python Shell, prompt 41, line 1
4 builtins.IndexError: list index out of range
```

Now let's learn about removing items!

Deleting From a List

Deleting items from a list is pretty straight-forward. There are 4 primary methods of removing items from a list:

- `clear()`
- `pop()`
- `remove()`
- `del`

You can use `clear()` to remove everything from the list. Let's see how that works:

```
1 >>> my_list = [7, 8, 9]
2 >>> my_list.clear()
3 >>> my_list
4 []
```

After calling `clear()`, the `list` is now empty. This can be useful when you have finished working on the items in the `list` and you need to start over from scratch. Of course, you could also do this instead of `clear()`:

```
1 >> my_list = []
```

This will create a new empty `list`. If it is important for you to always use the same object, then using `clear()` would be better. If that does not matter, then setting it to an empty `list` will work well too.

If you would rather remove individual items, then you should check out `pop()` or `remove()`. Let's start with `pop()`:

```
1 >>> my_list = [7, 8, 9]
2 >>> my_list.pop()
3 9
4 >>> my_list
5 [7, 8]
```

You can pass an index to `pop()` to remove the item with that specific index and return it. Or you can call `pop()` without an argument, like in the example above, and it will default to removing the last item in the `list` and returning it. `pop()` is the most flexible way of removing items from a `list`.

If the `list` is empty or you pass in an index that does not exist, `pop()` will throw an exception:

```
1 >>> my_list.pop(10)
2 Traceback (most recent call last):
3   Python Shell, prompt 50, line 1
4     builtins.IndexError: pop index out of range
```

Now let's take a look at how `remove()` works:

```
1 >>> my_list = [7, 8, 9]
2 >>> my_list.remove(8)
3 >>> my_list
4 [7, 9]
```

`remove()` will delete the first instance of the passed in item. So in this case, you tell the `list` to remove the first occurrence of the number 8.

If you tell `remove()` to delete an item that is not in the `list`, you will receive an exception:

```
1 >>> my_list.remove(4)
2 Traceback (most recent call last):
3 Python Shell, prompt 51, line 1
4 builtins.ValueError: list.remove(x): x not in list
```

You can also use Python's built-in `del` keyword to delete items from a list:

```
1 >>> my_list = [7, 8, 9]
2 >>> del my_list[1]
3 >>> my_list
4 [7, 9]
```

You will receive an error if you try to remove an index that does not exist:

```
1 >>> my_list = [7, 8, 9]
2 >>> del my_list[6]
3 Traceback (most recent call last):
4     Python Shell, prompt 296, line 1
5 builtins.IndexError: list assignment index out of range
```

Now let's learn about sorting a list!

Sorting a List

Lists in Python can be sorted. You can use the built-in `sort()` method to sort a list *in-place* or you can use Python's `sorted()` function to return a new sorted list.

Let's create a list and try sorting it:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]
2 >>> my_list.sort()
3 >>> my_list
4 [1, 2, 4, 9, 10, 23]
```

Here you create a list with 6 integers in a pretty random order. To sort the list, you call its `sort()` method, which will sort it *in-place*. Remember that *in-place* means that `sort()` does not return anything.

A common misconception with Python is that if you call `sort()`, you can assign the now-sorted list to a variable, like this:

```
1 >>> sorted_list = my_list.sort()  
2 >>> print(sorted_list)  
3 None
```

However, when you do that, you will see that `sort()` doesn't actually return the sorted list. It always returns `None`.

Fortunately you can use Python's built-in `sorted()` method for this:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]  
2 >>> sorted_list = sorted(my_list)  
3 >>> sorted_list  
4 [1, 2, 4, 9, 10, 23]
```

If you use `sorted()`, it will return a new list, sorted ascending by default.

Both the `sort()` method and the `sorted()` function will also allow you to sort by a specified key and you can tell them to sort ascending or descending by setting its `reversed` flag.

Let's sort this list in descending order instead:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]  
2 >>> sorted_list = sorted(my_list, reverse=True)  
3 >>> sorted_list  
4 [23, 10, 9, 4, 2, 1]
```

When you have a more complicated data structure, such as a nested list or a dictionary, you can use `sorted()` to sort in special ways, such as by key or by value.

List Slicing

Python lists support the idea of slicing. Slicing a list is done by using square brackets and entering a start and stop value. For example, if you had `my_list[1:3]`, you would be saying that you want to create a new list with the element starting at index 1 through index 3 but not including index 3.

Here is an example:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]  
2 >>> my_list[1:3]  
3 [10, 2]
```

This slice returns index 1 (10) and index 2 (2) as a new list.

You can also use negative values to slice:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]
2 >>> my_list[-2:]
3 [23, 9]
```

In this example, you didn't specify an end value. That means you want to start at the second to last item in the list, 23, and take it to the end of the list.

Let's try another example where you specify only the end index:

```
1 >>> my_list = [4, 10, 2, 1, 23, 9]
2 >>> my_list[:3]
3 [4, 10, 2]
```

In this example, you want to grab all the values starting at index 0 up to but not including index 3.

Copying a List

Occasionally you will want to copy a list. One simple way to copy your list is to use the `copy` method:

```
1 >>> my_list = [1, 2, 3]
2 >>> new_list = my_list.copy()
3 >>> new_list
4 [1, 2, 3]
```

This successfully creates a new list and assigns it to the variable, `new_list`.

However note that when you do this, you are creating what is known as a “shallow copy”. What that means is that if you were to have mutable objects in your list, they can be changed and it will affect both lists. For example, if you had a dictionary in your list and the dictionary was modified, both lists will change, which may not be what you want. You will learn about dictionaries in **chapter 8**.

```
1 >>> my_list = [1, 2, 3]
2 >>> new_list = my_list.copy()
3 >>> my_list
4 [1, 2, 3]
5 >>> new_list
6 [1, 2, 3]
```

You can also copy a list by using this funny syntax:

```
1 >>> my_list = [1, 2, 3]
2 >>> new_list = my_list[:]
3 >>> new_list
4 [1, 2, 3]
```

This example is telling Python to create a slice from the 0 (first) element to the last, which in effect is the whole list.

Finally, you could also use Python's `list()` function to copy a list:

```
1 >>> my_list = [1, 2, 3]
2 >>> new_list = list(my_list)
3 >>> new_list
4 [1, 2, 3]
```

No matter which method you choose though, whether you duplicate a list by using `[:]`, `copy()` or `list()`, all three will create a shallow copy. To avoid running into weird issues where changing one list affects the copied list, you should use the `deepcopy` method from the `copy` module instead.

Wrapping Up

In this chapter, you learned all about Python's wonderful `list` data type. You will be using lists extensively when you are programming in Python.

You learned the following in this chapter:

- Creating Lists
- List Methods
- List Slicing
- List Copying

Now you are ready to move on and learn about tuples!

Review Questions

1. How do you create a `list`?
2. Create a list with 3 items and then use `append()` to add two more.
3. What is wrong with this code?

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list.remove(4)
```

4. How do you remove the 2nd item in this list?

```
1 >>> my_list = [1, 2, 3]
```

5. Create a list that looks like this: [4, 10, 2, 1, 23]. Use string slicing to get only the middle 3 items.

Chapter 7 - Learning About Tuples

Tuples are another sequence type in Python. Tuples consist of a number of values that are separated by commas. A tuple is immutable whereas a list is not. Immutable means that the tuple has a fixed value and cannot change. You cannot add or delete items in a tuple. Immutable objects are useful when you need a constant hash value. The most popular example of a hash value in Python is the key to a Python dictionary, which you will learn about in [chapter 8](#).

In this chapter, you will learn how to:

- Create tuples
- Work with tuples
- Concatenate tuples
- Special case tuples

Let's find out how to create tuples!

Creating Tuples

You can create tuples in several different ways. Let's take a look:

```
1 >>> a_tuple = 4, 5
2 >>> type(a_tuple)
3 <class 'tuple'>
```

One of the simplest methods of creating a tuple is to have a sequence of values separated by commas. Those values could be integers, lists, dictionaries, or any other object.

Depending on where in the code you are trying to create a tuple, just using commas might be ambiguous; you can always use parentheses to make it explicit:

```
1 >>> a_tuple = (2, 3, 4)
2 >>> type(a_tuple)
3 <class 'tuple'>
```

However, parentheses by themselves do *not* make a tuple!

```
1 >>> not_a_tuple = (5)
2 >>> type(not_a_tuple)
3 <class 'int'>
```

You can cast a `list` into a `tuple` using the `tuple()` function:

```
1 >>> a_tuple = tuple(['1', '2', '3'])
2 >>> type(a_tuple)
3 <class 'tuple'>
```

This example demonstrates how to convert or cast a Python `list` into a `tuple`.

Working With Tuples

There are not many ways to work with tuples due to the fact that they are immutable. If you were to run `dir(tuple())`, you would find that tuples have only two methods:

- `count()`
- `index()`

You can use `count()` to find out how many elements match the value that you pass in:

```
1 >>> a_tuple = (1, 2, 3, 3)
2 >>> a_tuple.count(3)
3 2
```

In this case, you can find out how many times the integer 3 appears in the tuple.

You can use `index()` to find the first index of a value:

```
1 >>> a_tuple = (1, 2, 3, 3)
2 >>> a_tuple.index(2)
3 1
```

This example shows you that the number 2 is at index 1, which is the second item in the tuple. Tuples are zero-indexed, meaning that the first element starts at zero.

You can use the indexing methodology that you learned about in the previous chapter to access elements within a tuple:

```
1 >>> a_tuple = (1, 2, 3, 3)
2 >>> a_tuple[2]
3 3
```

The first “3” in the tuple is at index 2.

Let’s try to modify an element in your tuple:

```
1 >>> a_tuple[0] = 8
2 Traceback (most recent call last):
3     Python Shell, prompt 92, line 1
4 TypeError: 'tuple' object does not support item assignment
```

Here you try to set the first element in the tuple to 8. However, this causes a `TypeError` to be raised because tuples are immutable and cannot be changed.

Concatenating Tuples

Tuples can be joined together, which in programming is called “concatenation”. However, when you do that, you will end up creating a new tuple:

```
1 >>> a_tuple = (1, 2, 3, 3)
2 >>> id(a_tuple)
3 140617302751760
4 >>> a_tuple += (6, 7)
5 >>> id(a_tuple)
6 140617282270944
```

Here you concatenate a second tuple to the first tuple. You can use Python’s `id()` function to see that the variable, `a_tuple`, has changed. The `id()` function returns the id of the object. An object’s ID is equivalent to an address in memory. The ID number changed after concatenating the second tuple. That means that you have created a new object.

Special Case Tuples

There are two special-case tuples. A tuple with zero items and a tuple with one item. The reason they are special cases is that the syntax to create them is a little different.

To create an empty tuple, you can do one of the following:

```
1 >>> empty = tuple()
2 >>> len(empty)
3 0
4 >>> type(empty)
5 <class 'tuple'>
6 >>> also_empty = ()
7 >>> len(also_empty)
8 0
```

You can create an empty tuple by calling the `tuple()` function with no arguments or via assignment when using an empty pair of parentheses.

Now let's create a tuple with a single element:

```
1 >>> single = 2,
2 >>> len(single)
3 1
4 >>> type(single)
5 <class 'tuple'>
```

To create a tuple with a single element, you can assign a value with a following comma. Note the trailing comma after the 2 in the example above.

While the parentheses are usually optional, I highly recommend them for single-item tuples as the comma can be easy to miss.

Wrapping Up

The tuple is a fundamental data type in Python. It is used quite often and is certainly one that you should be familiar with. You will be using tuples in other data types. You will also use tuples to group related data, such as a name, address and country.

In this chapter, you learned how to create a tuple in three different ways. You also learned that tuples are immutable. Finally, you learned how to concatenate tuples and create empty tuples. Now you are ready to move on to the next chapter and learn all about dictionaries!

Review Questions

1. How do you create a tuple?
2. Can you show how to access the 3rd element in this tuple?

```
1  >>> a_tuple = (1, 2, 3, 4)
```

3. Is it possible to modify a tuple after you create it? Why or why not?

4. How do you create a tuple with a single item?

Chapter 8 - Learning About Dictionaries

Dictionaries are another fundamental data type in Python. A dictionary is a (key, value) pair. Some programming languages refer to them as hash tables. They are described as a *mapping* that maps keys (hashable objects) to values (any object). Immutable objects are hashable (*immutable* means unable to change).

Starting in Python 3.7, dictionaries are ordered. What that means is that when you add a new (key, value) pair to a dictionary, it remembers what order they were added. Prior to Python 3.7, this was not the case and you could not rely on insertion order.

You will learn how to do the following in this chapter:

- Creating dictionaries
- Accessing dictionaries
- Dictionary methods
- Modifying dictionaries
- Deleting items from your dictionary

Let's start off by learning about creating dictionaries!

Creating Dictionaries

You can create a dictionary in a couple of different ways. The most common method is by placing a comma-separated list of key: value pairs within curly braces.

Let's look at an example:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict  
4 { 'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

You can also use Python's built-in `dict()` function to create a dictionary. `dict()` will accept a series of keyword arguments (i.e. `1='one'`, `2='two'`, etc), a list of tuples, or another dictionary.

Here are a couple of examples:

```
1 >>> numbers = dict(one=1, two=2, three=3)
2 >>> numbers
3 {'one': 1, 'two': 2, 'three': 3}
4 >>> info_list = [('first_name', 'James'), ('last_name', 'Doe'),
5 ('email', 'jdoes@gmail.com')]
6 >>> info_dict = dict(info_list)
7 >>> info_dict
8 {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoes@gmail.com'}
```

The first example uses `dict()` on a series of keyword arguments. You will learn more about these when you learn about functions. You can think of keyword arguments as a series of keywords with the equals sign between them and their value.

The second example shows you how to create a list that has 3 tuples inside of it. Then you pass that list to `dict()` to convert it to a dictionary.

Accessing Dictionaries

Dictionaries' claim to fame is that they are very fast. You can access any value in a dictionary via the key. If the key is not found, you will receive a `KeyError`.

Let's take a look at how to use a dictionary:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',
2 'email': 'jdoe@gmail.com'}
3 >>> sample_dict['first_name']
4 'James'
```

To get the value of `first_name`, you must use the following syntax: `dictionary_name[key]`

Now let's try to get a key that doesn't exist:

```
1 >>> sample_dict['address']
2 Traceback (most recent call last):
3 Python Shell, prompt 118, line 1
4 builtins.KeyError: 'address'
```

Well that didn't work! You asked the dictionary to give you a value that wasn't in the dictionary!

You can use Python's `in` keyword to ask if a key is in the dictionary:

```
1 >>> 'address' in sample_dict
2 False
3 >>> 'first_name' in sample_dict
4 True
```

You can also check to see if a key is **not** in a dictionary by using Python's `not` keyword:

```
1 >>> 'first_name' not in sample_dict
2 False
3 >>> 'address' not in sample_dict
4 True
```

Another way to access keys in dictionaries is by using one of the dictionary methods. Let's find out more about dictionary methods now!

Dictionary Methods

As with most Python data types, dictionaries have special methods you can use. Let's check out some of the dictionary's methods!

`d.get(key[, default])`

You can use the `get()` method to get a value. `get()` requires you to specify a key to look for. It optionally allows you to return a default if the key is not found. The default for that value is `None`. Let's take a look:

```
1 >>> print(sample_dict.get('address'))
2 None
3 >>> print(sample_dict.get('address', 'Not Found'))
4 Not Found
```

The first example shows you what happens when you try to `get()` a key that doesn't exist without setting `get`'s `default`. In that case, it returns `None`. Then the second example shows you how to set the default to the string "Not Found".

`d.clear()`

The `clear()` method can be used to remove all the items from your dictionary.

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict  
4 {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}  
5 >>> sample_dict.clear()  
6 >>> sample_dict  
7 {}
```

d.copy()

If you need to create a shallow copy of the dictionary, then the `copy()` method is for you:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> copied_dict = sample_dict.copy()  
4 >>> copied_dict  
5 {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

If your dictionary has objects or dictionaries inside of it, then you may end up running into logic errors as a result of using this method, because changing one dictionary will affect the other. In this case you should use Python's `copy` module, which has a `deepcopy` function that will create a completely separate copy for you.

You may remember this issue being mentioned back in the chapter on lists. These are common problems with creating “shallow” copies.

d.items()

The `items()` method will return a new view of the dictionary's items:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict.items()  
4 dict_items([('first_name', 'James'), ('last_name', 'Doe'),  
5 ('email', 'jdoe@gmail.com')])
```

This view object will change as the dictionary object itself changes.

d.keys()

If you need to get a view of the keys that are in a dictionary, then `keys()` is the method for you. As a view object, it will provide you with a dynamic view of the dictionary's keys. You can iterate over a view and also check membership via the `in` keyword:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> keys = sample_dict.keys()  
4 >>> keys  
5 dict_keys(['first_name', 'last_name', 'email'])  
6 >>> 'email' in keys  
7 True  
8 >>> len(keys)  
9 3
```

d.values()

The `values()` method also returns a view object, but in this case it is a dynamic view of the dictionary's values:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> values = sample_dict.values()  
4 >>> values  
5 dict_values(['James', 'Doe', 'jdoe@gmail.com'])  
6 >>> 'Doe' in values  
7 True  
8 >>> len(values)  
9 3
```

d.pop(key[, default])

Do you need to remove a key from a dictionary? Then `pop()` is the method for you. The `pop()` method takes a key and an option default string. If you don't set the default and the key is not found, a `KeyError` will be raised.

Here are some examples:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict.pop('something')  
4 Traceback (most recent call last):  
5     Python Shell, prompt 146, line 1  
6 builtins.KeyError: 'something'  
7 >>> sample_dict.pop('something', 'Not found!')  
8 'Not found!'  
9 >>> sample_dict.pop('first_name')
```

```
10 'James'
11 >>> sample_dict
12 {'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

d.popitem()

The `popitem()` method is used to remove and return a `(key, value)` pair from the dictionary. The pairs are returned in last-in first-out (LIFO) order, which means that the last item added will also be the first one that is removed when you use this method. If called on an empty dictionary, you will receive a `KeyError`.

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',
2 'email': 'jdoe@gmail.com'}
3 >>> sample_dict.popitem()
4 ('email', 'jdoe@gmail.com')
5 >>> sample_dict
6 {'first_name': 'James', 'last_name': 'Doe'}
```

d.update([other])

Update a dictionary with the `(key, value)` pairs from `other`, overwriting existing keys. The `other` can be another dictionary, a list of tuples, etc.

`update()` will return `None` when called.

Let's look at a couple of examples:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',
2 'email': 'jdoe@gmail.com'}
3 >>> sample_dict.update([('something', 'else')])
4 >>> sample_dict
5 {'first_name': 'James',
6 'last_name': 'Doe',
7 'email': 'jdoe@gmail.com',
8 'something': 'else'}
```

Let's try using `update()` to overwrite a pre-existing key:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict.update([('first_name', 'Mike')])  
4 >>> sample_dict  
5 {'first_name': 'Mike', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

Modifying Your Dictionary

You will need to modify your dictionary from time to time. Let's assume that you need to add a new (key, value) pair:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict['address'] = '123 Dunn St'  
4 >>> sample_dict  
5 {'first_name': 'James',  
6 'last_name': 'Doe',  
7 'email': 'jdoe@gmail.com',  
8 'address': '123 Dunn St'  
9 }
```

To add a new item to a dictionary, you can use the square braces to enter a new key and set it to a value.

If you need to update a pre-existing key, you can do the following:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict['email'] = 'jame@doe.com'  
4 >>> sample_dict  
5 {'first_name': 'James', 'last_name': 'Doe', 'email': 'jame@doe.com'}
```

In this example, you set `sample_dict['email']` to `jame@doe.com`. Whenever you set a pre-existing key to a new value, you will overwrite the previous value.

You can also use the `update()` method from the previous section to modify your dictionary.

Deleting Items From Your Dictionary

Sometimes you will need to remove a key from a dictionary. You can use Python's `del` keyword for that:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> del sample_dict['email']  
4 >>> sample_dict  
5 {'first_name': 'James', 'last_name': 'Doe'}
```

In this case, you tell Python to delete the key “email” from `sample_dict`

The other method for removing a key is to use the dictionary’s `pop()` method, which was mentioned in the previous section:

```
1 >>> sample_dict = {'first_name': 'James', 'last_name': 'Doe',  
2 'email': 'jdoe@gmail.com'}  
3 >>> sample_dict.pop('email')  
4 'jdoe@gmail.com'  
5 >>> sample_dict  
6 {'first_name': 'James', 'last_name': 'Doe'}
```

When you use `pop()`, it will remove the key and return the value that is being removed.

Wrapping Up

The dictionary data type is extremely useful. You will find it handy to use for quick lookups of all kinds of data. You can set the value of the key: `value` pair to any object in Python. So you could store lists, tuples, and other objects as values in a dictionary.

You learned the following topics in this chapter:

- Creating dictionaries
- Accessing dictionaries
- Dictionary methods
- Modifying dictionaries
- Deleting items from your dictionary

It is fairly common to need a dictionary that will create a key when you try to access one that does not exist. If you have such a need, you should check out Python’s `collections` module. It has a `defaultdict` class that is made for exactly that use case.

Review Questions

1. How do you create a dictionary?
2. You have the following dictionary. How do you change the `last_name` field to ‘Smith’?

```
1  >>> my_dict = {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

3. Using the dictionary above, how would you remove the email field from the dictionary?
4. How do you get just the values from a dictionary?

Chapter 9 - Learning About Sets

A set data type is defined as an “unordered collection of distinct hashable objects” according to the Python 3 documentation. You can use a set for membership testing, removing duplicates from a sequence and computing mathematical operations, like intersection, union, difference, and symmetric difference.

Due to the fact that they are unordered collections, a set does not record element position or order of insertion. Because of that, they also do not support indexing, slicing or other sequence-like behaviors that you have seen with lists and tuples.

There are two types of set built-in to the Python language:

- `set` - which is mutable
- `frozenset` - which is immutable and hashable

This chapter will focus on `set`.

You will learn how to do the following with sets:

- Creating a set
- Accessing set members
- Changing items
- Adding items
- Removing items
- Deleting a set

Let's get started by creating a set!

Creating a Set

Creating a set is pretty straight-forward. You can create them by adding a series of comma-separated objects inside of curly braces or you can pass a sequence to the built-in `set()` function.

Let's look at an example:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set  
3 {'c', 'a', 'b'}  
4 >>> type(my_set)  
5 <class 'set'>
```

A set uses the same curly braces that you used to create a dictionary. Note that instead of key:value pairs, you have a series of values. When you print out the set, you can see that duplicates were removed automatically.

Now let's try creating a set using `set()`:

```
1 >>> my_list = [1, 2, 3, 4]  
2 >>> my_set = set(my_list)  
3 >>> my_set  
4 {1, 2, 3, 4}  
5 >>> type(my_set)  
6 <class 'set'>
```

In this example, you created a list and then cast it to a set using `set()`. If there had been any duplicates in the list, they would have been removed.

Now let's move along and see some of the things that you can do with this data type.

Accessing Set Members

You can check if an item is in a set by using Python's `in` operator:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> "a" in my_set  
3 True
```

Sets do not allow you to use slicing or the like to access individual members of the set. Instead, you need to iterate over a set. You can do that using a loop, such as a `while` loop or a `for` loop.

You won't be covering loops until chapter 12, but here is the basic syntax for iterating over a collection using a `for` loop:

```
1 >>> for item in my_set:  
2 ...     print(item)  
3 ...  
4 c  
5 a  
6 b
```

This will loop over each item in the set one at a time and print it out.

You can access items in sets much faster than lists. A Python list will iterate over each item in a list until it finds the item you are looking for. When you look for an item in a set, it acts much like a dictionary and will find it immediately or not at all.

Changing Items

While both `dict` and `set` require hashable members, a `set` has no value to change. However, you can add items to a set as well as remove them. Let's find out how!

Adding Items

There are two ways to add items to a set:

- `add()`
- `update()`

Let's try adding an item using `add()`:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.add('d')  
3 >>> my_set  
4 {'d', 'c', 'a', 'b'}
```

That was easy! You were able to add an item to the set by passing it into the `add()` method.

If you'd like to add multiple items all at once, then you should use `update()` instead:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.update(['d', 'e', 'f'])  
3 >>> my_set  
4 {'a', 'c', 'd', 'e', 'b', 'f'}
```

Note that `update()` will take any iterable you pass to it. So it could take, for example, a list, tuple or another set.

Removing Items

You can remove items from sets in several different ways.

You can use:

- `remove()`
- `discard()`
- `pop()`

Let's go over each of these in the following sub-sections!

Using `.remove()`

The `remove()` method will attempt to remove the specified item from a set:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.remove('a')  
3 >>> my_set  
4 {'c', 'b'}
```

If you happen to ask the set to `remove()` an item that does not exist, you will receive an error:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.remove('f')  
3 Traceback (most recent call last):  
4   Python Shell, prompt 208, line 1  
5     builtins.KeyError: 'f'
```

Now let's see how the closely related `discard()` method works!

Using `.discard()`

The `discard()` method works in almost exactly the same way as `remove()` in that it will remove the specified item from the set:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.discard('b')  
3 >>> my_set  
4 {'c', 'a'}
```

The difference with `discard()` though is that it **won't** throw an error if you try to remove an item that doesn't exist:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.discard('d')  
3 >>>
```

If you want to be able to catch an error when you attempt to remove an item that does not exist, use `remove()`. If that doesn't matter to you, then `discard()` might be a better choice.

Using `.pop()`

The `pop()` method will remove and return an arbitrary item from the set:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.pop()  
3 'c'  
4 >>> my_set  
5 {'a', 'b'}
```

If your set is empty and you try to `pop()` an item out, you will receive an error:

```
1 >>> my_set = {"a"}  
2 >>> my_set.pop()  
3 'a'  
4 >>> my_set.pop()  
5 Traceback (most recent call last):  
6   Python Shell, prompt 219, line 1  
7     builtins.KeyError: 'pop from an empty set'
```

This is very similar to the way that `pop()` works with the `list` data type, except that with a `list`, it will raise an `IndexError`. Also lists are ordered while sets are not, so you can't be sure what you will be removing with `pop()` since sets are not ordered.

Clearing or Deleting a Set

Sometimes you will want to empty a set or even completely remove it.

To empty a set, you can use `clear()`:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.clear()  
3 >>> my_set  
4 set()
```

If you want to completely remove the set, then you can use Python's `del` built-in:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> del my_set  
3 >>> my_set  
4 Traceback (most recent call last):  
5     Python Shell, prompt 227, line 1  
6 builtins.NameError: name 'my_set' is not defined
```

Now let's learn what else you can do with sets!

Set Operations

Sets provide you with some common operations such as:

- `union()` - Combines two sets and returns a new set
- `intersection()` - Returns a new set with the elements that are common between the two sets
- `difference()` - Returns a new set with elements that are not in the other set

These operations are the most common ones that you will use when working with sets.

The `union()` method is actually kind of like the `update()` method that you learned about earlier, in that it combines two or more sets together into a new set. However the difference is that it returns a new set rather than updating the original set with new items:

```
1 >>> first_set = {'one', 'two', 'three'}  
2 >>> second_set = {'orange', 'banana', 'peach'}  
3 >>> first_set.union(second_set)  
4 {'two', 'banana', 'three', 'peach', 'orange', 'one'}  
5 >>> first_set  
6 {'two', 'three', 'one'}
```

In this example, you create two sets. Then you use `union()` on the first set to add the second set to it. However `union` doesn't update the `set`. It creates a new `set`. If you want to save the new `set`, then you should do the following instead:

```
1 >>> united_set = first_set.union(second_set)
2 >>> united_set
3 {'two', 'banana', 'three', 'peach', 'orange', 'one'}
```

The `intersection()` method takes two sets and returns a new set that contains only the items that are the same in both of the sets.

Let's look at an example:

```
1 >>> first_set = {'one', 'two', 'three'}
2 >>> second_set = {'orange', 'banana', 'peach', 'one'}
3 >>> first_set.intersection(second_set)
4 {'one'}
```

These two sets have only one item in common: the string “one”. So when you call `intersection()`, it returns a new set with a single element in it. As with `union()`, if you want to save off this new set, then you would want to do something like this:

```
1 >>> intersection = first_set.intersection(second_set)
2 >>> intersection
3 {'one'}
```

The `difference()` method will return a new set with the elements in the set that are **not** in the other set. This can be a bit confusing, so let's look at a couple of examples:

```
1 >>> first_set = {'one', 'two', 'three'}
2 >>> second_set = {'three', 'four', 'one'}
3 >>> first_set.difference(second_set)
4 {'two'}
5 >>> second_set.difference(first_set)
6 {'four'}
```

When you call `difference()` on the `first_set`, it returns a set with “two” as its only element. This is because “two” is the only string not found in the `second_set`. When you call `difference()` on the `second_set`, it will return “four” because “four” is not in the `first_set`.

There are other methods that you can use with sets, but they are used pretty infrequently. You should go check the documentation for full details on set methods should you need to use them.

Wrapping Up

Sets are a great data type that is used for pretty specific situations. You will find sets most useful for de-duplicating lists or tuples or by using them to find differences between multiple lists.

In this chapter, you learned about the following:

- Creating a set
- Accessing set members
- Changing items
- Adding items
- Removing items
- Deleting a set

Any time you need to use a set-like operation, you should take a look at this data type. However, in all likelihood, you will be using lists, dictionaries, and tuples much more often.

Review Questions

1. How do you create a set?
2. Using the following set, how would you check to see if it contains the string, “b”?


```
1  >>> my_set = {"a", "b", "c", "c"}
```

3. How do you add an item to a set?
4. Remove the letter “c” from the following set using a set method:

```
1  >>> my_set = {"a", "b", "c", "c"}
```

5. How do you find the common items between two sets?

Chapter 10 - Boolean Operations and None

You will find that you often need to know if something is `True` or `False`. For example, you might want to know if someone is old enough to create a bank account. If they are, that is usually represented as `True`. These values are known as Booleans or `bool` for short.

In Python, `False` maps to 0 (zero) and `True` maps to 1 (one).

You can easily see this is true using Python's interpreter:

```
1 >>> True == 1
2 True
3 >>> False == 0
4 True
5 >>> False == True
6 False
```

When you want to compare two values in Python, you need to use `==` instead of a single `=`. A single `=` is known as the assignment operator, as was mentioned in previous chapters. It assigns the value on the right to the variable on the left.

Let's try to assign a value to `True` and see what happens:

```
1 >>> True = 1
2 Traceback (most recent call last):
3   Python Shell, prompt 4, line 1
4 Syntax Error: can't assign to keyword: <string>, line 1, pos 0
```

Python doesn't allow that!

You can't assign anything to keywords in Python.

The `bool()` Function

Python also provides the `bool()` function, which allows you to cast other types to `True` or `False`.

Let's give it a try:

```
1 >>> bool('1')
2 True
3 >>> bool('2')
4 True
5 >>> bool('0')
6 True
```

Anything greater than zero should be cast as `True`. But wait, that third one is a string with a zero in it and it returned `True` as well! What's going on here?

Python has the concept of “truthy” and “falsey”. What that means is that when you are dealing with non-Numeric types, `True` will map to sequences with one or more items and `False` will map to sequences with zero items.

In this case, the string, `'0'`, has one character, so it maps to `True`. Let's try it with an empty string:

```
1 >>> bool('')
2 False
```

Since the string is empty (i.e. it has no characters in it), it will cast to `False`.

Let's see what happens when we try casting some of Python's other types:

```
1 >>> bool([])
2 False
3 >>> bool(['something'])
4 True
5 >>> bool({})
6 False
7 >>> bool({1: 'one'})
8 True
9 >>> bool(12)
10 True
```

Here you try casting an empty list, a list with one item, an empty dictionary, a dictionary with one key/value pair and an integer. Empty lists and dictionaries map to `False`, while lists and dictionaries with one or more items map to `True`. Integers or floats that are 0 or 0.0 will map to `False`, while any other value will map to `True`.

What About `None`?

Python also has the concept of `None`, which is Python's null value. `None` is a keyword in Python and its data type is `NoneType`. `None` is not the same as 0, `False` or an empty string. In fact, comparing `None` to anything other than itself will return `False`:

```
1 >>> None == 1
2 False
3 >>> None == []
4 False
5 >>> None == ''
6 False
7 >>> None == None
8 True
```

You can assign `None` to a variable. Note that all instances of `None` point to the same object though:

```
1 >>> x = None
2 >>> y = None
3 >>> x
4 >>> id(x)
5 4478513256
6 >>> id(y)
7 4478513256
```

When you want to check if a variable is `None`, you should use Python's `is` operator. The reason for that is that `is` will check the variable's identity and verify that it really is `None`. You will learn more about why this is important in the next chapter.

Wrapping Up

The `bool` or Boolean type is important in programming as it a very simple way to test if something is `True` or `False`. You also learned a little about Python's `None` type, which is similar to null in other languages. You will be using the Boolean values `True` and `False`, as well as `None`, often when you are programming in Python.

Review Questions

1. What number does `True` equal?
2. How do you cast other data types to `True` or `False`?
3. What is Python's null type?

Chapter 11 - Conditional Statements

Developers have to make decisions all the time. How do you approach this problem? Do you use technology X or technology Y? Which programming language(s) can you use to solve this? Your code also sometimes needs to make a decision.

Here are some common things that code checks every day:

- Are you authorized to do that?
- Is that a valid email address?
- Is that value valid in that field?

These sorts of things are controlled using *conditional statements*. This topic is usually called *control flow*. In Python, you can control the flow of your program using `if`, `elif` and `else` statements. You can also do a crude type of control flow using exception handling, although that is usually not recommended.

Some programming languages also have `switch` or `case` statements that can be used for control flow. Python does not have those.

In this chapter you will learn about the following:

- Comparison operators
- Creating a simple conditional
- Branching conditional statements
- Nesting conditionals
- Logical operators
- Special operators

Let's get started by learning about comparison operators!

Comparison Operators

Before you get started using conditionals, it will be useful to learn about comparison operators. Comparison operators let you ask if something equals something else or if they are greater than or less than a value, etc.

Python's comparison operators are shown in the following table:

Operator	Meaning
>	Greater than - This is True if the left operand is greater than the right
<	Less than - This is True if the left operand is less than the right one
==	Equal to - This is True only when both operands are equal
!=	Not equal to - This is True if the operands are not equal
>=	Greater than or equal to - This is True when the left operand is greater than or equal to the right
<=	Less than or equal to - This is True when the left operand is less than or equal to the right

Now that you know what comparison operators are available to you in Python, you can start using them!

Here are some examples:

```

1  >>> a = 2
2  >>> b = 3
3  >>> a == b
4  False
5  >>> a > b
6  False
7  >>> a < b
8  True
9  >>> a >= b
10 False
11 >>> a <= b
12 True
13 >>> a != b
14 True

```

Go ahead and play around with the comparison operators yourself in a Python REPL. It's good to try it out a bit so that you completely understand what is happening.

Now let's learn how to create a conditional statement.

Creating a Simple Conditional

Creating a conditional statement allows your code to branch into two or more different paths. Let's take authentication as an example. If you go to your webmail account on a new computer, you will need to login to view your email. The code for the main page will either load up your email box when you go there or it will prompt you to login.

You can make a pretty safe bet that the code is using a conditional statement to check and see if you are authenticated / authorized to view the email. If you are, it loads your email. If you are not, it loads the login screen.

Let's create a pretend authentication example:

```
1 >>> authenticated = True
2 >>> if authenticated:
3 ...     print('You are logged in')
4 ...
5 You are logged in
```

In this example, you create a variable called `authenticated` and set it to `True`. Then you create a conditional statement using Python's `if` keyword. A conditional statement in Python takes this form:

```
1 if <expression>:
2     # do something here
```

To create a conditional, you start it with the word `if`, followed by an expression which is then ended with a colon. When that expression evaluates to `True`, the code underneath the conditional is executed.



Indentation Matters in Python

Python cares about indentation. A code block is a series of lines of code that is indented uniformly. Python determines where a code block begins and ends by this indentation.

Other languages use parentheses or semi-colons to mark the beginning or end of a code block.

Indenting your code uniformly is required in Python. If you do not do this correctly, your code will not run as you intended.

One other word of warning. Do not mix tabs and spaces. IDLE will complain if you do and your code may have hard-to-diagnose issues. The Python style guide (PEP8) recommends using 4 spaces to indent a code block. You can indent your code any number of spaces as long as it is consistent. However, 4 spaces are usually recommended.

If `authenticated` had been set to `False`, then nothing would have been printed out.

This code would be better if you handled both conditions though. Let's find out how to do that next!

Branching Conditional Statements

You will often need to do different things depending on the answer to a question. So for this hypothetical situation, you want to let the user know when they haven't authenticated so that they will login.

To get that to work, you can use the keyword `else`:

```
1 >>> authenticated = False
2 >>> if authenticated:
3 ...     print('You are logged in')
4 ... else:
5 ...     print('Please login')
6 ...
7 Please login
```

What this code is doing is checking the value of `authenticated`: if it evaluates to True it will print “You are logged in”, otherwise it will print “Please login”. In a real program, you would have more than just a `print()` statement. You would have code that would redirect the user to the login page or, if they were authenticated, it would run code to load their inbox.

Let’s look at a new scenario. In the following code snippet, you will create a conditional statement that will check your age and let you know how you can participate in elections depending on that factor:

```
1 >>> age = 10
2 >>> if age < 18:
3 ...     print('You can follow the elections on the news')
4 ... elif age < 35:
5 ...     print('You can vote in all elections')
6 ... elif age >= 35:
7 ...     print('You can stand for any election')
8 ...
9 You can follow the elections on the news
```

In this example, you use `if` and `elif`. The keyword `elif` is short for “else if”. So the code here is checking the age against different hard-coded values. If the age is less than 18, then the citizen can follow the elections on the news.

If they are older than 18 but less than 35, they can vote in all elections. Next you check if the citizen’s age is greater than or equal to 35. Then they can run for any office themselves and participate in their democracy as a politician.

You could change the last `elif` to be simply an `else` clause if you wanted to, but Python encourages developers to be explicit in their code and it’s easier to understand by using `elif` in this case.

You can use as many `elif` statements as you need, although it is usually recommended to only have a handful – a long `if/elif` statement probably needs to be reworked.

Nesting Conditionals

You can put an `if` statement inside of another `if` statement. This is known as nesting.

Let’s look at a silly example:

```
1  >>> age = 18
2  >>> car = 'Ford'
3  >>> if age >= 18:
4      ...     if car in ['Honda', 'Toyota']:
5          ...         print('You buy Japanese cars')
6          ...     elif car in ['Ford', 'Chevrolet']:
7              ...         print('You buy American cars')
8      ... else:
9          ...     print('You are too young to buy cars!')
10 ...
11 You buy American cars
```

This code has multiple paths that it can take because it depends on two variables: `age` and `car`. If the `age` is greater than a certain value, then it falls into the first code block and will execute the nested `if` statement, which checks the `car` type. If the `age` is less than an arbitrary amount then it will simply print out a message.

Theoretically, you can nest conditionals any number of times. However, the more nesting you do, the more complicated it is to debug later. You should keep the nesting to only one or two levels deep in most cases.

Fortunately, logical operators can help alleviate this issue!

Logical Operators

Logical operators allow you to chain multiple expressions together using special keywords.

Here are the three logical operators that Python supports:

- `and` - Only True if both the operands are true
- `or` - True if either of the operands are true
- `not` - True if the operand is false

Let's try using the logical operator `and` with the example from the last section to flatten your conditional statements:

```
1 >>> age = 18
2 >>> car = 'Ford'
3 >>> if age >= 18 and car in ['Honda', 'Toyota']:
4 ...     print('You buy Japanese cars')
5 ... elif age >= 18 and car in ['Ford', 'Chevrolet']:
6 ...     print('You buy American cars')
7 ... else:
8 ...     print('You are too young to buy cars!')
9 ...
10 You buy American cars
```

When you use `and`, both expressions must evaluate to True for the code underneath them to execute. So the first conditional checks to see if the age is greater than or equal to 21 AND the car is in the list of Japanese cars. Since it isn't both of those things, you drop down to the first `elif` and check those conditions. This time both conditions are True, so it prints your car preference.

Let's see what happens if you change the `and` to an `or`:

```
1 >>> age = 18
2 >>> car = 'Ford'
3 >>> if age >= 18 or car in ['Honda', 'Toyota']:
4 ...     print('You buy Japanese cars')
5 ... elif age >= 18 or car in ['Ford', 'Chevrolet']:
6 ...     print('You buy American cars')
7 ... else:
8 ...     print('You are too young to buy cars!')
9 ...
10 You buy Japanese cars
```

Wait a minute! You said your car was "Ford", but this code is saying you buy Japanese cars! What's going on here?

Well, when you use a logical `or` the code in that code block will execute if either of the statements are True.

Let's break this down a bit. There are two expressions in `if age >= 21 or car in ['Honda', 'Toyota']`. The first one is `age >= 21`. That evaluates to True. As soon as Python sees the `or` and that the first statement is True, it evaluates the whole thing as True. Either your age is greater than or equal to 21 **or** your car is Ford. Either way, it's true and that code gets executed.

Using `not` is a bit different. It doesn't really fit with this example at all, but does work with our previous authentication example:

```
1 >>> authenticated = False
2 >>> if not authenticated:
3 ...     print('Please login')
4 ... else:
5 ...     print('You are logged in')
6 ...
7 Please login
```

By using `not` we switched the success and failure blocks – this can be useful when the failure-handling code is short.

You can combine logical operators in a conditional statement. Here's an example:

```
1 >>> color = 'white'
2 >>> age = 10
3 >>> if age <= 14 and (color == 'white' or color == 'green'):
4 ...     print(f'This milk is {age} days old and looks {color}')
5 ... else:
6 ...     print(f'You should not drink this {age} day old milk...')
7 ...
8 This milk is 10 days old and looks white
```

This time around, you will run the code in the first half of the `if` statement if the age is smaller than or equal to 14 and the color of the milk is white or green. The parentheses around the 2nd and 3rd expressions are very important because of precedence, which means how important a particular operator is. `and` is more important than `or`, so without the parentheses Python interprets the above code as if you had typed `if (age <= 14 and color == 'white') or color == 'green':`. Not what you had intended! To prove this to yourself, change `color` to '`'green'`', `age` to `100`, remove the parentheses from the `if` statement, and run that code again.

Special Operators

There are some special operators that you can use in conditional expressions.

- `is` - True when the operands are identical (i.e. have the same `id`)
- `is not` - True when the operands are not identical
- `in` - True when the value is in a collection (`list`, `tuple`, `set`, etc.)
- `not in` - True when the value is not in a collection

The first two are used for testing **identity**. You want to know whether or not two items refer to the same object; we could use the `id()` function and either `==` or `!=`, but using `is` and `is not` is much simpler.

The last two are for checking membership: whether or not an item is in a collection – such as a value being in a list, or a key in a dictionary.

Let's look at how identity works:

```
1  >>> x = [1, 2, 3]
2  >>> y = [1, 2, 3]
3  >>> x == y
4  True
5  >>> x is y
6  False
7  >>> id(x)
8  140328193994832
9  >>> id(y)
10 140328193887760
```

Wait, what? Didn't you just assign the same list to both `x` and `y`? Not really, no. The first `[1, 2, 3]` creates a list which Python then assigns to `x`; the second `[1, 2, 3]` creates another list and Python assigns that one to `y` – two creations means two objects. Even though they are equal objects, they are still different.

Let's try again with strings:

```
1  >>> x = 'hello world'
2  >>> y = 'hello world'
3  >>> x == y
4  True
5  >>> x is y
6  False
7  >>> id(x)
8  139995196972928
9  >>> id(y)
10 139995196972984
```

Okay, looking good! One more time:

```
1  >>> x = 'hi'
2  >>> y = 'hi'
3  >>> x == y
4  True
5  >>> x is y
6  True
```

What just happened? Well, let's think about this for a moment... a `list` is mutable, which means we can change it, but a `str` is immutable, which means we cannot change it. Because immutable objects cannot be changed, Python is free to reuse existing, equivalent objects instead of creating new ones. So be very careful to only use `is` when you actually mean *exact same object* – using `is` for equality will sometimes accidentally work, but will eventually fail and be a bug in your program.

You can use `in` and `not in` to test if something is in a collection. Collections in Python refer to such things as lists, strings, tuples, dictionaries, etc.

Here's one way you could use this knowledge:

```
1 >>> valid_chars = 'yn'
2 >>> char = 'x'
3 >>> if char in valid_chars:
4     ...     print(f'{char} is a valid character')
5 ... else:
6     ...     print(f'{char} is not in {valid_chars}')
7 ...
8 x is not in yn
```

Here you check to see if the `char` is in the string of `valid_chars`. If it isn't, it will print out what the valid letters are.

Try changing `char` to a valid character, such as a lowercase "y" or lowercase "n" and re-run the code.

Here is one way you could use `not in`:

```
1 >>> my_list = [1, 2, 3, 4]
2 >>> 5 not in my_list
3 True
```

In this case, you are checking to see if an integer is not in a `list`.

Let's use another authentication example to demonstrate how you might use `not in`:

```
1 >>> ids = [1234, 5678]
2 >>> my_id = 1001
3 >>> if my_id not in ids:
4     ...     print('You are not authorized!')
5 ...
6 You are not authorized!
```

Here you have a set of known ids. These ids could be numeric like they are here or they could be email addresses or something else. Regardless, you need to check if the given id, `my_id`, is in your list of known ids. If it's not, then you can let the user know that they are not authorized to continue.

Wrapping Up

Conditional statements are very useful in programming. You will be using them often to make decisions based on what the user has chosen to do. You will also use conditional statements based on responses from databases, websites and anything else that your program gets its input from.

It can take some craftsmanship to create a really good conditional statement, but you can do it if you put enough thought and effort into your code!

Review Questions

1. Give a couple of examples of **comparison operators**:
2. Why does indentation matter in Python?
3. How do you create a **conditional statement**?
4. How do you use **logical operators** to check more than one thing at once?
5. What are some examples of **special operators**?
6. What is the difference between these two?

```
1 x = [4, 5, 6]
2 y = [4, 5, 6]
```

and

```
1 x = [4, 5, 6]
2 y = x
```

Chapter 12 - Learning About Loops

There are many times when you are writing code that you will need to process each object in a collection. In order to do that you will iterate over that collection, which means getting each object from that collection one at a time. Collections of objects include strings such as "Hello, world", lists like [1, 2, 3], and even files. The process of iterating over something is done via a loop, and objects that support being iterated over are called *iterables*.

In Python, there are two types of loop constructs:

- The `for` loop
- The `while` loop

Besides iterating over sequences and other collections, you can use a loop to do the same thing multiple times. One example of this is a web server: it waits, listening for a client to send it a message; when it receives the message, the code inside the loop will call a function in response.

Another example is the game loop. When you beat a game or lose a game, the game doesn't usually exit. Instead, it will ask you if you want to play again. This is done by wrapping the entire program in a loop.

In this chapter you will learn how to:

- Create a `for` loop
- Loop over a string
- Loop over a dictionary
- Extract multiple values from a tuple
- Use `enumerate` with loops
- Create a `while` loop
- Breakout of a loop
- Use `continue`
- Use `else` with loops
- Nest loops

Let's get started by looking at the `for` loop!

Creating a `for` Loop

The `for` loop is the most popular looping construct in Python. A `for` loop is created using the following syntax:

```
1 for x in iterable:  
2     # do something
```

Now the code above does nothing. So let's write a for loop that iterates over a list, one item at a time:

```
1 >>> my_list = [1, 2, 3]  
2 >>> for item in my_list:  
3 ...     print(item)  
4 ...  
5 1  
6 2  
7 3
```

In this code, you create a list with three integers in it. Next you create a for loop that says “for each item in my list, print out the item”.

Of course, most of the time you will actually want to do something to the item. For example, you might want to double it:

```
1 >>> my_list = [1, 2, 3]  
2 >>> for item in my_list:  
3 ...     print(f'{item * 2}')  
4 ...  
5 2  
6 4  
7 6
```

Or you might want to only print out only the even-numbered items:

```
1 >>> my_list = [1, 2, 3]  
2 >>> for item in my_list:  
3 ...     if item % 2 == 0:  
4 ...         print(f'{item} is even')  
5 ...  
6 2 is even
```

Here you use the modulus operator, `%`, to find the remainder of the item divided by 2. If the remainder is 0, then you know that the item is an even number.

You can use loops and conditionals and any other Python construct to create complex pieces of code that are only limited by your imagination.

Let's learn what else you can loop over besides lists.

Looping Over a String

One of the differences of the `for` loop in Python versus other programming languages is that you can iterate over any collection. So you can iterate over many data types.

Let's look at iterating over a string:

```
1  >>> my_str = 'abcdefg'  
2  >>> for letter in my_str:  
3  ...     print(letter)  
4  ...  
5  a  
6  b  
7  c  
8  d  
9  e  
10 f  
11 g
```

This shows you how easy it is to iterate over a string.

Now let's try iterating over another common data type!

Looping Over a Dictionary

Python dictionaries also support looping. By default, when you loop over a dictionary, you will loop over its keys:

```
1  >>> users = {'mdriscoll': 'password', 'guido': 'python', 'steve': 'guac',  
2  'ethanf': 'enum'}  
3  >>> for user in users:  
4  ...     print(user)  
5  ...  
6  mdriscoll  
7  guido  
8  steve  
9  ethanf
```

You can loop over both the key and the value of a dictionary if you make use of its `items()` method:

```
1 >>> users = {'mdriscoll': 'password', 'guido': 'python', 'steve': 'guac',
2 'ethanf': 'enum'}
3 >>> for user, password in users.items():
4 ...     print(f"{user}'s password is {password}")
5 ...
6 mdriscoll's password is password
7 guido's password is python
8 steve's password is guac
9 ethanf's password is enum
```

In this example, you specify that you want to extract the `user` and the `password` in each iteration. As you might recall, the `items()` method returns a view that is formatted like a list of tuples. Because of that, you can extract each key: `value` pair from this view and print them out.

Note that you should not modify a dict while looping over it. Instead, you should create a copy and loop over the copy while modifying the original.

This leads us to looping over tuples and getting out individual items from a tuple while looping!

Extracting Multiple Values in a Tuple While Looping

Sometimes you will need to loop over a list of tuples and get each item within the tuple. It sounds kind of weird, but you will find that it is a fairly common programming task.

```
1 >>> list_of_tuples = [(1, 'banana'), (2, 'apple'), (3, 'pear')]
2 >>> for number, fruit in list_of_tuples:
3 ...     print(f'{number} - {fruit}')
4 ...
5 1 - banana
6 2 - apple
7 3 - pear
```

To get this to work, you take advantage of the fact that you know each tuple has two items in it. Since you know the format of the list of tuples ahead of time, you know how to extract the values.

If you hadn't extracted the items individually from the tuples, you would have ended up with this kind of output:

```
1 >>> list_of_tuples = [(1, 'banana'), (2, 'apple'), (3, 'pear')]
2 >>> for item in list_of_tuples:
3 ...     print(item)
4 ...
5 (1, 'banana')
6 (2, 'apple')
7 (3, 'pear')
```

This is probably not what you expected. You will usually want to extract an item from the tuple or perhaps multiple items, rather than extracting the entire tuple.

Now let's discover another useful way to loop!

Using enumerate with Loops

Python comes with a built-in function called `enumerate`. This function takes in an iterable, like a string, list, or set, and returns a tuple in the form of `(count, item)`. The first value of `count` is 0 because, in Python, counting starts at zero. If the iterable is a sequence, like a string or list, then `count` is also the position of the item returned.

Here's an example:

```
1 >>> my_str = 'abcdefg'
2 >>> for pos, letter in enumerate(my_str):
3 ...     print(f'{pos} - {letter}')
4 ...
5 0 - a
6 1 - b
7 2 - c
8 3 - d
9 4 - e
10 5 - f
11 6 - g
```

Now let's look at the other type of loop that Python supports!

Creating a while Loop

Python has one other type of looping construct that is called the `while` loop. A `while` loop is created with the keyword `while` followed by an expression. In other words, `while` loops will run until a specific condition fails, or is no longer truthy.

Let's take a look at how these loops work:

```
1 >>> count = 0
2 >>> while count < 10:
3     ...     print(count)
4     ...     count += 1
```

This loop is formulated in much the same way as a conditional statement. You tell Python that you want the loop to run as long as the `count` is less than 10. Inside of the loop, you print out the current `count` and then you increment the `count` by one.

If you forgot to increment the `count`, the loop would run until you stop or terminate the Python process.

You can create an infinite loop by making that mistake or you could do something like this:

```
1 while True:
2     print('Program running')
```

Since the expression is always `True`, this code will print out the string, “Program running”, forever or until you kill the process.

Breaking Out of a Loop

Sometimes you want to stop a loop early. For example, you might want to loop until you find something specific. A good use case would be looping over the lines in a text file and stopping when you find the first occurrence of a particular string.

To stop a loop early, you can use the keyword `break`:

```
1 >>> count = 0
2 >>> while count < 10:
3     ...     if count == 4:
4         ...         print(f'{count=}')
5         ...         break
6     ...     print(count)
7     ...     count += 1
8
9 0
10 1
11 2
12 3
13 count=4
```

In this example, you want the loop to stop when the count reaches 4. To make that happen, you add a conditional statement that checks if `count` equals 4. When it does, you print out that the count equals 4 and then use the `break` statement to break out of the loop. (Remember that the trailing “=” in `f'{count=}'` requires Python 3.8; in earlier versions would need to write `f'count={count}'`.)

You can also use `break` in a `for` loop:

```
1 >>> list_of_tuples = [(1, 'banana'), (2, 'apple'), (3, 'pear')]
2 >>> for number, fruit in list_of_tuples:
3 ...     if fruit == 'apple':
4 ...         print('Apple found!')
5 ...         break
6 ...     print(f'{number} - {fruit}')
7 ...
8 1 - banana
9 Apple found!
```

For this example, you want to break out of the loop when you find an apple. Otherwise you print out what fruit you have found. Since the apple is in the second tuple, you will never get to the third one.

When you use `break`, the loop will only break out of the innermost loop that the `break` statement is in – an important thing to remember when you have nested loops!

You can use `break` to help control the flow of the program. In fact, conditional and looping statements are known as `flow control` statements.

Another loop flow control statement is `continue`. Let's look at that next!

Using `continue`

The `continue` statement is used for continuing to the next iteration in the loop. You can use `continue` to skip over something.

Let's write a loop that skips over even numbers:

```
1 >>> for number in range(2, 12):
2 ...     if number % 2 == 0:
3 ...         continue
4 ...     print(number)
5 ...
6 3
7 5
8 7
9 9
10 11
```

In this code, you loop over a range of numbers starting at 2 and ending at 11. For each number in this range, you use the modulus operator, %, to get the remainder of the number divided by 2. If the remainder is zero, it's an even number and you use the `continue` statement to continue to the next value in the sequence. This effectively skips even numbers so that you only print out the odd ones.

You can use clever conditional statements to skip over any number of things in a collection by using the `continue` statement.

Loops and the `else` Statement

A little known feature about Python loops is that you can add an `else` statement to them like you do with an `if/else` statement. The `else` statement only gets executed when no `break` statement occurs.

Another way to look at it is that the `else` statement only executes if the loop completes successfully.

The primary use case for the `else` statement in a loop is for searching for an item in a collection. You could use the `else` statement to raise an exception if the item was not found, or create the missing item, or whatever is appropriate for your use-case.

Let's look at a quick example:

```
1 >>> my_list = [1, 2, 3]
2 >>> for number in my_list:
3 ...     if number == 4:
4 ...         print('Found number 4!')
5 ...         break
6 ...     print(number)
7 ... else:
8 ...     print('Number 4 not found')
9 ...
10 1
11 2
12 3
13 Number 4 not found
```

This example loops over a list of three integers. It looks for the number 4 and will break out of the loop if it is found. If that number is not found, then the else statement will execute and let you know.

Try adding the number 4 to the list and then re-run the code:

```
1  >>> my_list = [1, 2, 3, 4]
2  >>> for number in my_list:
3  ...     if number == 4:
4  ...         print('Found number 4')
5  ...         break
6  ...     print(number)
7  ... else:
8  ...     print('Number 4 not found')
9 ...
10 1
11 2
12 3
13 Found number 4
```

A more proper way of communicating an error would be to raise an exception to signal the absence of the number 4 rather than printing a message. You will learn how to do that in [chapter 14](#).

Nesting Loops

Loops can also be nested inside of each other. There are many reasons to nest loops. One of the most common reasons is to unravel a nested data structure.

Let's use a nested list for your example:

```
1  >>> nested = [['mike', 12], ['jan', 15], ['alice', 8]]
2  >>> for lst in nested:
3  ...     print(f'List = {lst}')
4  ...     for item in lst:
5  ...         print(f'Item -> {item}')
```

The outer loop will extract each nested list and print it out as well. Then in the inner loop, your code will extract each item within the nested list and print it out.

If you run this code, you should see output that looks like this:

```
1 List = ['mike', 12]
2 Item -> mike
3 Item -> 12
4 List = ['jan', 15]
5 Item -> jan
6 Item -> 15
7 List = ['alice', 8]
8 Item -> alice
9 Item -> 8
```

This type of code is especially useful when the nested lists are of varying lengths. You may need to do extra processing on the lists that have extra data or not enough data in them, for example.

Wrapping Up

Loops are very helpful for iterating over data. In this chapter, you learned about Python's two looping constructs:

- The `for` loop
- The `while` loop

Specifically, you learned about the following topics:

- Creating a `for` loop
- Looping over a string
- Looping over a dictionary
- Extracting multiple values from a tuple
- Using `enumerate` with loops
- Creating a `while` loop
- Breaking out of a loop
- Using `continue`
- Loops and the `else` statement
- Nesting loops

With a little practice, you will soon become quite adept at using loops in your own code!

Review Questions

1. What two types of loops does Python support?
2. How do you loop over a string?
3. What keyword do you use to exit a loop?
4. How do you “skip” over an item when you are iterating?
5. What is the `else` statement for in loops?
6. What are the flow control statements in Python?

Chapter 13 - Python Comprehensions

Python supports a short hand method for creating lists, dictionaries and sets that is known as **comprehensions**. The common use case is that you want to create a new list where each element has had some kind of operation done to them. For example, if you have a list of numbers and you want to create a new list with all the numbers doubled, you could use a comprehension.

In this chapter you will learn about:

- List comprehensions
- Dictionary comprehensions
- Set comprehensions

Let's go ahead and learn about list comprehensions first!

List Comprehensions

A list comprehension allows you to create a list from another collection, such as another list or dict. A list comprehension, like a list, starts and ends with square brackets, [] ; it also has a for loop built in to it, all on a single line. This can optionally be followed by zero or more for or if clauses (inside the square brackets).

Here is an example:

```
1 >>> sequence = [1, 2, 3]
2 >>> new_list = [x for x in sequence]
```

This is equivalent to the following loop:

```
1 >>> sequence = [1, 2, 3]
2 >>> new_list = []
3 >>> for x in sequence:
4     ...     new_list.append(x)
5 ...
6 >>> new_list
7 [1, 2, 3]
```

Usually when you use a list comprehension, you want to do something to each of the items in the collection. For example, let's try doubling each of the items:

```
1 >>> sequence = [1, 2, 3]
2 >>> new_list = [x * 2 for x in sequence]
3 >>> new_list
4 [2, 4, 6]
```

In this example, for each item (x) in the sequence, you multiply x by 2.

Filtering List Comprehensions

You can add an `if` statement to a list comprehension as a type of filter. Python comes with a built-in `range()` function that takes in an integer. It then returns an iterable `range` object that allows you to get a range of integers starting at 0 and ending at the integer you passed in minus one.

Here is how it works:

```
1 >>> range(10)
2 range(0, 10)
3 >>> list(range(10))
4 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can use Python's `list()` function to turn the `range` object into a list of numbers, 0-9.

Now let's say you want to create a list that contains only the odd numbers in that range. You can use a list comprehension with an `if` statement in it to do that:

```
1 >>> odd_numbers = [x for x in range(10) if x % 2]
2 >>> odd_numbers
3 [1, 3, 5, 7, 9]
```

Let's try using nested `for` loops in a list comprehension. For this exercise, you will create a `dict` and transform it into a `list` of tuples:

```
1 >>> my_dict = {1: 'dog', 2: 'cat', 3: 'python'}
2 >>> [(num, animal) for num in my_dict for animal in my_dict.values() if my_dict[num] \
3 == animal]
4 [(1, 'dog'), (2, 'cat'), (3, 'python')]
```

This code creates a `tuple` of number and animals for each number in the dictionary and each animal, but it filters it so that it will only create the `tuple` if the dictionary key equals its value.

Here is the equivalent as a regular `for` loop:

```
1 >>> my_dict = {1: 'dog', 2: 'cat', 3: 'python'}
2 >>> my_list = []
3 >>> for num in my_dict:
4 ...     for animal in my_dict.values():
5 ...         if my_dict[num] == animal:
6 ...             my_list.append((num, animal))
7 ...
8 >>> my_list
9 [(1, 'dog'), (2, 'cat'), (3, 'python')]
```

Without the filter the above list would have been

```
1 [(1, 'dog'), (1, 'cat'), (1, 'python'), (2, 'dog'), (2, 'cat'), ... ]
```

You can make the previous list comprehension a bit more readable by putting some line breaks in it, like this:

```
1 >>> my_dict = {1: 'dog', 2: 'cat', 3: 'python'}
2 >>> [(num, animal) for num in my_dict
3 ...     for animal in my_dict.values()
4 ...     if my_dict[num] == animal]
5 [(1, 'dog'), (2, 'cat'), (3, 'python')]
```

This is easier to read than the one-liner version. List comprehensions are fun to write, but they can be difficult to debug or reacquaint yourself with. Always be sure to give good names to the variables inside of list comprehensions. If the comprehension gets too complex, it would probably be better to break it down into an actual `for` loop.

Nested List Comprehensions

You can also nest list comprehensions inside of each other. For the most part, this is not recommended. If you search the Internet, you will find that the most common use case for nesting list comprehensions is for matrix math.

A matrix is usually represented as a list of lists where the internal lists contain integers or floats.

Let's look at an example of that:

```
1 >>> matrix = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
2 >>> [[element * 2 for element in row] for row in matrix]
3 [[18, 16, 14], [12, 10, 8], [6, 4, 2]]
```

This matrix has three lists in it. These internal lists can be thought of as rows. Next you create a nested list comprehension that will loop over each element in a row and multiply that element by 2. Then in the outer portion of the list comprehension, you will loop over each row in the matrix itself.

If you get the chance, you should go check out Python's documentation on list comprehensions. It has several other interesting examples in it that are well worth your time.

Dictionary Comprehensions

Dictionary comprehensions were originally created in Python 3.0, but they were then backported to Python 2.7. You can read all about them in [Python Enhancement Proposal 274 \(PEP 274\)](#)¹, which goes into all the details of how they work.

The syntax for a dictionary comprehension is quite similar to a list comprehension. Instead of square brackets, you use curly braces. Inside the braces, you have a key: value expression followed by the for loop which itself can be followed by additional if or for clauses.

You can write a dictionary comprehension like this:

```
1 >>> {key: value for key, value in enumerate('abcde')}\n2 {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

In this example, you create a key: value pair for every key and value that is returned from enumerate. The enumerate function returns the current count (starting at zero) and the current item as it iterates over the data structure.

You probably won't see dictionary comprehensions as often as you will list comprehensions, as lists are much more common as temporary data structures. You also have to be careful when you create a dictionary comprehension as the keys have to be hashable. If they aren't, you will receive an exception.

Dictionary comprehensions also support using conditional statements for filters. Feel free to give that a try on your own!

Set Comprehensions

You learned about sets back in chapter 8. They are useful for creating collections that contain a unique group of elements. You can create a set using a set comprehension.

To create a set comprehension, you will need to use curly braces and loop over a collection. In fact, the syntax for a set comprehension matches a list comprehension completely except that set comprehensions use curly braces instead of square brackets.

Let's take a look:

¹<http://www.python.org/dev/peps/pep-0274/>

```
1 >>> my_list = list('aaabbcde')
2 >>> my_list
3 ['a', 'a', 'a', 'b', 'b', 'c', 'd', 'e']
4 >>> my_set = {item for item in my_list}
5 >>> my_set
6 {'d', 'e', 'c', 'b', 'a'}
```

Set comprehensions are pretty straightforward. Here you loop over each item in the `list` and put it into a `set`. Then you print out the `set` to verify that the elements were deduped.

Wrapping Up

This chapter covered Python's comprehension syntax. You can create lists, dictionaries, and sets using comprehensions. Comprehensions can be used to filter collections. In general, a comprehension is a one line for loop that returns a data structure. List comprehensions are the most common type of comprehension.

With a little practice, you will not only be able to write your own comprehensions, but you'll be able to read and understand others too!

Review Questions

1. How do you create a **list comprehension**?
2. What is a good use case for a list comprehension?
3. Create a dictionary using a **dict comprehension**
4. Create a set using a **set comprehension**

Chapter 14 - Exception Handling

Creating software is hard work. To make your software better, your application needs to keep working even when the unexpected happens. For example, let's say your application needs to pull information down from the Internet. What happens if the person using your application loses their Internet connectivity?

Another common issue is what to do if the user enters invalid input. Or tries to open a file that your application doesn't support.

All of these cases can be handled using Python's built-in exception handling capabilities, which are commonly referred to as the `try` and `except` statements.

In this chapter you will learn about:

- Common exceptions
- Handling exceptions
- Raising exceptions
- Examining exception objects
- Using the `finally` statement
- Using the `else` statement

Let's get starting by learning about some of the most common exceptions.

The Most Common Exceptions

Python supports lots of different exceptions. Here is a short list of the ones that you are likely to see when you first begin using the language:

- `Exception` - The base exception that all the others are based on
- `AttributeError` - Raised when an attribute reference or assignment fails.
- `ImportError` - Raised when an import statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.
- `ModuleNotFoundError` - A subclass of `ImportError` which is raised by `import` when a module could not be located
- `IndexError` - Raised when a sequence subscript is out of range.
- `KeyError` - Raised when a mapping (dictionary) key is not found in the set of existing keys.
- `KeyboardInterrupt` - Raised when the user hits the interrupt key (normally Control-C or Delete).

- `NameError` - Raised when a local or global name is not found.
- `OSError` - Raised when a function returns a system-related error.
- `RuntimeError` - Raised when an error is detected that doesn't fall in any of the other categories.
- `SyntaxError` - Raised when the parser encounters a syntax error.
- `TypeError` - Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
- `ValueError` - Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.
- `ZeroDivisionError` - Raised when the second argument of a division or modulo operation is zero.

For a full listing of the built-in exceptions, you can check out the Python documentation here:

- <https://docs.python.org/3/library/exceptions.html>.

Now let's find out how you can actually handle an exception when one occurs.

Handling Exceptions

Python comes with a special syntax that you can use to catch an exception. It is known as the `try/except` statement.

This is the basic form that you will use to catch an exception:

```
1 try:  
2     # Code that may raise an exception goes here  
3 except ImportError:  
4     # Code that is executed when an exception occurs
```

You put code that you expect might have an issue inside the `try` block. This might be code that opens a file or code that gets input from the user. The second block is known as the `except` block. In the above example the `except` block will only get executed if an `ImportError` is raised.

When you write the `except` without specifying the exception type, it is known as a **bare exception**. These are not recommended:

```
1 try:
2     with open('example.txt') as file_handler:
3         for line in file_handler:
4             print(line)
5 except:
6     print('An error occurred')
```

The reason it is bad practice to create a bare except is that you don't know what types of exceptions you are catching, nor exactly where they are occurring. This can make figuring out what you did wrong more difficult. If you narrow the exception types down to the ones you know how to deal with, then the unexpected ones will actually make your application crash with a useful message. At that point, you can decide if you want to catch that other exception or not.

Let's say you want to catch multiple exceptions. Here is one way to do that:

```
1 try:
2     with open('example.txt') as file_handler:
3         for line in file_handler:
4             print(line)
5     import something
6 except OSError:
7     print('An error occurred')
8 except ImportError:
9     print('Unknown import!')
```

This exception handler will catch two types of exceptions: `OSError` and `ImportError`. If another type of exception occurs, this handler won't catch it and your code will stop.

You can rewrite the code above to be a bit simpler by doing this:

```
1 try:
2     with open('example.txt') as file_handler:
3         for line in file_handler:
4             print(line)
5     import something
6 except (OSError, ImportError):
7     print('An error occurred')
```

Of course, by creating a tuple of exceptions, this will obfuscate which exception has occurred. In other words, this code makes it harder to know what problem actually happened.

Raising Exceptions

What do you do after you catch an exception? You have a couple of options. You can print out a message like you have been in the previous examples. You could also log the message to a log file for later debugging. Or, if the exception is one that you know needs to stop the execution of your application, you can re-raise the exception – possibly adding more information to it.

Raising an exception is the process of forcing an exception to occur. You raise exceptions in special cases. For example, if a file you need to access isn't found on the computer you might raise an exception.

You can use Python's built-in `raise` statement to raise an exception:

```
1 try:  
2     raise ImportError  
3 except ImportError:  
4     print('Caught an ImportError')
```

When you raise an exception, you can have it print out a custom message:

```
1 >>> raise Exception('Something bad happened!')  
2 Traceback (most recent call last):  
3   Python Shell, prompt 1, line 1  
4 builtins.Exception: Something bad happened!
```

If you don't provide a message, then the exception would look like this:

```
1 >>> raise Exception  
2 Traceback (most recent call last):  
3   Python Shell, prompt 2, line 1  
4 builtins.Exception:
```

Now let's learn about the exception object!

Examining the Exception Object

When an exception occurs, Python will create an exception object. You can examine the exception object by assigning it to a variable using the `as` statement:

```
1 >>> try:  
2 ...     raise ImportError('Bad import')  
3 ... except ImportError as error:  
4 ...     print(type(error))  
5 ...     print(error.args)  
6 ...     print(error)  
7 ...  
8 <class 'ImportError'>  
9 ('Bad import',)  
10 Bad import
```

In this example, you assigned the `ImportError` object to `error`. Now you can use Python's `type()` function to learn what kind of exception it was. This would allow you to solve the issue mentioned earlier in this chapter when you have a tuple of exceptions but you can't immediately know which exception you caught.

If you want to dive even deeper into debugging exceptions, you should look up Python's `traceback` module.

Using the `finally` Statement

There is more to the `try/except` statement than just `try` and `except`. You can add a `finally` statement to it as well. The `finally` statement is a block of code that will always get run even if there is an exception raised inside of the `try` portion.

You can use the `finally` statement for cleanup. For example, you might need to close a database connection or a file handle. To do that, you can wrap the code in a `try/except/finally` statement.

Let's look at a contrived example:

```
1 >>> try:  
2 ...     1 / 0  
3 ... except ZeroDivisionError:  
4 ...     print('You can not divide by zero!')  
5 ... finally:  
6 ...     print('Cleaning up')  
7 ...  
8 You can not divide by zero!  
9 Cleaning up
```

This example demonstrates how you can handle the `ZeroDivisionError` exception as well as add clean up code.

You can also skip the `except` statement entirely and create a `try/finally` instead:

```
1 >>> try:  
2 ...     1/0  
3 ... finally:  
4 ...     print('Cleaning up')  
5 ...  
6 Cleaning up  
7 Traceback (most recent call last):  
8 Python Shell, prompt 6, line 2  
9 builtins.ZeroDivisionError: division by zero
```

This time you don't handle the `ZeroDivisionError` exception, but the `finally` statement's code block runs anyway.

Using the `else` Statement

There is one other statement that you can use with Python's exception handling and that is the `else` statement. You can use the `else` statement to execute code when there are no exceptions.

Here is an example:

```
1 >>> try:  
2 ...     print('This is the try block')  
3 ... except IOError:  
4 ...     print('An IOError has occurred')  
5 ... else:  
6 ...     print('This is the else block')  
7 ...  
8 This is the try block  
9 This is the else block
```

In this code, no exception occurred, so the `try` block and the `else` blocks both run.

Let's try raising an `IOError` and see what happens:

```
1 >>> try:  
2 ...     raise IOError  
3 ...     print('This is the try block')  
4 ... except IOError:  
5 ...     print('An IOError has occurred')  
6 ... else:  
7 ...     print('This is the else block')  
8 ...  
9 An IOError has occurred
```

Since an exception was raised, only the `try` and the `except` blocks ran. Note that the `try` block stopped running at the `raise` statement. It never reached the `print()` function at all. Once an exception is raised, all the following code is skipped over and you go straight to the exception handling code.

Wrapping Up

Now you know the basics of using Python's built-in exception handling. In this chapter you learned about the following topics:

- Common exceptions
- Handling exceptions
- Raising exceptions
- Examining exception objects
- Using the `finally` statement
- Using the `else` statement

Learning how to catch exceptions effectively takes practice. Once you have learned how to catch exceptions, you will be able to harden your code and make it work in a much nicer way even when the unexpected happens.

Once you have learned about classes in [chapter 18](#), you will be able to create your own custom exceptions if you want to.

Review Questions

1. What are a couple of common exceptions?
2. How do you catch an exception in Python?
3. What do you need to do to raise a run time error?

4. What is the `finally` statement for?
5. How is the `else` statement used with an exception handler?

Chapter 15 - Working with Files

Application developers are always working with files. You create them whenever you write a new script or application. You write reports in Microsoft Word, you save emails or download books or music. Files are everywhere. Your web browser downloads lots of little files to make your browsing experience faster.

When you write programs, you have to interact with pre-existing files or write out files yourself. Python provides a nice, built-in function called `open()` that can help you with these tasks.

In this chapter you will learn how to:

- Open files
- Read files
- Write files
- Append to files

Let's get started!

The `open()` Function

You can open a file for reading, writing or appending. To open a file, you can use the built-in `open()` function.

Here is the `open()` function's arguments and defaults:

```
1 open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
2       closefd=True, opener=None)
```

When you open a file, you are required to pass in a file name or file path. The default when opening a file is to open it in read-only mode, which is what the 'r' means.

The following table goes over the other modes that can be used when opening a file:

Character	Meaning
'r'	open file for reading (default)
'w'	open for writing. If file exists, replace its contents
'a'	open for writing. If file exists, appends to end
'b'	binary mode
't'	text mode (default)
'+'	reading and writing

You will focus on reading, writing and appending in this chapter. If you need to encode your file in a specific format, such as UTF-8, you can set that via the `encoding` parameter. See the documentation for a full listing of the encoding types that Python supports.

There are two primary methods used to open a file. You can do something like this:

```
1 file_handler = open('example.txt')
2 # do something with the file
3 file_handler.close()
```

Here you open the file and you close it. But what happens if an exception occurs when you try to open the file? For example, let's say you tried to open a file that didn't exist. Or you opened a file, but you can't write to it. These things happen and they can cause a file handle to be left open and not closed properly.

One solution is to use `try/finally`:

```
1 try:
2     file_handler = open('example.txt')
3 except:
4     # ignore the error, print a warning, or log the exception
5     pass
6 finally:
7     file_handler.close()
```

However the best way to open a file in Python is to use Python's special `with` statement. The `with` statement activates what is known as a **context manager**. Context managers are used when you want to set something up and tear something down. In this example, you want to open a file, do something and then close the file.

The core developers of Python made `open()` into a context manager. What that means is that you can also open a file like this:

```
1 with open('example.txt') as file_handler:  
2     # do something with the handler here  
3     data = file_handler.read()
```

What this does is it opens the file and assigns the file object to `file_handler`. Then any code that is indented to be inside of the `with` statement is considered to be a part of the **context**. That is where you would interact with the file handler, whether that be reading or writing to the file. Then when you get out of the `with` statement, it will automatically close the file.

It's like having a `finally` statement that is built-in!

Now that you know how to open a file, let's move on and learn how to read a file with Python.

Reading Files

Reading files with the Python programming language is pretty straight-forward. In fact, when you open a file and don't set the `mode` argument, the default is to open the file in "read-only" mode.

Here is an example:

```
1 with open('example.txt') as file_handler:  
2     for line in file_handler:  
3         print(line)
```

This code will open the text file and then loop over each line in the file and print it out. Yes, the `file_handler` can be iterated over using Python's `for` loop, which is very handy. In fact, this is actually one of the recommended methods for reading a file as you are reading it in chunks so that you won't run out of memory.

An alternative way to loop over the lines in a file would be to do the following instead:

```
1 with open('example.txt') as file_handler:  
2     lines = file_handler.readlines()  
3     for line in lines:  
4         print(line)
```

If you go this route, then you just read the entire file into memory. Depending on how much RAM your machine has in it, you may run out of memory. This is why the first method is recommended. However, if you know the file is pretty small, there is another way to read the entire file into memory:

```
1 with open('example.txt') as file_handler:  
2     file_contents = file_handler.read()
```

The `read()` method will read the entire file into memory and assign it to your variable.

Occasionally you may want to read a file in smaller or larger chunks. This can be done by specifying the size in bytes to `read()`. You could use a `while` loop for this:

```
1 while True:  
2     with open('example.txt') as file_handler:  
3         data = file_handler.read(1024)  
4         if not data:  
5             break  
6         print(data)
```

In this example, you read 1024 bytes at a time. When you call `read()` and it returns an empty string, then the `while` loop will stop because the `break` statement will get executed.

Reading Binary Files

Sometimes you will need to read a binary file. Python can do that too by combining the `r` mode with `b`:

```
1 with open('example.pdf', 'rb') as file_handler:  
2     file_contents = file_handler.read()
```

Note the second argument to `open()` is `rb`. That tells Python to open the file in read-only binary mode. If you were to print out the `file_contents`, you would see what amounts to gibberish as most binary documents are not human readable.

Writing Files

Writing a new file in Python uses pretty much the exact same syntax as reading. But instead of setting the mode to `r`, you set it to `w` for write-mode. If you need to write in binary mode, then you would open the file in `wb` mode.

WARNING: When using the `w` and `wb` modes, if the file already exists, you will end up overwriting it. Python does not warn you in any way. Python does provide a way to check for a file's existence by using the `os` module via `os.path.exists()`. See Python's documentation for more details.

Let's write a single line of text to a file:

```
1 >>> with open('example.txt', 'w') as file_handler:  
2 ...     file_handler.write('This is a test')
```

This will write a single line of text to a file. If you write more text, it will be written right next to the previous text. So if you need to add a new line, then you will need to write one out using \n.

To verify that this worked, you can read the file and print out its contents:

```
1 >>> with open('example.txt') as file_handler:  
2 ...     print(file_handler.read())  
3 ...  
4 This is a test
```

If you need to write multiple lines at once, you can use the `writelines()` method, which accepts a sequence of strings. You could use a list of strings and pass them to `writelines()`, for example.

Seeking Within a File

The file handler also provides one other method that is worth mentioning. That method is `seek()` which you can use to change the file object's position. In other words, you can tell Python where in the file to start reading from.

The `seek()` method accepts two arguments:

- `offset` - A number of bytes from whence
- `whence` - The reference point

You can set `whence` to one of these three values:

- 0 - The beginning of the file (default)
- 1 - The current file position
- 2 - The end of the file

Let's use the file that you wrote to earlier in the chapter for an example:

```
1 >>> with open('example.txt') as file_handler:  
2 ...     file_handler.seek(4)  
3 ...     chunk = file_handler.read()  
4 ...     print(chunk)  
5 ...  
6 is a test
```

Here you open the file in read-only mode. Then you seek to the 4th byte and read the rest of the file into the variable `chunk`. Finally, you print out the `chunk` and see that you have retrieved only part of the file.

Appending to Files

You can also append data to a pre-existing file using the `a` mode, which is the append mode.

Here is an example:

```
1 >>> with open('example.txt', 'a') as file_handler:  
2 ...     file_handler.write('Here is some more text')
```

If the file exists, this will add a new string to the end of the file. On the other hand, if the file does not exist, Python will create the file and add this data to it.

Catching File Exceptions

When you are working with files, you will sometimes encounter errors. For example, you might not have the right permissions to create or edit the file. In that event, Python will raise an `OSError`. There are other errors that occasionally occur, but that is the most common one when working with files.

You can use Python's exception handling facilities to keep your program working:

```
1 try:  
2     with open('example.txt') as file_handler:  
3         for line in file_handler:  
4             print(line)  
5     except OSError:  
6         print('An error has occurred')
```

This code will attempt to open a file and print out its contents one line at a time. If an `OSError` is raised, you will catch it with the `try/except` and print out a message to the user.

Wrapping Up

Now you know the basics of working with files in Python. In this chapter you learned how to open files. Then you learned how to read and write files. You also learned how to seek within a file, append to a file, and handle exceptions when accessing files. At this point, you really only need to practice what you have learned. Go ahead and try out the things you have learned in this chapter and see what you can do on your own!

Review Questions

1. How do you open a file?
2. What do you need to do to read a file?
3. Write the following sentence to a file named **test.txt**:

1 The quick red fox jumped over the python

4. How do you append new data to a pre-existing file?
5. What do you need to do to catch a file exception?

Chapter 16 - Importing

One of Python's strongest advantages over other programming languages is its large, comprehensive standard library. The standard library is a set of modules or libraries that you can import into your own applications to enhance your programs.

Here are just a few examples of modules you might use:

- `argparse` - Create command-line interfaces
- `email` - Create, send, and process email
- `logging` - Create run-time logs of program execution
- `pathlib` - Work with file names and paths
- `subprocess` - Open and interact with other processes
- `sys` - Work with system specific functions and information
- `urllib` - Work with URLs

There are dozens and dozens of other libraries. You can see a full listing here:

- <https://docs.python.org/3/library/index.html>

If there isn't something in the standard library that will work for your use-case, you can usually find a 3rd party package that will. You will learn more about installing 3rd party packages in **chapter 20**.

In this chapter, you will learn how to:

- Use `import`
- Use `from` to import specific bits and pieces
- Use `as` to give the imported thing a new name
- Import everything

Let's get started by learning how to import a library!

Using `import`

Python has several different ways to import libraries. The simplest and most popular is to use the `import` keyword followed by the name of the library you wish to import. Imports should usually be put at the top of your Python file or script so that all the code in your program has access to the library.

Let's take a look at an example:

```
1 >>> import sys
2 >>> dir(sys)
3 [ '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__',
4  '__loader__', '__name__', '__package__', '__spec__', '__stderr__',
5  '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
6  '_debugmallocstats', '_getframe', '_git', '_home', '_xoptions',
7  'abiflags', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix',
8  'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
9  'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
10 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
11 'float_info', 'float_repr_style', 'get_asyncgen_hooks',
12 'getCoroutine_wrapper', 'getallocatedblocks', 'getcheckinterval',
13 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencodeerrors',
14 'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
15 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'hash_info',
16 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing',
17 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
18 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
19 'set_asyncgen_hooks', 'setCoroutine_wrapper', 'setcheckinterval',
20 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval',
21 'setattr', 'stderr', 'stdin', 'stdout', 'thread_info', 'version',
22 'version_info', 'warnoptions']
```

What happened here? When you wrote `import sys`, it imported Python's `sys` module, which is useful for figuring out such things as what arguments were passed to a Python script, adding an audit hook, and more.

You can read all the nitty gritty details here:

- <https://docs.python.org/3/library/sys.html>

You used Python's `dir()` command to see what is available to you in the `sys` library. This is known as **introspection** in Python, which you will learn more about in **chapter 19**.

A more fun import is one of Python's Easter eggs:

```
1 >>> import this
2 The Zen of Python, by Tim Peters
3
4 Beautiful is better than ugly.
5 Explicit is better than implicit.
6 Simple is better than complex.
7 Complex is better than complicated.
8 Flat is better than nested.
9 Sparse is better than dense.
10 Readability counts.
11 Special cases aren't special enough to break the rules.
12 Although practicality beats purity.
13 Errors should never pass silently.
14 Unless explicitly silenced.
15 In the face of ambiguity, refuse the temptation to guess.
16 There should be one-- and preferably only one --obvious way to do it.
17 Although that way may not be obvious at first unless you're Dutch.
18 Now is better than never.
19 Although never is often better than *right* now.
20 If the implementation is hard to explain, it's a bad idea.
21 If the implementation is easy to explain, it may be a good idea.
22 Namespaces are one honking great idea -- let's do more of those!
```

When you run that code, it prints out the “Zen of Python”, which is a fun little set of “rules” that describe the “best” way to write Python.

You can also import multiple libraries in one line of code:

```
1 >>> import math, os
```

This is usually discouraged, but it is certainly allowed. If you want to follow Python’s style guide, PEP8, then you really shouldn’t do this. But if it is okay with your organization, then it’s up to you. Just be consistent!

Once you have imported a library, you can call its functions and classes. For example:

```
1 >>> import math
2 >>> math.sqrt(4)
3 2.0
```

Here you called the `sqrt()` function that was in the `math` module to get the square root of 4.

Sometimes you may want to import just bits and pieces from a module. Let’s find out how to do that next!

Using `from` to Import Specific Bits & Pieces

There are times when it is nice to import parts of a library. Python supports this using the following syntax:

```
1 from module import function
```

You can import functions, classes, and variables from a module.

Let's look at a more realistic example:

```
1 >>> from math import sqrt
```

Here you are importing the `sqrt()` (square root) function from the `math` module.

You can also import multiple items from a module:

```
1 from math import sin, cos, tan
```

In this example, you import the `sin()`, `cos()`, and `tan()` functions, which are used for find the sine, cosine and tangent of an angle.

Besides importing individual functions, you can also import anything else in that module:

- variables
- enumerations
- classes
- sub-modules

For example, the `http` module has (sub-)modules of its own:

```
1 >>> import http
2 >>> type(http)
3 <class 'module'>
4 >>> from http import client
5 >>> type(client)
6 <class 'module'>
```

When you use many functions from a module, it can be helpful if that module has a shorter name. Let's find out how to do that!

Using `as` to assign a new name

You can use `as` to assign a different name for things you import. For example, if you are using many `math` functions and don't want to have to type `math` many times, you can do this:

```
1 >>> import math as m
2 >>> m.sqrt(4)
3 2.0
```

Likewise, if you would rather have sine, cosine, and tangent spelled out, you would do the following:

```
1 >>> from math import sin as sine, cos as cosine, tan as tangent
2 >>> sine(90)
3 0.8939966636005579
```

Python also supports importing everything at once. Let's find out how!

Importing Everything

You can import all the functions and variables in one go as well. However, you really shouldn't do this. The reason is that when you import everything, you don't always know what you import. This can cause **namespace contamination**, which is a fancy term for importing a function or variable and then accidentally re-using one of those names yourself, or overwriting one of your own previously imported functions.

Let's look at a simple example:

```
1 >>> from math import *
2 >>> tan = 5
3 >>> tan(10)
4 Traceback (most recent call last):
5   Python Shell, prompt 10, line 1
6 builtins.TypeError: 'int' object is not callable
```

Here you import everything from the `math` library via the `*` wildcard. The asterisk tells Python to import everything in the module.

Then you decide to create a variable called `tan`. What you may not realize is that you imported `tan` from the `math` module. If you try to call the `tan` function, you'll get a `TypeError` because you overwrote the `tan` function with your own variable, `tan`. This is known as **shadowing**.

The only time `from ... import *` is acceptable is when the module is specifically designed to support it. Overall, though, it is recommended that you avoid importing everything using the wildcard because it can make debugging issues like this very difficult. One of the few examples that you will find in Python where importing everything is done often is with the Tkinter library. Tkinter is a built-in cross-platform GUI library for Python that you can use to create desktop applications. The variables, functions, and classes that you import from Tkinter are fairly unusual, so the likelihood of you accidentally overwriting / shadowing one of them is low. However it is still not recommended even in this case.

Wrapping Up

Python has a wonderful standard library that you can use to build all kinds of interesting applications. In this chapter you learned how to import them into your own code. Once imported, you learned the basics of accessing the functions and variables that are in the newly imported code.

You also learned how to import parts of a library as well as how to import everything. There is a lot more to importing than what is covered in this book. You might want to check out my sequel to this book, **Python 201: Intermediate Python** which covers more about importing as well as the `importlib` library, which lets you do lots of other interesting things when importing.

Review Questions

1. How do you include the `math` library from Python's standard library in your code?
2. How do you include `cos` from the `math` library in your own code?
3. How do you import a module/function/etc. with a different name?
4. Python has a special syntax you can use to include everything. What is it?

Chapter 17 - Functions

Functions are reusable pieces of code. Anytime you find yourself writing the same code twice, that code should probably go in a function. You have actually used some functions in previous chapters.

For example, Python has many built-in functions, such as `dir()` and `sum()`. You also imported the `math` module and used its square root function, `sqrt()`.

In this chapter you will learn about:

- Creating a function
- Calling a function
- Passing arguments
- Type hinting your arguments
- Passing keyword arguments
- Required and default arguments
- `*args` and `**kwargs`
- Positional-only arguments
- Scope

Let's get started!

Creating a Function

A function starts with the keyword, `def` followed by the name of the function, two parentheses and then a colon. Next you indent one or more lines of code under the function to form the function “block”.

Here is an empty function:

```
1 def my_function():
2     pass
```

When you create a function, it is usually recommended that the name of the function is all lowercase with words separated by underscores. This is called **snake-case**.

The `pass` is a keyword in Python that Python knows to ignore. You can also define an empty function like this:

```
1 def my_function():
2     ...
```

In this example, the function has no contents besides an ellipses.

Let's learn how to use a function next!

Calling a Function

Now that you have a function, you need to make it do something. Let's do that first:

```
1 def my_function():
2     print('Hello from my_function')
```

Now instead of an ellipses or the keyword `pass`, you have a function that prints out a message.

To call a function, you need to write out its name followed by parentheses:

```
1 >>> def my_function():
2     ...     print('Hello from my_function')
3 ...
4 >>> my_function()
5 Hello from my_function
```

That was nice and easy!

Now let's learn about passing arguments to your functions.

Passing Arguments

Most functions let you pass arguments to them. The reason for this is that you will normally want to pass one or more positional arguments to a function so that the function can do something with them.

Let's create a function that takes an argument called `name` and then prints out a welcome message:

```
1 >>> def welcome(name):
2     ...     print(f'Welcome {name}')
3 ...
4 >>> welcome('Mike')
5 Welcome Mike
```

If you have used other programming languages, you might know that some of them require functions to return something. Python automatically returns `None` if you do not specify a return value.

Let's try calling the function and assigning its result to a variable called `return_value`:

```
1 >>> def welcome(name):
2     ...     print(f'Welcome {name}')
3 ...
4 >>> return_value = welcome('Mike')
5 Welcome Mike
6 >>> print(return_value)
7 None
```

When you print out the `return_value`, you can see that it is `None`.

Type Hinting Your Arguments

Some programming languages use static types so that when you compile your code, the compiler will warn you of type related errors. Python is a dynamically typed language, so that doesn't happen until run time

However, in Python 3.5, the `typing` module was added to Python to allow developers to add **type hinting** to their code. This allows you to specify the types of arguments and return values in your code, but does **not** enforce it. You can use external utilities, such as `mypy` (<http://mypy-lang.org/>) to check that your code base is following the type hints that you have set.

Type hinting is not required in Python and it is not enforced by the language, but it is useful when working with teams of developers, especially when the teams are made up of people who are unfamiliar with Python.

Let's rewrite that last example so that it uses type hinting:

```
1 >>> def welcome(name: str) -> None:
2     ...     print(f'Welcome {name}')
3 ...
4 >>> return_value = welcome('Mike')
5 Welcome Mike
6 >>> print(return_value)
7 None
```

This time when you put in the `name` argument, you end it with a colon (:) followed by the type that you expect. In this case, you expect a string type to be passed in. After that you will note the `-> None`: bit of code. The `->` is special syntax to indicate what the return value is expected to be. For this code, the return value is `None`.

If you want to return a value explicitly, then you can use the `return` keyword followed by what you wish to return.

When you run the code, it executes in exactly the same manner as before.

To demonstrate that type hinting is not enforced, you can tell Python to return an integer by using the `return` keyword:

```
1 >>> def welcome(name: str) -> None:
2     ...     print(f'Welcome {name}')
3     ...     return 5
4 ...
5 >>> return_value = welcome('Mike')
6 Welcome Mike
7 >>> print(return_value)
8 5
```

When you run this code, you can see the type hint says that the return value should be `None`, but you coded it such that it returns the integer 5. Python does not throw an exception.

You can use the `mypy` tool against this code to verify that it is following the type hinting. If you do so, you will find that it does show an issue. You will learn how to use `mypy` in Part II of this book.

The main takeaway here is that Python supports type hinting. Python does not enforce types though. However, some Python editors can use type hinting internally to warn you about issues related to types, or you can use `mypy` manually to find issues.

Now let's learn what else you can pass to a function.

Passing Keyword Arguments

Python also allows you to pass in keyword arguments. A keyword argument is specified by passing in a named argument, for example you might pass in `age=10`.

Let's create a new example that shows a regular argument and a single keyword argument:

```
1 >>> def welcome(name: str, age: int=15) -> None:
2     ...     print(f'Welcome {name}. You are {age} years old.')
3 ...
4 >>> welcome('Mike')
5 Welcome Mike. You are 15 years old.
```

This example has a regular argument, `name` and a keyword argument, `age`, which is defaulted to 15. When you call this code without specifying the `age`, you see that it defaults to 15.

To make things extra clear, here's a different way you can call it:

```
1 >>> def welcome(name: str, age: int) -> None:
2 ...     print(f'Welcome {name}. You are {age} years old.')
3 ...
4 >>> welcome(age=12, name='Mike')
5 Welcome Mike. You are 12 years old.
```

In this example, you specified both `age` and `name` parameters. When you do that, you can specify them in any order. For example, here you specified them in reverse order and Python still understood what you meant because you specified BOTH values.

Let's see what happens when you don't use keyword arguments:

```
1 >>> def welcome(name: str, age: int) -> None:
2 ...     print(f'Welcome {name}. You are {age} years old.')
3 ...
4 >>> welcome(12, 'Mike')
5 Welcome 12. You are Mike years old.
```

When you pass in values without specifying where they should go, they will be passed in order. So `name` becomes `12` and `age` becomes `'Mike'`.

Required and Default Arguments

Default arguments are a handy way to make your function callable with less arguments, whereas required arguments are ones that you have to pass in to the function for the function to execute.

Let's look at an example that has one required argument and one default argument:

```
1 >>> def multiply(x: int, y: int=5) -> int:
2 ...     return x * y
3 ...
4 >>> multiply(5)
5 25
```

The first argument, `x` is required. If you call `multiply()` without any arguments, you will receive an error:

```
1 >>> multiply()
2 Traceback (most recent call last):
3   Python Shell, prompt 25, line 1
4 builtins.TypeError: multiply() missing 1 required positional argument: 'x'
```

The second argument `y`, is not required. In other words, it is a default argument where the default is `5`. This allowed you to call `multiply()` with only one argument!

What are *args and **kwargs?

Most of the time, you will want your functions to only accept a small number of arguments, keyword arguments or both. You normally don't want too many arguments as it becomes more complicated to change your function later.

However Python does support the concept of *any number of arguments or keyword arguments*.

You can use this special syntax in your functions:

- *args - An arbitrary number of arguments
- **kwargs - An arbitrary number of keyword arguments

The bit that you need to pay attention to is the * and the **. The name, arg or kwarg can be anything, but it is a convention to name them args and kwargs. In other words, most Python developers call them *args or **kwargs. While you aren't forced to do so, you probably should so that the code is easy to recognize and understand.

Let's look at an example:

```
1 >>> def any_args(*args):  
2     ...     print(args)  
3     ...  
4 >>> any_args(1, 2, 3)  
5 (1, 2, 3)  
6 >>> any_args(1, 2, 'Mike', 4)  
7 (1, 2, 'Mike', 4)
```

Here you created any_args() which accepts any number of arguments including zero and prints them out.

You can actually create a function that has a required argument plus any number of additional arguments:

```
1 >>> def one_required_arg(required, *args):  
2     ...     print(f'{required=}')  
3     ...     print(args)  
4     ...  
5 >>> one_required_arg('Mike', 1, 2)  
6 required='Mike'  
7 (1, 2)
```

So in this example, your function's first argument is required. If you were to call one_required_arg() without any arguments, you would get an error.

Now let's try adding keyword arguments:

```
1 >>> def any_keyword_args(**kwargs):
2     ...     print(kwargs)
3 ...
4 >>> any_keyword_args(1, 2, 3)
5 Traceback (most recent call last):
6   Python Shell, prompt 7, line 1
7 builtins.TypeError: any_keyword_args() takes 0 positional arguments but 3 were given
```

Oops! You created the function to accept keyword arguments but only passed in normal arguments. This caused a `TypeError` to be thrown.

Let's try passing in the same values as keyword arguments:

```
1 >>> def any_keyword_args(**kwargs):
2     ...     print(kwargs)
3 ...
4 >>> any_keyword_args(one=1, two=2, three=3)
5 {'one': 1, 'two': 2, 'three': 3}
```

This time it worked the way you would expect it to.

Now let's inspect our `*args` and `**kwargs` and see what they are:

```
1 >>> def arg_inspector(*args, **kwargs):
2     ...     print(f'args are of type {type(args)}')
3     ...     print(f'kwargs are of type {type(kwargs)}')
4 ...
5 >>> arg_inspector(1, 2, 3, x='test', y=5)
6 args are of type <class 'tuple'>
7 kwargs are of type <class 'dict'>
```

What this means is that `args` is a tuple and `kwargs` are a dict.

Let's see if we can pass our function a tuple and dict for the `*args` and `**kwargs`:

```
1 >>> my_tuple = (1, 2, 3)
2 >>> my_dict = {'one': 1, 'two': 2}
3 >>> def output(*args, **kwargs):
4 ...     print(f'{args=}')
5 ...     print(f'{kwargs=}')
6 ...
7 >>> output(my_tuple)
8 args=((1, 2, 3),)
9 kwargs={}
10 >>> output(my_tuple, my_dict)
11 args=((1, 2, 3), {'one': 1, 'two': 2})
12 kwargs={}
```

Well that didn't work quite right. Both the `tuple` and the `dict` ended up in the `*args`. Not only that, but the `tuple` stayed a tuple instead of being turned into three arguments.

You can make this work if you use a special syntax though:

```
1 >>> def output(*args, **kwargs):
2 ...     print(f'{args=}')
3 ...     print(f'{kwargs=}')
4 ...
5 >>> output(*my_tuple)
6 args=(1, 2, 3)
7 kwargs={}
8 >>> output(**my_dict)
9 args=()
10 kwargs={'one': 1, 'two': 2}
11 >>> output(*my_tuple, **my_dict)
12 args=(1, 2, 3)
13 kwargs={'one': 1, 'two': 2}
```

In this example, you call `output()` with `*my_tuple`. Python will extract the individual values in the `tuple` and pass each of them in as arguments. Next you passed in `**my_dict`, which tells Python to pass in each key/value pair as keyword arguments.

The final example passes in both the `tuple` and the `dict`.

Pretty neat!

Positional-only Parameters

Python 3.8 added a new feature to functions known as **positional-only parameters**. These use a special syntax to tell Python that some parameters have to be positional and some have to be keyword.

Let's look at an example:

```
1 >>> def positional(name, age, /, a, b, *, key):
2 ...     print(name, age, a, b, key)
3 ...
4 >>> positional(name='Mike')
5 Traceback (most recent call last):
6   Python Shell, prompt 21, line 1
7 builtins.TypeError: positional() got some positional-only arguments passed as
8 keyword arguments: 'name'
```

The first two parameters, `name` and `age` are positional-only. They can't be passed in as keyword arguments, which is why you see the `TypeError` above. The arguments, `a` and `b` can be positional or keyword. Finally, `key`, is keyword-only.

The forward slash, `/`, indicates to Python that all arguments before the forward slash as positional-only arguments. Anything following the forward slash are positional or keyword arguments up to the `*`. The asterisk indicates that everything following it as keyword-only arguments.

Here is a valid way to call the function:

```
1 >>> positional('Mike', 17, 2, b=3, keyword='test')
2 Mike 17 2 3 test
```

However if you try to pass in only positional arguments, you will get an error:

```
1 >>> positional('Mike', 17, 2, 3, 'test')
2 Traceback (most recent call last):
3   Python Shell, prompt 25, line 1
4 builtins.TypeError: positional() takes 4 positional arguments but 5 were given
```

The `positional()` function expects the last argument to be a keyword argument.

The main idea is that positional-only parameters allow the parameter name to change without breaking client code.

You may also use the same name for positional-only arguments and `**kwargs`:

```
1 >>> def positional(name, age, /, **kwargs):
2 ...     print(f'{name=}')
3 ...     print(f'{age=}')
4 ...     print(f'{kwargs=}')
5 ...
6 >>> positional('Mike', 17, name='Mack')
7 name='Mike'
8 age=17
9 kwargs={'name': 'Mack'}
```

You can read about the full implementation and reasoning behind the syntax here:

- <https://www.python.org/dev/peps/pep-0570>

Let's move on and learn a little about the topic of scope!

Scope

All programming languages have the idea of scope. Scope tells the programming language what variables or functions are available to them.

Let's look at an example:

```
1 >>> name = 'Mike'
2 >>> def welcome(name):
3 ...     print(f'Welcome {name}')
4 ...
5 >>> welcome()
6 Traceback (most recent call last):
7   Python Shell, prompt 34, line 1
8 builtins.TypeError: welcome() missing 1 required positional argument: 'name'
9 >>> welcome('Nick')
10 Welcome Nick
11 >>> name
12 'Mike'
```

The variable `name` is defined outside of the `welcome()` function. If you try to call `welcome()` without passing it an argument, it throws an error even though the argument matches the variable `name`. If you pass in a value to `welcome()`, that variable is only changed inside of the `welcome()` function. The `name` that you defined outside of the function remains unchanged.

Let's look at an example where you define variables inside of functions:

```
1  >>> def add():
2  ...     a = 2
3  ...     b = 4
4  ...     return a + b
5  ...
6  >>> def subtract():
7  ...     a = 3
8  ...     return a - b
9  ...
10 >>> add()
11 6
12 >>> subtract()
13 Traceback (most recent call last):
14   Python Shell, prompt 40, line 1
15     Python Shell, prompt 38, line 3
16 builtins.NameError: name 'b' is not defined
```

In `add()`, you define `a` and `b` and add them together. The variables, `a` and `b` have **local** scope. That means they can only be used within the `add()` function.

In `subtract()`, you only define `a` but try to use `b`. Python doesn't check to see if `b` exists in the `subtract()` function until runtime.

What this means is that Python does not warn you that you are missing something here until you actually call the `subtract()` function. That is why you don't see any errors until there at the end.

Python has a special `global` keyword that you can use to allow variables to be used across functions.

Let's update the code and see how it works:

```
1  >>> def add():
2  ...     global b
3  ...     a = 2
4  ...     b = 4
5  ...     return a + b
6  ...
7  >>> def subtract():
8  ...     a = 3
9  ...     return a - b
10 ...
11 >>> add()
12 6
13 >>> subtract()
14 -1
```

This time you define `b` as `global` at the beginning of the `add()` function. This allows you to use `b` in `subtract()` even though you haven't defined it there.

Globals are usually not recommended. It is easy to overlook them in large code files, which makes tracking down subtle errors difficult – for example, if you had called `subtract()` before you called `add()` you would still get the error, because even though `b` is `global`, it doesn't exist until `add()` has been run.

In most cases where you would want to use a `global`, you can use a `class` instead. You will learn about classes in the next chapter.

There is nothing wrong with using globals as long as you understand what you are doing. They can be helpful at times. But you should use them with care.

Wrapping Up

Functions are a very useful way to reuse your code. They can be called repeatedly. Functions allow you to pass and receive data too.

In this chapter, you learned about the following topics:

- Creating a function
- Calling a function
- Passing arguments
- Type hinting your arguments
- Passing keyword arguments
- Required and default arguments
- `*args` and `**kwargs`
- Positional-only arguments
- Scope

You can use functions to keep your code clean and useful. A good function is self-contained and can be used easily by other functions. While it isn't covered in this chapter, you can nest functions inside of each other. You will learn about one good use-case for that in **Part II** of the book.

Review Questions

1. How do you create a function that accepts two positional arguments?
2. Create a function named `address_builder` that accepts the following and add **type hints**:
 - `name (string)`

- address (string)
 - zip code (integer)
3. Using the function from question 2, give the **zip code** a default of 55555
 4. How do you allow an arbitrary number of keyword arguments to be passed to a function?
 5. What syntax do you use to force a function to use positional-only parameters?

Chapter 18 - Classes

Everything in Python is an object. What that means is that everything you create in Python has functions or attributes or both attached to them that you can use. This is because everything in Python comes from a class.

Take a string, for example:

```
1 >>> name = 'Mike'
2 >>> dir(name)
3 ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
4 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
5 '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
6 '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
7 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
8 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
9 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
10 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
11 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
12 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
13 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
14 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
15 'title', 'translate', 'upper', 'zfill']
```

There are 78 methods and attributes associated with a `str` in Python. You covered some of these back in [chapter 9](#) when you were learning about strings.

Python also lets you create your own custom objects using classes. This lets you make your own methods and attributes for your object, so you can make your code do whatever you want!

Note, however, that while Python has powerful class support, also known as Object Oriented Programming (OOP), it is also easily usable as a functional language (i.e. no classes). It is up to you how you use Python!

In this chapter you will learn:

- Class creation
- `self` – what it means
- Public and private methods / attributes
- Subclass creation
- Polymorphism

Let's get started by creating a class!

Class Creation

A class is a blueprint that Python uses to build your object. A popular example to use is a ball.

A ball has the following attributes:

- Size
- Weight
- Color

Balls can also have actions. They can *do* something. Those actions are called **methods** in a class. A method is just a function inside of a class.

Typical ball methods would be:

- Roll
- Bounce
- Toss

To create a class, you will need to use Python's built-in keyword: `class` followed by the name of the class and then a colon. It is conventional to name classes using camel-case: `MyClass`. At the very least, the first letter should be capitalized.

Here's how you create an empty `Ball` class:

```
1 class Ball:  
2     pass
```

This class doesn't do anything. In programming circles, you might hear this class referred to as a **stub**.

Let's write a class that has the 3 attributes mentioned earlier:

```
1 class Ball:  
2  
3     def __init__(self, color, size, weight):  
4         """Initializer"""  
5         self.color = color  
6         self.size = size  
7         self.weight = weight
```

This class takes three arguments to create it:

- color
- size
- weight

When you call a class, you are creating an *instance* of that class. Here is an example:

```
1 >>> class Ball:  
2 ...  
3 ...     def __init__(self, color, size, weight):  
4 ...         self.color = color  
5 ...         self.size = size  
6 ...         self.weight = weight  
7 ...  
8 >>> beach_ball = Ball('red', 15, 1)  
9 >>> print(beach_ball)  
10 <__main__.Ball object at 0x101f5a040>  
11 >>> print(f'My ball is {beach_ball.color} and weighs {beach_ball.weight} lb')  
12 My ball is red and weighs 1 lb
```

To create an instance of a `Ball`, you call it by passing in the 3 parameters mentioned earlier. This works in exactly the same way as it did with functions. However in a class, you have functions that are known as methods. The other difference is that to call a class, you usually need to have a special method called `__init__`.

This is called an **initializer**. You use `__init__()` to initialize your created object. `__init__()` is typically only called once, when you create the instance of the class. Along with the initializer, `__init__`, there is the **constructor**, `__new__`. You will rarely, if ever, need to specify `__new__` and can just use the default provided by Python.

When you print out the `beach_ball`, you can see that Python tells you it is a `Ball` object, but not much else. You will learn how to make that more useful later on in the chapter. You can also print out the attributes you set, which is shown above in the last two lines of code.

If you want to add type hinting to your class, this is how you would do that:

```
1 >>> class Ball:  
2 ...  
3 ...     def __init__(self, color: str, size: float, weight: float) -> None:  
4 ...         self.color = color  
5 ...         self.size = size  
6 ...         self.weight = weight
```

You may have noticed that the first argument to `__init__()` is something called `self`.

Let's find out what that's about!

Figuring Out `self`

When you create a class, you need a way to keep track of the instances that you create of it. For example, if you create two balls with unique characteristics, you don't want the second ball object to overwrite the first one.

Python classes use the `self` argument to keep track of which instance is which. If you have programmed in Java or a Java-like programming language, they call their `self` argument `this`.

This is easier to understand if you can see it in action:

```
1  >>> class Ball:  
2  ...  
3  ...     def __init__(self, color: str, size: float, weight: float) -> None:  
4  ...         self.color = color  
5  ...         self.size = size  
6  ...         self.weight = weight  
7  >>> bowling_ball = Ball('black', 6, 12)  
8  >>> beach_ball = Ball('red', 12, 1)  
9  >>> id(bowling_ball)  
10 4327842432  
11 >>> id(beach_ball)  
12 4327842576
```

The quickest way to tell if you overwrote an instance of a class is to see if the instance IDs match. Here it is shown that the IDs differ. That is one way to tell that you have different objects.

Another easy way to tell is to access the attributes of the class:

```
1  >>> bowling_ball.color  
2  'black'  
3  >>> beach_ball.color  
4  'red'
```

The `self` is basically replaced by the name of the instance. So `bowling_ball` is the `self` for that instance while `beach_ball` is the `self` in its own instance.

In other words, `self` is used by Python to keep track of which instance is which.

This becomes even more clear if you add a method to the class.

Let's create a file named `ball.py` and add the following code to it:

```
1 # ball.py
2
3 class Ball:
4
5     def __init__(self, color: str, size: float, weight: float,
6                  ball_type: str) -> None:
7         self.color = color
8         self.size = size
9         self.weight = weight
10        self.ball_type = ball_type
11
12    def bounce(self):
13        if self.ball_type.lower() == 'bowling':
14            print("Bowling balls can't bounce!")
15        else:
16            print(f"The {self.ball_type} ball is bouncing!")
17
18 if __name__ == "__main__":
19     ball_one = Ball('black', 6, 12, 'bowling')
20     ball_two = Ball('red', 12, 1, 'beach')
21
22     ball_one.bounce()
23     ball_two.bounce()
```

You added a new argument to `__init__()` called `ball_type` to help you keep track of the type of ball you are creating. You also created a new method, `bounce()`.

Next you wrote some code at the bottom inside a strange-looking `if` block. When you want to run code when you run `python ball.py`, you should put it inside of these special `if` blocks:

```
1 if __name__ == '__main__':
2     # code goes here
```

When you run a module directly, the name of the module is set to `__main__`. You can check the name of the module via the special module attribute, `__name__`. That is what this special syntax is for. If you happened to import your `ball.py` script into another Python script, the name would be set to something else because it is no longer the main module.

Let's run this script. Open up a terminal (or cmd.exe on Windows) and navigate to the folder where you saved `ball.py`.

Then run the following command:

```
1 python ball.py
```

This should output the following:

```
1 Bowling balls can't bounce!
2 The beach ball is bouncing!
```

Your code demonstrates that Python is using `self` to keep track of which `Ball` instance is which. When you call `bounce()`, you will note that you can check the instance's `ball_type` by pre-pending it with `self`, which looks like `self.ball_type`. When you call the `bounce()` method outside of the class though, you call it with the instance name. For example, `ball_one.bounce()`.

Note that the word `self` is just a convention, albeit nearly universally followed. This is important because Python will assign the instance to the first parameter given, whether it's called `self`, `inst`, `this`, or `xyz` – in other words, if you forget to specify a parameter for `self` (or whatever spelling you like) you will get some weird errors.

Public and Private Methods / Attributes

Most programming languages have the concept of **public** and **private**. A public attribute or method is visible to all of Python. What that means is that when you create an instance of a class, you can access all of that class's public methods. A private method or attribute can only be used directly within the class where they are defined.

There is a third category called **protected**. Protected methods can only be seen inside the class they were defined in or in a sub-class (see the next section for more info on subclasses).

Python does not really have the concept of private or protected. Everything is effectively public in Python.

However there is a convention that if something should be private in Python, you should begin that method or attribute with a single or double underscore. This signals to other developers that that method or attribute should be treated as a private one and not used outside of that class.

When you see a method or attribute that has leading and ending double underscore, such as `__init__()`, then that is considered to be a “magic method”. They are also sometimes referred to as “dunder” methods, for double-underscore methods.

These magic methods help define the way Python data model works. Classes in general are where you will work with the concept of object oriented programming (OOP). Once you start using classes, you are doing OOP and you are working with Python's data model.

You can read about Python's built-in magic methods at the following:

- <https://docs.python.org/3/reference/datamodel.html>

You can use these methods to add “magic” to your classes. For example, you can create objects that can do arithmetic, check if they are greater than or less, and much more. You should check out the documentation or go digging on the Internet. There are several good tutorials on this topic out there.

Subclass Creation

Once you have a class, you can take that class and create a subclass. A subclass inherits the attributes and methods of the class it is based on.

You can also override the parent’s attributes and methods. What that means is that if the child has a method or attribute that is named the same as a method/attribute in the parent, Python will use the child’s instead.

Let’s take our `Ball` class from earlier and use it as an example to subclass from:

```
1 # ball.py
2
3 class Ball:
4
5     def __init__(self, color: str, size: float, weight: float,
6                  ball_type: str) -> None:
7         self.color = color
8         self.size = size
9         self.weight = weight
10        self.ball_type = ball_type
11
12    def bounce(self):
13        if self.ball_type.lower() == 'bowling':
14            print("Bowling balls can't bounce!")
15        else:
16            print(f"The {self.ball_type} ball is bouncing!")
```

Now let’s subclass `Ball` and create a new module called `bowling_ball.py` in the same folder as `ball.py`:

```
1 # bowling_ball.py
2 import ball
3
4 class BowlingBall(ball.Ball):
5
6     def roll(self):
7         print(f'You are rolling the {self.ball_type} ball')
8
9 if __name__ == '__main__':
10    ball = BowlingBall()
11    ball.roll()
```

To subclass Ball, you write the following: `class BowlingBall(ball.Ball)`. This subclasses Ball and creates a new class, BowlingBall.

This class does not have an `__init__()` method defined. That means that you will inherit the parent's `__init__()` and use it instead. You also added a new method, `roll()` that the parent class does not have.

Now let's try to run this code in your terminal:

```
1 python3 bowling_ball.py
2 Traceback (most recent call last):
3   File "bowling_ball.py", line 10, in <module>
4     ball = BowlingBall()
5   TypeError: __init__() missing 4 required positional arguments: 'color', 'size', 'wei\
6     ght', and 'ball_type'
```

Whoops! We didn't create the `BowlingBall` instance correctly. As alluded to above, `BowlingBall` is using `Ball`'s `__init__` method, so we have to supply the arguments that `Ball`'s `__init__` is expecting.

Let's fix the code so it works properly:

```
1 # bowling_ball.py
2 import ball
3
4 class BowlingBall(ball.Ball):
5
6     def roll(self):
7         print(f'You are rolling the {self.ball_type} ball')
8
9 if __name__ == '__main__':
10    ball = BowlingBall('green', 10, 15, 'bowling')
11    ball.roll()
```

Now when you run it, you should get the following output

```
1 You are rolling the bowling ball
```

Try going back and editing the code again to add a call to the `bounce()` method and make sure it works for you as well.

Polymorphism

Basing a class on another class is known as **inheritance**. This is also a very basic form of **polymorphism**. Polymorphic classes have a shared, common interface (methods and attributes) possibly from their parents via inheritance.

While you can make your classes more rigid by using **Abstract Base Classes** via Python's `abc` module, you will usually use the concept of **duck-typing** instead. The big idea behind duck-typing is that if it walks like a duck and talks like a duck, it can be treated like a duck.

What that means is that if a Python class has the same interface as its parent or similar class, then it doesn't matter all that much if the implementation underneath is different!

Making the Class Nicer

Do you remember how when you printed out an instance of your class? If you don't, the result looked like this:

```
1 <__main__.Ball object at 0x101f5a040>
```

That tells you what class the instance is and the hexadecimal value at the end tells you the memory location of the object. However this isn't exactly useful. It would be nice if it printed out the `ball_type` and maybe the `color` of the ball.

Let's update your code using a different "magic method" that will help make the printing more useful.

Copy the code from `ball.py` into a new file named `ball_printable.py` and then edit it like this:

```
1 # ball_printable.py
2
3 class Ball:
4
5     def __init__(self, color: str, size: float, weight: float,
6                  ball_type: str) -> None:
7         self.color = color
8         self.size = size
9         self.weight = weight
10        self.ball_type = ball_type
11
12    def bounce(self):
13        if self.ball_type.lower() == 'bowling':
14            print("Bowling balls can't bounce!")
15        else:
16            print(f"The {self.ball_type} ball is bouncing!")
17
18    def __repr__(self):
19        return f"<Ball: {self.color} {self.ball_type} ball>"
20
21
22 if __name__ == "__main__":
23     ball_one = Ball('black', 6, 12, 'bowling')
24     ball_two = Ball('red', 12, 1, 'beach')
25
26     print(ball_one)
27     print(ball_two)
```

Here you added the new `__repr__()` magic method, which you can use to create a nice string representation of the object when it is printed. When you go to print an instance of an object, Python will look to see if you have either `__repr__()` or `__str__()` defined. Most of the time, it is recommended to use `__repr__()` for developers as a debugging tool.

When you run this code, it will output the following:

```
1 <Ball: black bowling ball>
2 <Ball: red beach ball>
```

If you go ahead and implement `__str__()` as well, it will actually replace `__repr__()` when you print.

Here is an example:

```
1 # ball_printable.py
2
3 class Ball:
4
5     def __init__(self, color: str, size: float, weight: float,
6                  ball_type: str) -> None:
7         self.color = color
8         self.size = size
9         self.weight = weight
10        self.ball_type = ball_type
11
12    def bounce(self):
13        if self.ball_type.lower() == 'bowling':
14            print("Bowling balls can't bounce!")
15        else:
16            print(f'The {self.ball_type} ball is bouncing!')
17
18    def __repr__(self):
19        return f'<Ball: {self.color} {self.ball_type} ball>'
20
21    def __str__(self):
22        return f'{self.color} {self.ball_type} ball'
23
24
25 if __name__ == "__main__":
26     ball_one = Ball('black', 6, 12, 'bowling')
27     ball_two = Ball('red', 12, 1, 'beach')
28
29     print(ball_one)
30     print(ball_two)
31
32     print(f'{ball_one.__repr__()}' )
33     print(f'{ball_one.__str__()}' )
```

In this code, you added the `__str__()` method and had it return a very similar string to `__repr__()` except that it doesn't add the class name to the string. Then to show the difference, you print out the ball objects. Finally you print out the the result of calling `__repr__()` and `__str__()` directly.

This is what you should get as output:

```
1 black bowling ball
2 red beach ball
3 <Ball: black bowling ball>
4 black bowling ball
```

This only scratches the surface of what you can do with Python's magic methods.

Wrapping Up

You covered a lot of valuable information in this chapter. You learned about:

- Class creation
- `self` – what it means
- Public and private methods / attributes
- Subclass creation
- Polymorphism

You also learned how to use some basic magic methods in Python to modify how an object will behave. You can use the knowledge from this chapter to design your own objects that are as simple or as complex as you need. You can create nice parent classes to use as a template for subclasses too.

As you write your code, always be on the lookout for how you might make it simpler and easier to understand. Use good comments, docstrings, and names for functions and attributes and you will be well on your way to having some great self-documenting code!

Review Questions

1. How do you create a class in Python?
2. What do you name a class initializer?
3. Explain the use of `self` in your own words
4. What does overriding a method do?
5. What is a subclass?

Part II - Beyond the Basics

The second section of this book features chapters on intermediate level material. Here you will learn about topics that will help you become a better developer.

In this section of the book, you will cover the following:

- Chapter 19 - Introspection
- Chapter 20 - Installing Packages with `pip`
- Chapter 21 - Python Virtual Environments
- Chapter 22 - Type Hinting
- Chapter 23 - Threading
- Chapter 24 - Multiprocessing
- Chapter 25 - Launching Subprocesses with Python
- Chapter 26 - Debugging Your Code
- Chapter 27 - Decorators
- Chapter 28 - Assignment Expressions
- Chapter 29 - Profiling Your Code
- Chapter 30 - An Introduction to Testing Your Code
- Chapter 31 - Jupyter Notebook

When you have finished Part II, you will have many new tools at your fingertips to use in your projects.

Let's start learning about them now!

Chapter 19 - Introspection

One of the reasons Python is special is that it provides lots of tools that allow you to learn about Python itself. When you learn about yourself, it is called **introspection**. There is a similar type of introspection that can happen with programming languages.

Python provides several built-in functions that you can use to learn about the code that you are working with.

In this chapter, you will learn how to use:

- The `type()` function
- The `dir()` function
- The `help()` function
- Other useful built-in tools

Let's get started!

Using the `type()` Function

When you are working with unfamiliar code, it can be useful to check and see what type it is. Is the code a string, an integer or some kind of object? This is especially true when you are working with code that you have never used before, be it a new module, package, or business application.

Let's look at an example:

```
1 def multiplier(x, y):  
2     return str(x * y)
```

You will encounter lots of code that has no documentation or documentation that is incorrect. This is an example of a function that appears to multiply numbers. It does in fact do that, but notice that instead of returning a numeric value, it returns a string.

This is where using the `type()` function can be valuable. Let's try calling this code and checking the return type:

```
1 >>> return_value = multiplier(4, 5)
2 >>> print(return_value)
3 20
4 >>> type(return_value)
5 <class 'str'>
```

Looking at the code, you would expect a function named `multiplier()` to take some numeric types and also return a numeric result, and when you print out the result, it looks numeric – however, it is a string! You will actually run into this issue quite often and it can lead to some weird errors.

Using the `dir()` Function

There are thousands of Python packages that you can install to get new features in Python. You will learn how to do that in [chapter 20](#). However the Python standard library has lots of built-in modules that you may not be familiar with either.

You can use Python's built-in `dir()` function to learn more about a module that you are unfamiliar with.

When you run `dir()`, it will return a list of all the functions and attributes that are available to you.

Let's try using it on the `code` module:

```
1 >>> import code
2 >>> dir(code)
3 ['CommandCompiler', 'InteractiveConsole', 'InteractiveInterpreter', '__all__',
4 '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
5 '__package__', '__spec__', 'compile_command', 'interact', 'sys', 'traceback']
```

Most people aren't familiar with the `code` module. You use it to create read-eval-print loops in Python. One way to learn how you might use this mysterious module is to use the `dir()` function. Sometimes by reading the module's function names, you can deduce what it does. In this example, you can see that there is a way to `compile_command` and `interact`.

Those are clues to what the library does. But since you still aren't sure what they do, you can always ask for `help()`.

Getting `help()`

Python also provides the useful `help()` function. The `help()` function will read the docstrings in the module and print them out. So if the developer added good documentation to their code, then you can use Python to introspect it.

Let's give it a try using the `code` module:

```
1 >>> help(code)
2 Help on module code:
3
4 NAME
5     code - Utilities needed to emulate Python's interactive interpreter.
6
7 MODULE REFERENCE
8     https://docs.python.org/3.8/library/code
9
10    The following documentation is automatically generated from the Python
11    source files. It may be incomplete, incorrect or include features that
12    are considered implementation detail and may vary between Python
13    implementations. When in doubt, consult the module reference at the
14    location listed above.
15
16 CLASSES
17     builtins.object
18         InteractiveInterpreter
19             InteractiveConsole
20
21     class InteractiveConsole(InteractiveInterpreter)
22         | InteractiveConsole(locals=None, filename='<console>')
23         |
24         | Closely emulate the behavior of the interactive Python interpreter.
```

This example shows the first page of help that you get when you run `help(code)`. You can **spacebar** to page through the documentation or **return** to advance a line at a time. If you want to exit help, press **q**.

Let's try getting help on the `interact()` function:

```
1 >>> help(code.interact)
2
3 Help on function interact in module code:
4
5 interact(banner=None, readfunc=None, local=None, exitmsg=None)
6     Closely emulate the interactive Python interpreter.
7
8     This is a backwards compatible interface to the InteractiveConsole
9     class. When readfunc is not specified, it attempts to import the
10    readline module to enable GNU readline if it is available.
11
12    Arguments (all optional, all default to None):
```

```
13
14     banner -- passed to InteractiveConsole.interact()
15     readfunc -- if not None, replaces InteractiveConsole.raw_input()
16     local -- passed to InteractiveInterpreter.__init__()
17     exitmsg -- passed to InteractiveConsole.interact()
18 (END)
```

You can use `dir()` to get information on the methods and attributes in an unfamiliar module. Then you can use `help()` to get more details about the module or any of its components.

Other Built-in Introspection Tools

Python includes several other tools you can use to help you figure out the code.

Here are a few more helpful built-in functions:

- `callable()`
- `len()`
- `locals()`
- `globals()`

Let's go over these and see how you might use them!

Using `callable()`

The `callable()` function is used to test if an object is callable. What does that mean though?

A callable is something in Python that you can call to get a result from, such as a function or class. When you are dealing with an unknown module, you can't always tell what's a variable or a function or a class. The `help()` function from before will tell you, but if you don't want to go into help mode, using `callable()` might be the way to go.

Here are some examples:

```
1 >>> a = 5
2 >>> callable(a)
3 False
4 >>> def adder(x, y):
5 ...     pass
6 ...
7 >>> callable(adder)
8 True
9 >>> class Ball:
10 ...     pass
11 ...
12 >>> callable(Ball)
13 True
```

The first example shows that a variable is not a callable object. So you can say with good confidence that the variable, `a`, is quite probably a variable. The next two examples are testing a function, `adder`, and a class, `Ball`, with `callable()`. Both of these return `True` because they can both be called. Of course, when you “call” a class, you are instantiating it, so it’s not quite the same as a function.

Using `len()`

The `len()` function is useful for finding the length of an object. You will probably end up using it a lot for debugging whether or not a string or list is as long as it ought to be.

Here are a couple of examples:

```
1 >>> len('abcd')
2 4
3 >>> len([1, 'two', 3])
4 3
```

You can use `len()` on many different types of objects in Python.

However, there are times where it won’t work, such as with integers:

```
1 >>> a = 4
2 >>> len(a)
3 Traceback (most recent call last):
4   Python Shell, prompt 15, line 1
5 builtins.TypeError: object of type 'int' has no len()
```

Using `locals()`

The `locals()` function returns a dictionary of the current local symbol table. What that means is that it tells you what is currently available to you in your current context, or scope.

Let's look at an example. Open up the Python REPL by running `python3` in your terminal. Then run the following:

```
1 >>> locals()
2 { '__name__': '__main__', '__doc__': None, '__package__': None,
3  '__loader__': <class '__frozen_importlib.BuiltinImporter'>, '__spec__': None,
4  '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

This tells you what is currently in your namespace, or scope. These aren't especially useful though. Let's create a couple of variables and re-run the command:

```
1 >>> a = 3
2 >>> b = 4
3 >>> locals()
4 { '__name__': '__main__', '__doc__': None, '__package__': None,
5  '__loader__': <class '__frozen_importlib.BuiltinImporter'>, '__spec__': None,
6  '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
7  'a': 3, 'b': 4}
```

Now you can see that the variables you created are included in the dictionary.

Using `locals()` becomes more useful when you use it inside of a function:

```
1 >>> def adder(x, y):
2 ...     print(locals())
3 ...
4 >>> adder(4, 5)
5 { 'x': 4, 'y': 5}
```

This lets you see what arguments were passed into a function as well as what their values were.

Using `globals()`

The `globals()` function is quite similar to the `locals()` function; the difference is that `globals()` always returns the module-level namespace, while `locals()` returns the current namespace. In fact, if you run them both at the module level (i.e. outside of any function or class), the dictionaries that they return will match.

Let's try calling `globals()` inside `adder()`:

```
1  >>> def adder(x, y):
2      ...     print(globals())
3  ...
4  >>> adder(4, 5)
5  {'__name__': '__main__', '__doc__': None, '__package__': None,
6  '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x10ef3db80>,
7  '__spec__': None, '__builtins__': <module 'builtins' (built-in)>,
8  'adder': <function adder at 0x10ead00d0>}
```

As you can see, when you run this code you get the module level attributes, functions, etc., included in the dictionary, but you do **not** get the arguments to the function, nor their values.

Wrapping Up

Python has a rich set of tools that you can use to learn about your code and other people's code. In this chapter, you learned about the following topics:

- The `type()` function
- The `dir()` functions
- The `help()` function
- Other useful built-in tools

You can use these tools together to figure out how to use most Python packages. There will be a few that are not well documented both on their website and internally too. Those kinds of packages should be avoided as you may not be able to ascertain what they are doing.

Review Questions

1. What is introspection?
2. What is the `type()` function used for?
3. How is `dir()` helpful?

Chapter 20 - Installing Packages with pip

Python's standard library is amazing. You can build all kinds of different programs using nothing but what comes with Python. However, there is an entire ecosystem of additional Python packages that you can install. The vast majority of these packages can be found on the Python Package Index (PyPI) or Github.

You can browse PyPI here:

- <https://pypi.org/>

You can think of packages as enhancements. They add new functionality to your code. For example, you might want to add computer vision to your application. You could go down the tedious road of writing it yourself or you could install `opencv-python` which is a wrapper around the popular OpenCV package.

In this chapter, you will learn about the following:

- Installing a Package
- Exploring Command Line Options
- Installing with `requirements.txt`
- Upgrading a Package
- Checking What's Installed
- Uninstalling Packages
- Alternatives to pip

Let's get started!

Installing a Package

Starting with Python 3.4, the `pip` installer program was included with the Python programming language. Pip is also a module in Python. There are cases where the system Python does not have `pip` installed with it. The system Python is the Python version that came with your operating system.

You can check if you have it as a module like this:

```
1 python3 -m pip
```

The `-m` command line option tells Python that you are going to run a Python library directly. If this works correctly, you will see a lot of output printed to your screen detailing what commands you can use with `pip`. Most modules in Python allow you to import them as well. In `pip`'s case, it is recommended to call it using `pip`'s command line interface.

On the off chance that you don't have `pip` installed, you can get it from here:

- <https://pip.pypa.io/en/stable/installing/>

Once you have `pip`, you can install a package by opening your terminal and running the following:

```
1 pip install package_name
```

This assumes that `pip` is installed as an application and is on the path. The disadvantage of this approach is that you cannot tell which Python version this `pip` is attached to. So if you have Python 3.6 and Python 3.8 installed, the command above might be installing your package to 3.6 rather than the latest version.



What is the “Path”?

When talking about the “path”, operating systems have a way of knowing what application to run when you open a terminal and type a command like `python` or `cmd.exe`.

If `pip` has been installed correctly on your operating system, then you should be able to call `pip` without specifying its install location too. If that doesn't work though, then you will need to use Google to find out how to add an application to your path as each operating system does it differently.

To specify which Python you want to install a package to, you can do this:

```
1 python3.8 -m pip install package_name
```

Here you are specifying that you want to install a package to **Python 3.8**. On Windows, you may need to enter the full path to Python 3.8 rather than using `python3.8`.

Whether or not you are on Windows, when you run this command, `pip` will go looking for the package you specified from the Python Package Index, download the file and install it.

Once your package is installed, you should be able to import it and use it as you would any other Python library.

You can install multiple packages all at once by separating the names of the package with spaces:

```
1 python -m pip install package_1 package_2 package_3
```

Now let's learn about the command line options you can use with pip!

Exploring Command Line Options

The pip utility has several commands you can use.

Here are the current commands for the version that came with Python 3.8:

```
1 Commands:  
2     install           Install packages.  
3     download          Download packages.  
4     uninstall         Uninstall packages.  
5     freeze            Output installed packages in requirements format.  
6     list               List installed packages.  
7     show               Show information about installed packages.  
8     check              Verify installed packages have compatible dependencies.  
9     config             Manage local and global configuration.  
10    search             Search PyPI for packages.  
11    wheel              Build wheels from your requirements.  
12    hash               Compute hashes of package archives.  
13    completion         A helper command used for command completion.  
14    debug              Show information useful for debugging.  
15    help               Show help for commands.
```

There are some general options (i.e. command line options) that you can use as well:

```
1 General Options:  
2     -h, --help          Show help.  
3     --isolated          Run pip in an isolated mode, ignoring  
4                           environment variables and user configuration.  
5     -v, --verbose        Give more output. Option is additive, and can be  
6                           used up to 3 times.  
7     -V, --version        Show version and exit.  
8     -q, --quiet          Give less output. Option is additive, and can be  
9                           used up to 3 times (corresponding to WARNING,  
10                          ERROR, and CRITICAL logging levels).  
11    --log <path>        Path to a verbose appending log.  
12    --proxy <proxy>      Specify a proxy in the form  
13                          [user:passwd@]proxy.server:port.  
14    --retries <retries> Maximum number of retries each connection should
```

```
15                      attempt (default 5 times).
16  --timeout <sec>      Set the socket timeout (default 15 seconds).
17  --exists-action <action> Default action when a path already exists:
18                                (s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort.
19  --trusted-host <hostname> Mark this host as trusted, even though it does
20                                not have valid or any HTTPS.
21  --cert <path>          Path to alternate CA bundle.
22  --client-cert <path>    Path to SSL client certificate, a single file
23                                containing the private key and the certificate
24                                in PEM format.
25  --cache-dir <dir>       Store the cache data in <dir>.
26  --no-cache-dir          Disable the cache.
27  --disable-pip-version-check
28                                Don't periodically check PyPI to determine
29                                whether a new version of pip is available for
30                                download. Implied with --no-index.
31  --no-color              Suppress colored output
```

This chapter won't cover all the options that pip supports. Instead, it will focus on a small handful of the ones that you are most likely to use.

You can get additional information about options by running `-h` after a command. For example:

```
1 python -m pip install -h
```

This will show **Install Options** and **Package Index Options**.

Installing with requirements.txt

The pip install utility will automatically attempt to install all dependencies of the package. The dependencies are specified in a file called `requirements.txt`. You can install all the dependencies of the package using the `requirements.txt` file directly:

```
1 python -m pip install -r requirements.txt
```

The requirements file format requires that each package that is required be put on its own line. You can specify a version, or range of versions, to install; if you do not, then the latest version will be installed.

Here is an example of what might be in a `requirements.txt` file:

```
1 nose  
2 sqlalchemy
```

This example doesn't specify any versions, so the latest version of both the `nose` and `sqlalchemy` packages would be installed.

Here is an example of a `requirements.txt` with a version specified:

```
1 docopt == 0.6.1
```

When you install your dependencies using this file, it will install the 0.6.1 version of `docopt`.

Upgrading a Package

Packages from the Python Package Index can be updated quite rapidly. If you are using one of these packages and you want to upgrade to the latest, `pip` has a way to do that.

You can use `-U` or `--upgrade` to upgrade to the latest version of a package.

Here is an example:

```
1 python -m pip install --upgrade package_name
```

This will upgrade the specified package to its latest version.

Checking What's Installed

The `pip` library allows you to see what packages you have installed with it.

You can use `list` to show the installed packages, for example:

```
1 python3.8 -m pip list
```

When I ran this command, I got the following output:

```
1 Package      Version  
2 ----- -----  
3 pip          19.2.3  
4 setuptools   41.2.0
```

Your output will depend on what you have installed with pip.



Packages Not Installed with pip

If you installed Python packages using `apt-get` or `brew` or via an installer, then they will not show up when you run the `pip list` command because they were not installed with pip. You will need to use your package manager to see what you have installed.

Now let's learn how to uninstall a package!

Uninstalling Packages

Packages can also be uninstalled using pip.

You can use either of the following commands to make it work:

- 1 `pip uninstall [options] <package>`
- 2 `pip uninstall [options] -r <requirements file>`

The `uninstall` command can be used to uninstall multiple packages in the same way as the `install` command can be used to install multiple packages.

Once uninstalled, you can use the `list` command to verify that the package(s) have been uninstalled.

Alternatives to pip

There are several alternatives to using pip as your package manager. Here are a few of them:

- Conda - <https://docs.conda.io/en/latest/>
- Pipenv - <https://github.com/pypa/pipenv>
- pipx - <https://github.com/pipxproject/pipx>
- Poetry - <https://python-poetry.org/>

Conda is a utility provided with the Anaconda version of Python. If you are a data scientist or doing a lot of machine learning, you might find that using conda is the way to go. Anaconda is designed for data scientists and comes with many scientific Python packages pre-installed, such as **NumPy** and **Matplotlib**.

The **Pipenv** package is a wrapper around pip and a Python virtual environment. You'll learn more about virtual environments in the next chapter. Pipenv is also a dependency manager.

The **pipx** project is also used for installing Python packages in an isolated environment.

The Poetry package is a dependency management tool. It is kind of like conda in that respect and seems to be gaining a lot of support.

Conda, Pipenv and Poetry are more focused on dependency management and versioning of packages than pip itself is. They also promote using virtual environments for your code. If you will be working with multiples versions of packages or see yourself needing to test multiple packages often, then you may want to look into these packages.



Each of these tools works differently and are either not compatible with each other or are only partially compatible. Do not expect to be able to uninstall a conda package with pip or list conda packages with pip, for example.

Wrapping Up

You learned a lot about installing and managing packages with pip. To review, here is what you covered:

- Installing a Package
- Exploring Command Line Options
- Installing with requirements.txt
- Upgrading a Package
- Checking What's Installed
- Uninstalling Packages
- Alternatives to pip

At this point, you can install, list, upgrade and uninstall packages using pip. You also know about some alternatives that you could use instead of pip.

Review Questions

1. How do you install a package with pip?
2. What command do you use to see the version of the packages you installed?
3. How do you uninstall a package?

Chapter 21 - Python Virtual Environments

Python has the concept of the virtual environments built-in to the language. A Python virtual environment is an environment where you can install 3rd party packages for testing without affecting the system Python installation. Each virtual environment has its own set of installed packages and, depending on the virtual environment and how it's set up, may have its own copy of the Python binary and standard library as well.

There are several different ways to create Python virtual environments. You will focus on the following two methods:

- The built-in `venv` module
- The `virtualenv` package

There are other tools that you can use to create virtual Python environments. You will learn a little about them in the last section of this chapter.

For now, let's get started by looking at the `venv` library!

Python's `venv` Library

Python added the `venv` module in version 3.3. You can read all about it here:

- <https://docs.python.org/3/library/venv.html>

To use `venv`, you can run Python using the `-m` flag. The `-m` flag tells Python to run the specified module that follows `-m`.

Let's try it out. Open up a `cmd.exe` on Windows or a terminal in Mac or Linux. Then type the following:

```
1 python -m venv test
```

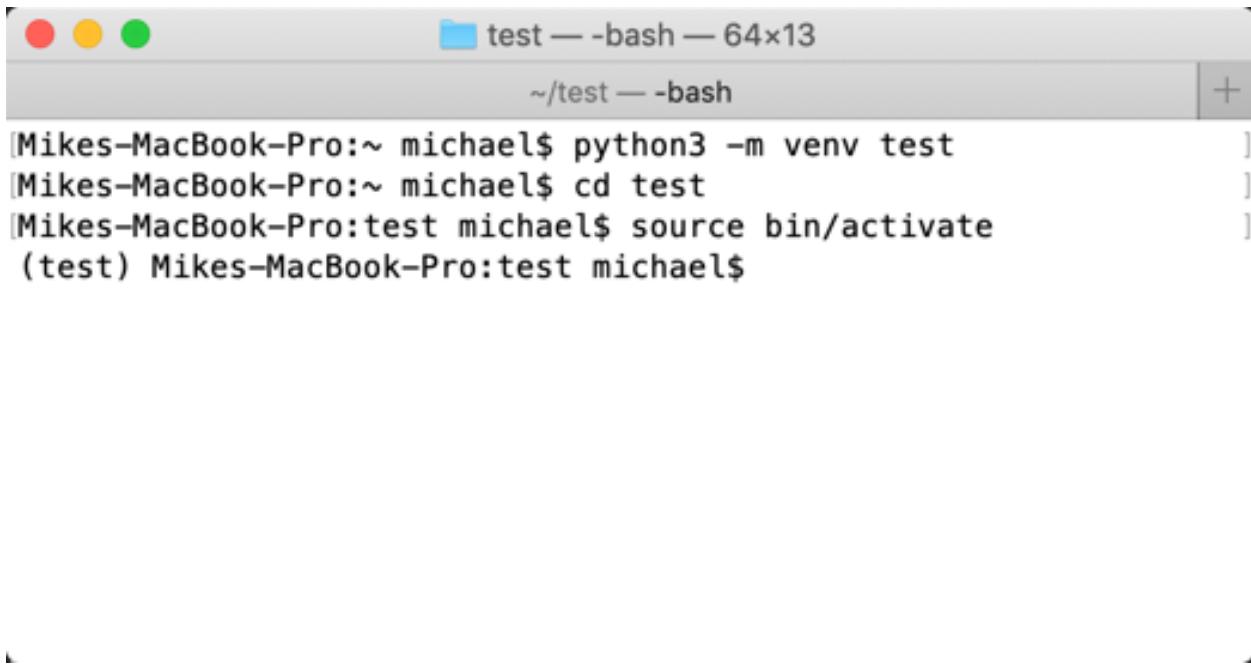
This will create a folder named `test` in whatever directory that you are open to in your terminal session.

To activate the virtual environment, you will need change directories into the `test` folder and run this on Linux/Mac:

```
1 source bin/activate
```

If you are a Windows user, you can activate it by running the bat file inside of the **Scripts** sub-folder that you will find in your **test** folder.

Now you should see something like this:



The screenshot shows a terminal window on a Mac OS X desktop. The window title is "test — -bash — 64x13". The window content shows the following command history:

```
[Mikes-MacBook-Pro:~ michael$ python3 -m venv test
[Mikes-MacBook-Pro:~ michael$ cd test
[Mikes-MacBook-Pro:test michael$ source bin/activate
(test) Mikes-MacBook-Pro:test michael$
```

Fig. 21-1: Activating a venv

Note that the name of the prompt is now “test”. That indicates that the virtual environment has been activated and is ready to use.

You can now install new packages and they will install to your virtual environment instead of your system Python.

When you are finished, you can deactivate the virtual environment by running **deactivate** in the terminal or command prompt. The exact nature of **deactivate** is implementation dependent: it may be a script or batch file or something else.

PyCharm, WingIDE and VS Code all support using Python virtual environments. In fact, you can usually create and activate them from within the IDE rather than doing it on the command line.

The `virtualenv` Package

The `virtualenv` package was the original method for creating Python virtual environments. You can read the documentation for the `virtualenv` package here:

- <https://virtualenv.pypa.io/en/latest/>

A subset of `virtualenv` was eventually integrated into Python's own `venv` module. The actual `virtualenv` package is better than `venv` in the following ways:

- It's faster
- Easier to extend
- Can create virtual environments for multiple Python versions
- Can be upgraded via `pip`
- Has a rich programmatic API

You can install `virtualenv` by using `pip`:

```
1 pip install virtualenv
```

Once installed, you can create a virtual environment using your terminal or `cmd.exe` like this:

```
1 virtualenv FOLDER_NAME
```

Activating and deactivating the virtual environment works exactly as it did when you created a virtual environment using Python's `venv` module.

There are quite a few command line parameters you can use with `virtualenv`. You can read the full listing here:

https://virtualenv.pypa.io/en/latest/cli_interface.html

Most of the time, you can use the defaults. But there are times when it is nice to configure your virtual environment to use other `pip` versions, or give it access to your system's site-packages folder. Check out the link above to see what all you can do with `virtualenv`.

Other Tools

There are other tools you can use to work with Python virtual environments. Here are just a few:

- Anaconda - <https://www.anaconda.com/>
- pipx - <https://pypi.org/project/pipx/>
- pipenv - <https://github.com/pypa/pipenv>

Anaconda has its own tooling for creating virtual environments.

The other two are popular packages for creating and managing virtual environments. Both `pipx` and `pipenv` are quite popular. You should read up on them and determine if they might be useful for your own projects.

Wrapping Up

Python virtual environments are a great way to isolate your system Python while allowing you to test out new packages. You can test out multiple versions of a package by using multiple virtual environments. Then when you are done, you can simply delete the virtual environment's folder.

This allows for quick iterations to verify that nothing in your package stack is causing breakage. Standard practice is to always use a virtual Python environment whenever you are testing out a new package.

Go ahead and give it a try. You'll soon find that it becomes second nature and it's super useful to boot!

Review Questions

1. How do you create a Python virtual environment?
2. What do you need to do after creating a virtual environment to use it?
3. Why would you use a Python virtual environment?

Chapter 22 - Type Checking in Python

Type checking or hinting is a newer feature of Python that was added in Python 3.5. Type hinting is also known as **type annotation**. You learned a little bit about type hinting back in chapter 17. Type hinting is adding special syntax to functions and variable declarations that tell the developer what type the argument or variable is.

Python does not enforce the type hints. You can still change types at will in Python because of this. However some integrated development environments, such as PyCharm, support type hinting and will highlight typing errors. You can also use a tool called **Mypy** to check your typing for you. You will learn more about that tool later on in this chapter.

You will be learning about the following:

- Pros and Cons of Type Hinting
- Built-in Type Hinting / Variable Annotation
- Collection Type Hinting
- Hinting Values That Could be None
- Type Hinting Functions
- What To Do When Things Get Complicated
- Classes
- Decorators
- Aliasing
- Other Type Hints
- Type Comments
- Static Type Checking

Let's get started!

Pros and Cons of Type Hinting

There are several things to know about up front when it comes to type hinting in Python. Let's look at the pros of type hinting first:

- Type hints are nice way to document your code in addition to docstrings
- Type hints can make IDEs and linters give better feedback and better autocomplete
- Adding type hints forces you to think about types, which may help you make good decisions during the design of your applications.

Adding type hinting isn't all rainbows and roses though. There are some downsides:

- The code is more verbose and arguably harder to write
- Type hinting adds development time
- Type hints only work in Python 3.5+. Before that, you had to use type comments
- Type hinting can have a minor start up time penalty in code that uses it, especially if you import the `typing` module.

When should you use type hinting then? Here are some examples:

- If you plan on writing short code snippets or one-off scripts, you don't need to include type hints.
- Beginners don't need to add type hints when learning Python
- If you are designing a library for other developers to use, adding type hints may be a good idea
- Large Python projects (i.e. thousands of lines of code) also can benefit from type hinting
- Some core developers recommend adding type hinting if you are going to write unit tests

Type hinting is a bit of a contentious topic in Python. You don't need to use it all the time, but there are certain cases where type hinting helps.

Let's spend the rest of this chapter learning how to use type hinting!

Built-in Type Hinting / Variable Annotation

You can add type hinting with the following built-in types:

- `int`
- `float`
- `bool`
- `str`
- `bytes`

These can be used both in functions and in variable annotation. The concept of variable annotation was added to the Python language in 3.6. Variable annotation allows you to add type hints to variables.

Here are some examples:

```
1 x: int # a variable named x without initialization
2 y: float = 1.0 # a float variable, initialized to 1.0
3 z: bool = False
4 a: str = 'Hello type hinting'
```

You can add a type hint to a variable without initializing it at all, as is the case in the first line of code. The other 3 lines of code show how to annotate each variable and initialize them appropriately.

Let's see how you would add type hinting for collections next!

Collection Type Hinting

A collection is a group of items in Python. Common collections or sequences are `list`, `dict`, `tuple` and `set`. However, you cannot annotate variables using these built-in types. Instead, you must use the `typing` module.

Let's look at a few examples:

```
1 >>> from typing import List
2 >>> names: List[str] = ['Mike']
3 >>> names
4 ['Mike']
```

Here you created a `list` with a single `str` in it. This specifies that you are creating a `list` of strings. If you know the list is always going to be the same size, you can specify each item's type in the list:

```
1 >>> from typing import List
2 >>> names: List[str, str] = ['Mike', 'James']
```

Hinting tuples is very similar:

```
1 >>> from typing import Tuple
2 >>> s: Tuple[int, float, str] = (5, 3.14, 'hello')
```

Dictionaries are a little different in that you should hint types the key and values are:

```
1 >>> from typing import Dict
2 >>> d: Dict[str, int] = {'one': 1}
```

If you know a collection will have variable size, you can use an ellipses:

```
1 >>> from typing import Tuple
2 >>> t: Tuple[int, ...] = (4, 5, 6)
```

Now let's learn what to do if an item is of type `None`!

Hinting Values That Could be `None`

Sometimes a value needs to be initialized as `None`, but when it gets set later, you want it to be something else.

For that, you can use `Optional`:

```
1 >>> from typing import Optional
2 >>> result: Optional[str] = my_function()
```

On the other hand, if the value can never be `None`, you should add an `assert` to your code:

```
1 >>> assert result is not None
```

Let's find out how to annotate functions next!

Type Hinting Functions

Type hinting functions is similar to type hinting variables. The main difference is that you can also add a return type to a function.

Let's take a look at an example:

```
1 def adder(x: int, y: int) -> None:
2     print(f'The total of {x} + {y} = {x+y}')
```

This example shows you that `adder()` takes two arguments, `x` and `y`, and that they should both be integers. The return type is `None`, which you specify using the `->` after the ending parentheses but before the colon.

Let's say that you want to assign the `adder()` function to a variable. You can annotate the variable as a `Callable` like this:

```
1 from typing import Callable
2
3 def adder(x: int, y: int) -> None:
4     print(f'The total of {x} + {y} = {x+y}')
5
6 a: Callable[[int, int], None] = adder
```

The `Callable` takes in a list of arguments for the function. It also allows you to specify the return type.

Let's look at one more example where you pass in more complex arguments:

```
1 from typing import Tuple, Optional
2
3
4 def some_func(x: int,
5               y: Tuple[str, str],
6               z: Optional[float] = None) -> Optional[str]:
7     if x > 10:
8         return None
9     return 'You called some_func'
```

For this example, you created `some_func()` that accepts 3 arguments:

- an `int`
- a two-item `tuple` of strings
- an optional `float` that is defaulted to `None`

Note that when you use defaults in a function, you should add a space before and after the equals sign when using type hints.

It also returns either `None` or a string.

Let's move on and discover what to do in even more complex situations!

What To Do When Things Get Complicated

You have already learned what to do when a value can be `None`, but what else can you do when things get complicated? For example, what do you do if the argument being passed in can be multiple different types?

For that specific use case, you can use `Union`:

```
1 >>> from typing import Union
2 >>> z: Union[str, int]
```

What this type hint means is that the variable, `z`, can be either a string or an integer.

There are also cases where a function may take in an object. If that object can be one of several different objects, then you can use `Any`.

```
1 x: Any = some_function()
```

Use `Any` with caution because you can't really tell what it is that you are returning. Since it can be "any" type, it is like catching all exceptions with a bare `except`. You don't know what exception you are catching with that and you also don't know what type you are hinting at when you use `Any`.

Classes

If you have a `class` that you have written, you can create an annotation for it as well.

```
1 >>> class Test:
2     ...
3     pass
4 >>> t: Test = Test()
```

This can be really useful if you are passing around instances of your class between functions or methods.

Decorators

Decorators are a special beast. They are functions that take other functions and modify them. You will learn about decorators later on in this book.

Adding type hints to decorators is kind of ugly.

Let's take a look:

```

1 from typing import Any, Callable, TypeVar, cast
2 F = TypeVar('F', bound=Callable[..., Any])
3
4
5 def my_decorator(func: F) -> F:
6     def wrapper(*args, **kwds):
7         print("Calling", func)
8         return func(*args, **kwds)
9     return cast(F, wrapper)

```

A TypeVar is a way to specify a custom type. You are creating a custom Callable type that can take in any number of arguments and returns Any. Then you create a decorator and add the new type as a type hint for the first argument as well as the return type.

The cast function is used by Mypy, the static code checker utility. It is used to cast a value to the specified type. In this case, you are casting the wrapper function as a type F.

Aliasing

You can create a new name for a type. For example, let's rename the List type to Vector:

```

1 >>> from typing import List
2 >>> Vector = List[int]
3 >>> def some_function(a: Vector) -> None:
4     ...     print(a)

```

Now Vector and List[int] refer to the same type hint. Aliasing a type hint is useful for complex types.

The typing documentation has a good example that is reproduced below:

```

1 from typing import Dict, Tuple
2
3 ConnectionOptions = Dict[str, str]
4 Address = Tuple[str, int]
5 Server = Tuple[Address, ConnectionOptions]

```

This code allows you to nest types inside of other types while still being able to write appropriate type hints.

Other Type Hints

There are several other type hints that you can use as well. For example, there are generic mutable types such as `MutableMapping` that you might use for a custom mutable dictionary.

There is also a `ContextManager` type that you would use for context managers.

Check out the full documentation for all the details of all the various types:

- <https://docs.python.org/3/library/typing.html>

Type Comments

Python 2.7 development ended January 1, 2020. However, there will be many lines of legacy Python 2 code that people will have to work with for years to come. Type hinting was never added to Python 2. But you can use a similar syntax as comments.

Here is an example:

```
1 def some_function(a):
2     # type: str -> None
3     print(a)
```

To make this work, you need to have the comment start with `type:`. This line must be on the same or following line of the code that it is hinting. If the function takes multiple arguments, then you would separate the hints with commas:

```
1 def some_function(a, b, c):
2     # type: (str, int, int) -> None
3     print(a)
```

Some Python IDEs may support type hinting in the docstring instead. PyCharm lets you do the following for example:

```
1 def some_function(a, b):
2     """
3     @type a: int
4     @type b: float
5     """
```

Mypy will work on the other comments, but not on these. If you are using PyCharm, you can use either form of type hinting.

If your company wants to use type hinting, you should advocate to upgrade to Python 3 to get the most out of it.

Static Type Checking

You have seen Mypy mentioned several times already. You can read all about it here:

- <http://mypy-lang.org/>

If you would like to run Mypy on your own code, you will need to install it using pip:

```
1 $ pip install mypy
```

Once you have mypy installed, you can run the tool like this:

```
1 $ mypy my_program.py
```

Mypy will analyze your code and print out any type errors that it finds. Mypy does this by reading and parsing your source files without actually running your code. This process of analyzing without running is known as **static checking**. Many linters also use static checking, but they are usually looking for syntax errors, unused or missing variables, and other problems that would prevent your code from running.

If there is no type hinting in your program, Mypy will not report any errors at all.

Let's write a badly type hinted function and save it to a file named `bad_type_hinting.py`:

```
1 # bad_type_hinting.py
2
3 def my_function(a: str, b: str) -> None:
4     return a.keys() + b.keys()
```

Now that you have some code, you can run Mypy against it:

```
1 $ mypy bad_type_hinting.py
2 bad_type_hinting.py:4: error: "str" has no attribute "keys"
3 Found 1 error in 1 file (checked 1 source file)
```

This output tells you that there is an issue on line 4. Strings do not have a `keys()` attribute.

Let's update the code to remove the calls to the nonexistent `keys()` method. You can save these changes to a new file named `bad_type_hinting2.py`:

```
1 # bad_type_hinting2.py
2
3 def my_function(a: str, b: str) -> None:
4     return a + b
```

Now you should run Mypy against your change and see if you fixed it:

```
1 $ mypy bad_type_hinting2.py
2 bad_type_hinting2.py:4: error: No return value expected
3 Found 1 error in 1 file (checked 1 source file)
```

Whoops! There's still an error. This time you know that you weren't expecting this function to return anything. You could fix the code so that it doesn't return anything or you could fix the type hint so that it returns a `str`.

You should try doing the latter and save the following code to `good_type_hinting.py`:

```
1 # good_type_hinting.py
2
3 def my_function(a: str, b: str) -> str:
4     return a + b
```

Now run Mypy against this new file:

```
1 $ mypy good_type_hinting.py
2 Success: no issues found in 1 source file
```

This time your code has no issues!

You can run Mypy against multiple files or even an entire folder. If you are dedicated to using type hinting in your code, then you should be running Mypy on your code frequently to make sure your code is error free.

Wrapping Up

You now know what type hinting or annotation is and how to do it. In fact, you have learned all the basics that you need to do type hinting effectively.

In this chapter, you learned about:

- Pros and Cons of Type Hinting
- Built-in Type Hinting / Variable Annotation

- Collection Type Hinting
- Hinting Values That Could be `None`
- Type Hinting Functions
- What To Do When Things Get Complicated
- Classes
- Decorators
- Aliasing
- Other Type Hints
- Type Comments
- Static Type Checking

If you get stuck, you should check out the following resources for help:

- https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html
- Typeshed - <https://github.com/python/typeshed>
- <https://docs.python.org/3/library/typing.html>

Type hinting is not necessary in Python. You can write all your code without ever adding any annotations to your code. But type hinting is good to understand and may prove handy to have in your toolbox.

Review Questions

1. What is type hinting in your own words?
2. Why would you use type hinting?
3. Demonstrate your understanding of type hinting by adding type annotations to the variables as well as the function. Don't forget the return type!

```
1 a = 1
2 b = 3.14
3
4 def my_function(x=[], y=None):
5     if y is not None:
6         result = [i * y for i in x]
7     else:
8         result = x
9     return result
```

Chapter 23 - Creating Multiple Threads

Concurrency is a big topic in programming. The concept of concurrency is to run multiple pieces of code at once. Python has a couple of different solutions that are built-in to its standard library. You can use threads or processes. In this chapter, you will learn about using threads.

When you run your own code, you are using a single thread. If you want to run something else in the background, you can use Python's `threading` module.

In this chapter you will learn the following:

- Pros of Using Threads
- Cons of Using Threads
- Creating Threads
- Subclassing Thread
- Writing Multiple Files with Threads

Note: *This chapter is not meant to be comprehensive in its coverage of threads. But you will learn enough to get started using threads in your application.*

Let's get started by going over the pros and cons of using threads!

Pros of Using Threads

Threads are useful in the following ways:

- They have a small memory footprint, which means they are lightweight to use
- Memory is shared between threads - which makes it easy to share state across threads
- Allows you to easily make responsive user interfaces
- Great option for I/O bound applications (such as reading and writing files, databases, etc)

Now let's look at the cons!

Cons of Using Threads

Threads are **not** useful in the following ways:

- Poor option for CPU bound code due to the **Global Interpreter Lock (GIL)** - see below
- They are not interruptible / able to be killed
- Code with threads is harder to understand and write correctly
- Easy to create race conditions

The Global Interpreter Lock is a mutex that protects Python objects. This means that it prevents multiple threads from executing Python bytecode at the same time. So when you use threads, they do **not** run on all the CPUs on your machine.

Threads are great for running I/O heavy applications, image processing, and NumPy's number crunching because they don't do anything with the GIL. If you have a need to run concurrent processes across multiple CPUs, use the `multiprocessing` module. You will learn about the `multiprocessing` module in the next chapter.

A **race condition** happens when you have a computer program that depends on a certain order of events to happen for it to execute correctly. If your threads execute something out of order, then the next thread may not work and your application can crash or behave in unexpected ways.

Creating Threads

Threads are confusing if all you do is talk about them. It's always good to familiarize yourself with how to write actual code. For this chapter, you will be using the `threading` module which uses the `_thread` module underneath.

The full documentation for the `threading` module can be found here:

- <https://docs.python.org/3/library/threading.html>

Let's write a simple example that shows how to create multiple threads. Put the following code into a file named `worker_threads.py`:

```
1 # worker_threads.py
2
3 import random
4 import threading
5 import time
6
7
8 def worker(name: str) -> None:
9     print(f'Started worker {name}')
10    worker_time = random.choice(range(1, 5))
11    time.sleep(worker_time)
12    print(f'{name} worker finished in {worker_time} seconds')
13
14 if __name__ == '__main__':
15     for i in range(5):
16         thread = threading.Thread(
17             target=worker,
18             args=(f'computer_{i}',),
19             )
20         thread.start()
```

The first three imports give you access to the `random`, `threading` and `time` modules. You can use `random` to generate pseudo-random numbers or choose from a sequence at random. The `threading` module is what you use to create threads and the `time` module can be used for many things related to time.

In this code, you use `time` to wait a random amount of time to simulate your “worker” code working.

Next you create a `worker()` function that takes in the `name` of the worker. When this function is called, it will print out which worker has started working. It will then choose a random number between 1 and 5. You use this number to simulate the amount of time the worker works using `time.sleep()`. Finally you print out a message that tells you a worker has finished and how long the work took in seconds.

The last block of code creates 5 worker threads. To create a thread, you pass in your `worker()` function as the `target` function for the thread to call. The other argument you pass to `thread` is a tuple of arguments that `thread` will pass to the target function. Then you call `thread.start()` to start running that thread.

When the function stops executing, Python will delete your thread.

Try running the code and you’ll see that the output will look similar to the following:

```
1 Started worker computer_0
2 Started worker computer_1
3 Started worker computer_2
4 Started worker computer_3
5 Started worker computer_4
6 computer_0 worker finished in 1 seconds
7 computer_3 worker finished in 1 seconds
8 computer_4 worker finished in 3 seconds
9 computer_2 worker finished in 3 seconds
10 computer_1 worker finished in 4 seconds
```

Your output will differ from the above because the workers `sleep()` for random amounts of time. In fact, if you run the code multiple times, each invocation of the script will probably have a different result.

`threading.Thread` is a class. Here is its full definition:

```
1 threading.Thread(
2     group=None, target=None, name=None,
3     args=(), kwargs={},
4     *,
5     daemon=None,
6 )
```

You could have named the threads when you created the thread rather than inside of the `worker()` function. The `args` and `kwargs` are for the target function. You can also tell Python to make the thread into a daemon. “Daemon threads” have no claim on the Python interpreter, which has two main consequences: 1) if only daemon threads are left, Python will shut down, and 2) when Python shuts down, daemon threads are abruptly stopped with no notification. The `group` parameter should be left alone as it was added for future extension when a `ThreadGroup` is added to the Python language.

Subclassing Thread

The `Thread` class from the `threading` module can also be subclassed. This allows you more fine-grained control over your thread’s creation, execution and eventual deletion. You will encounter subclassed threads often.

Let’s rewrite the previous example using a subclass of `Thread`. Put the following code into a file named `worker_thread_subclass.py`.

```
1 # worker_thread_subclass.py
2
3 import random
4 import threading
5 import time
6
7 class WorkerThread(threading.Thread):
8
9     def __init__(self, name):
10         threading.Thread.__init__(self)
11         self.name = name
12         self.id = id(self)
13
14     def run(self):
15         """
16             Run the thread
17         """
18         worker(self.name, self.id)
19
20     def worker(name: str, instance_id: int) -> None:
21         print(f'Started worker {name} - {instance_id}')
22         worker_time = random.choice(range(1, 5))
23         time.sleep(worker_time)
24         print(f'{name} - {instance_id} worker finished in '
25               f'{worker_time} seconds')
26
27 if __name__ == '__main__':
28     for i in range(5):
29         thread = WorkerThread(name=f'computer_{i}')
30         thread.start()
```

In this example, you create the `WorkerThread` class. The constructor of the class, `__init__()`, accepts a single argument, the `name` to be given to thread. This is stored off in an instance attribute, `self.name`. Then you override the `run()` method.

The `run()` method is already defined in the `Thread` class. It controls how the thread will run. It will call or invoke the function that you passed into the class when you created it. When you create your own `run()` method in your subclass, it is known as **overriding** the original. This allows you to add custom behavior such as logging to your thread that isn't there if you were to use the base class's `run()` method.

You call the `worker()` function in the `run()` method of your `WorkerThread`. The `worker()` function itself has a minor change in that it now accepts the `instance_id` argument which represents the class instance's unique id. You also need to update the `print()` functions so that they print out the

```
instance_id.
```

The other change you need to do is in the `__main__` conditional statement where you call `WorkerThread` and pass in the name rather than calling `threading.Thread()` directly as you did in the previous section.

When you call `start()` in the last line of the code snippet, it will call `run()` for you itself. The `start()` method is a method that is a part of the `threading.Thread` class and you did not override it in your code.

The output when you run this code should be similar to the original version of the code, except that now you are also including the instance id in the output. Give it a try and see for yourself!

Writing Multiple Files with Threads

There are several common use cases for using threads. One of those use cases is writing multiple files at once. It's always nice to see how you would approach a real-world problem, so that's what you will be doing here.

To get started, you can create a file named `writing_thread.py`. Then add the following code to your file:

```
1 # writing_thread.py
2
3 import random
4 import time
5 from threading import Thread
6
7
8 class WritingThread(Thread):
9
10     def __init__(self,
11                  filename: str,
12                  number_of_lines: int,
13                  work_time: int = 1) -> None:
14         Thread.__init__(self)
15         self.filename = filename
16         self.number_of_lines = number_of_lines
17         self.work_time = work_time
18
19     def run(self) -> None:
20         """
21             Run the thread
22         """
```

```

23     print(f'Writing {self.number_of_lines} lines of text to '
24         f'{self.filename}')
25     with open(self.filename, 'w') as f:
26         for line in range(self.number_of_lines):
27             text = f'This is line {line+1}\n'
28             f.write(text)
29             time.sleep(self.work_time)
30     print(f'Finished writing {self.filename}')
31
32 if __name__ == '__main__':
33     files = [f'test{x}.txt' for x in range(1, 6)]
34     for filename in files:
35         work_time = random.choice(range(1, 3))
36         number_of_lines = random.choice(range(5, 20))
37         thread = WritingThread(filename, number_of_lines, work_time)
38         thread.start()

```

Let's break this down a little and go over each part of the code individually:

```

1 import random
2 import time
3 from threading import Thread
4
5
6 class WritingThread(Thread):
7
8     def __init__(self,
9                  filename: str,
10                 number_of_lines: int,
11                 work_time: int = 1) -> None:
12         Thread.__init__(self)
13         self.filename = filename
14         self.number_of_lines = number_of_lines
15         self.work_time = work_time

```

Here you created the `WritingThread` class. It accepts a `filename`, a `number_of_lines` and a `work_time`. This allows you to create a text file with a specific number of lines. The `work_time` is for sleeping between writing each line to simulate writing a large or small file.

Let's look at what goes in `run()`:

```

1  def run(self) -> None:
2      """
3          Run the thread
4      """
5      print(f'Writing {self.number_of_lines} lines of text to '
6            f'{self.filename}')
7      with open(self.filename, 'w') as f:
8          for line in range(self.number_of_lines):
9              text = f'This is line {line+1}\n'
10             f.write(text)
11             time.sleep(self.work_time)
12     print(f'Finished writing {self.filename}')

```

This code is where all the magic happens. You print out how many lines of text you will be writing to a file. Then you do the deed and create the file and add the text. During the process, you `sleep()` to add some artificial time to writing the files to disk.

The last piece of code to look at is as follows:

```

1 if __name__ == '__main__':
2     files = [f'test{x}.txt' for x in range(1, 6)]
3     for filename in files:
4         work_time = random.choice(range(1, 3))
5         number_of_lines = random.choice(range(5, 20))
6         thread = WritingThread(filename, number_of_lines, work_time)
7         thread.start()

```

In this final code snippet, you use a list comprehension to create 5 file names. Then you loop over the files and create them. You use Python's `random` module to choose a random `work_time` amount and a random `number_of_lines` to write to the file. Finally you create the `WritingThread` and `start()` it.

When you run this code, you will see something like this get output:

```

1 Writing 5 lines of text to test1.txt
2 Writing 18 lines of text to test2.txt
3 Writing 7 lines of text to test3.txt
4 Writing 11 lines of text to test4.txt
5 Writing 11 lines of text to test5.txt
6 Finished writing test1.txt
7 Finished writing test3.txt
8 Finished writing test4.txt
9 Finished writing test5.txt
10 Finished writing test2.txt

```

You may notice some odd output like the line a couple of lines from the bottom. This happened because multiple threads happened to write to stdout at once.

You can use this code along with Python's `urllib.request` to create an application for downloading files from the Internet. Try that project out on your own.

Wrapping Up

You have learned the basics of threading in Python. In this chapter, you learned about the following:

- Pros of Using Threads
- Cons of Using Threads
- Creating Threads
- Subclassing Thread
- Writing Multiple Files with Threads

There is a lot more to threads and concurrency than what is covered here. You didn't learn about thread communication, thread pools, or locks for example. However you do know the basics of creating threads and you will be able to use them successfully. In the next chapter, you will continue to learn about concurrency in Python through discovering how `multiprocessing` works in Python!

Review Questions

1. What are threads good for?
2. Which module do you use to create a thread in Python?
3. What is the Global Interpreter Lock?

Chapter 24 - Creating Multiple Processes

Most CPU manufacturers are creating multi-core CPUs now. Even cell phones come with multiple cores! Python threads can't use those cores because of the Global Interpreter Lock. Starting in Python 2.6, the `multiprocessing` module was added which lets you take full advantage of all the cores on your machine.

In this chapter, you will learn about the following topics:

- Pros of Using Processes
- Cons of Using Processes
- Creating Processes with `multiprocessing`
- Subclassing Process
- Creating a Process Pool

This chapter is not a comprehensive overview of multiprocessing. The topic of multiprocessing and concurrency in general would be better suited in a book of its own. You can always check out the documentation for the `multiprocessing` module if you need to here:

- <https://docs.python.org/2/library/multiprocessing.html>

Now, let's get started!

Pros of Using Processes

There are several pros to using processes:

- Processes use separate memory space
- Code can be more straightforward compared to threads
- Uses multiple CPUs / cores
- Avoids the Global Interpreter Lock (GIL)
- Child processes can be killed (unlike threads)
- The `multiprocessing` module has an interface similar to `threading.Thread`
- Good for CPU-bound processing (encryption, binary search, matrix multiplication)

Now let's look at some of the cons of processes!

Cons of Using Processes

There are also a couple of cons to using processes:

- Interprocess communication is more complicated
- Memory footprint is larger than threads

Now let's learn how to create a process with Python!

Creating Processes with `multiprocessing`

The `multiprocessing` module was designed to mimic how the `threading.Thread` class worked. With that in mind, you can take the code from the previous chapter and modify it to use processes instead of threads.

Here's how you would modify the first example:

```
1 import multiprocessing
2 import random
3 import time
4
5
6 def worker(name: str) -> None:
7     print(f'Started worker {name}')
8     worker_time = random.choice(range(1, 5))
9     time.sleep(worker_time)
10    print(f'{name} worker finished in {worker_time} seconds')
11
12 if __name__ == '__main__':
13     processes = []
14     for i in range(5):
15         process = multiprocessing.Process(
16             target=worker,
17             args=(f'computer_{i},))
18         processes.append(process)
19         process.start()
20
21     for proc in processes:
22         proc.join()
```

The first change here is that you are importing the `multiprocessing` module. The other two imports are for the `random` and `time` modules respectively.

Then you have the silly `worker()` function that pretends to do some work. It takes in a `name` and returns nothing. Inside the `worker()` function, it will print out the name of the worker, then it will use `time.sleep()` to simulate doing some long-running process. Finally, it will print out that it has finished.

The last part of the code snippet is where you create 5 worker processes. You use `multiprocessing.Process()`, which works pretty much the same way as `threading.Thread()` did. You tell `Process` what target function to use and what arguments to pass to it. The main difference is that this time you are creating a list of processes. For each process, you call its `start()` method to start the process.

Then at the end, you loop over the list of processes and call its `join()` method, which tells Python to wait for the process to terminate.

When you run this code, you will see output that is similar to the following:

```
1 Started worker computer_0
2 Started worker computer_1
3 Started worker computer_2
4 Started worker computer_3
5 Started worker computer_4
6 computer_2 worker finished in 2 seconds
7 computer_1 worker finished in 3 seconds
8 computer_3 worker finished in 3 seconds
9 computer_0 worker finished in 4 seconds
10 computer_4 worker finished in 4 seconds
```

Each time you run your script, the output will be a little different because of the `random` module. Give it a try and see for yourself!

Subclassing Process

The `Process` class from the `multiprocessing` module can also be subclassed. It works in much the same way as the `threading.Thread` class does.

Let's take a look:

```
1 # worker_thread_subclass.py
2
3 import random
4 import multiprocessing
5 import time
6
7 class WorkerProcess(multiprocessing.Process):
8
9     def __init__(self, name):
10         multiprocessing.Process.__init__(self)
11         self.name = name
12
13     def run(self):
14         """
15             Run the thread
16         """
17         worker(self.name)
18
19     def worker(name: str) -> None:
20         print(f'Started worker {name}')
21         worker_time = random.choice(range(1, 5))
22         time.sleep(worker_time)
23         print(f'{name} worker finished in {worker_time} seconds')
24
25 if __name__ == '__main__':
26     processes = []
27     for i in range(5):
28         process = WorkerProcess(name=f'computer_{i}')
29         processes.append(process)
30         process.start()
31
32     for process in processes:
33         process.join()
```

This code should look familiar. The `WorkerProcess` class is exactly the same as the `WorkerThread` class from the previous chapter except that it is subclassing `Process` instead of `Thread`.

The only difference in this code is on line 28 where you instantiate the class. Here you must create a process and add it to a process list. Then to get it to work properly, you need to loop over the list of processes and call `join()` on each of them. This works exactly as it did in the previous process example from the last section.

The output from this class should also be quite similar to the output from the previous section.

Creating a Process Pool

If you have a lot of processes to run, sometime you will want to limit the number of processes that can run at once. For example, let's say you need to run 20 processes but you have a processor with only 4 cores. You can use the `multiprocessing` module to create a process pool that will limit the number of processes running to only 4 at a time.

Here's how you can do it:

```
1 import random
2 import time
3
4 from multiprocessing import Pool
5
6
7 def worker(name: str) -> None:
8     print(f'Started worker {name}')
9     worker_time = random.choice(range(1, 5))
10    time.sleep(worker_time)
11    print(f'{name} worker finished in {worker_time} seconds')
12
13 if __name__ == '__main__':
14     process_names = [f'computer_{i}' for i in range(15)]
15     pool = Pool(processes=5)
16     pool.map(worker, process_names)
17     pool.terminate()
```

In this example, you have the same `worker()` function. The real meat of the code is at the end where you create 15 process names using a list comprehension. Then you create a `Pool` and set the total number of processes to run at once to 5. To use the `pool`, you need to call the `map()` method and pass it the function you wish to call along with the arguments to pass to the function.

Python will now run 5 processes (or less) at a time until all the processes have finished. You need to call `terminate()` on the pool at the end or you will see a message like this:

```
1 /Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/multiprocessing/reso\
2 urce_tracker.py:216:
3 UserWarning: resource_tracker: There appear to be 6 leaked semaphore objects to clea\
4 n up at shutdown
```

Now you know how to create a process Pool with Python!

Wrapping Up

You have now learned the basics of using the `multiprocessing` module. You have learned the following:

- Pros of Using Processes
- Cons of Using Processes
- Creating Processes with `multiprocessing`
- Subclassing Process
- Creating a Process Pool

There is much more to `multiprocessing` than what is covered here. You could learn how to use Python's `Queue` module to get output from processes. There is the topic of interprocess communication. And there's much more too. However the objective was to learn how to create processes, not learn every nuance of the `multiprocessing` module. Concurrency is a large topic that would need much more in-depth coverage than what can be covered in this book.

Review Questions

1. What are processes good for?
2. How do you create a process in Python?
3. Can you create a process pool in Python? How?
4. What effect, if any, does the Global Interpreter Lock have on processes?
5. What happens if you don't use `process.join()`?

Chapter 25 - Launching Subprocesses with Python

There are times when you are writing an application and you need to run another application. For example, you may need to open Microsoft Notepad on Windows for some reason. Or if you are on Linux, you might want to run `grep`. Python has support for launching external applications via the `subprocess` module.

The `subprocess` module has been a part of Python since Python 2.4. Before that you needed to use the `os` module. You will find that the `subprocess` module is quite capable and straightforward to use.

In this chapter you will learn how to use:

- The `subprocess.run()` Function
- The `subprocess.Popen()` Class
- The `subprocess.Popen.communicate()` Function
- Reading and Writing with `stdin` and `stdout`

Let's get started!

The `subprocess.run()` Function

The `run()` function was added in **Python 3.5**. The `run()` function is the recommended method of using `subprocess`.

It can often be generally helpful to look at the definition of a function, to better understand how it works:

```
1 subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None,
2                 capture_output=False, shell=False, cwd=None, timeout=None, check=False,
3                 encoding=None, errors=None, text=None, env=None, universal_newlines=None)
```

You do not need to know what all of these arguments do to use `run()` effectively. In fact, most of the time you can probably get away with only knowing what goes in as the first argument and whether or not to enable `shell`. The rest of the arguments are helpful for very specific use-cases.

Let's try running a common Linux / Mac command, `ls`. The `ls` command is used to list the files in a directory. By default, it will list the files in the directory you are currently in.

To run it with `subprocess`, you would do the following:

```
1 >>> import subprocess
2 >>> subprocess.run(['ls'])
3 filename
4 CompletedProcess(args=['ls'], returncode=0)
```

You can also set `shell=True`, which will run the command through the shell itself. Most of the time, you will not need to do this, but it can be useful if you need more control over the process and want to access shell pipes and wildcards.

But what if you want to keep the output from a command so you can use it later on? Let's find out how you would do that next!

Getting the Output

Quite often you will want to get the output from an external process and then do something with that data. To get output from `run()` you can set the `capture_output` argument to `True`:

```
1 >>> subprocess.run(['ls', '-l'], capture_output=True)
2 CompletedProcess(args=['ls', '-l'], returncode=0,
3                   stdout=b'total 40\n-rw-r--r--@ 1 michael  staff  17083 Apr 15 13:17 some_file\n',
4                   stderr=b'')
```

Now this isn't too helpful as you didn't save the returned output to a variable. Go ahead and update the code so that you do and then you'll be able to access `stdout`.

```
1 >>> output = subprocess.run(['ls', '-l'], capture_output=True)
2 >>> output.stdout
3 b'total 40\n-rw-r--r--@ 1 michael  staff  17083 Apr 15 13:17 some_file\n'
```

The `output` is a `CompletedProcess` class instance, which lets you access the `args` that you passed in, the `returncode` of the subprocess, as well as the subprocess' `stdout` and `stderr`.

You will learn about the `returncode` in a moment. The `stderr` is where most programs print their error messages to, while `stdout` is for ordinary program output.

If you are interested, you can play around with this code and discover what is currently in those attributes, if anything:

```
1 output = subprocess.run(['ls', '-l'], capture_output=True)
2 print(output.returncode)
3 print(output.stdout)
4 print(output.stderr)
```

Let's move on and learn about `Popen` next.

The `subprocess.Popen()` Class

The `subprocess.Popen()` class has been around since the `subprocess` module itself was added. It has been updated several times in Python 3. If you are interested in learning about some of those changes, you can read about them here:

- <https://docs.python.org/3/library/subprocess.html#popen-constructor>

You can think of `Popen` as the low-level version of `run()`. If you have an unusual use-case that `run()` cannot handle, then you should be using `Popen` instead.

For now, let's look at how you would run the command in the previous section with `Popen`:

```
1 >>> import subprocess
2 >>> subprocess.Popen(['ls', '-l'])
3 <subprocess.Popen object at 0x10f88bdf0>
4 >>> total 40
5 -rw-r--r--@ 1 michael  staff  17083 Apr 15 13:17 some_file
6
7 >>>
```

The syntax is almost identical except that you are using `Popen` instead of `run()`.

Here is how you might get the return code from the external process:

```
1 >>> process = subprocess.Popen(['ls', '-l'])
2 >>> total 40
3 -rw-r--r--@ 1 michael  staff  17083 Apr 15 13:17 some_file
4
5 >>> return_code = process.wait()
6 >>> return_code
7 0
8 >>>
```

A `return_code` of `0` means that the program finished successfully. If you open up a program with a user interface, such as Microsoft Notepad, you will need to switch back to your REPL or IDLE session to add the `process.wait()` line. The reason for this is that Notepad will appear over the top of your program.

If you do not add the `process.wait()` call to your script, then you won't be able to catch the return code after manually closing any user interface program you may have started up via `subprocess`.

You can use your `process` handle to access the process id via the `pid` attribute. You can also kill (SIGKILL) the process by calling `process.kill()` or terminate (SIGTERM) it via `process.terminate()`.



What is SIGKILL / SIGTERM?

If you are not familiar with the terms “SIGKILL” and “SIGTERM”, just note that these are standard commands related to interacting with processes in UNIX systems. The subprocess.Popen() class provides a convenient interface to interact with these commands. If you’re interested, you can read more on Wikipedia about Signal Handling:

- [https://en.wikipedia.org/wiki/Signal_\(IPC\)](https://en.wikipedia.org/wiki/Signal_(IPC))

The subprocess.Popen.communicate() Function

There are times when you need to communicate with the process that you have spawned. You can use the Popen.communicate() method to send data to the process as well as extract data.

For this section, you will only use communicate() to extract data. Let’s use communicate() to get information using the ifconfig command, which you can use to get information about your computer’s network card on Linux or Mac. On Windows, you would use ipconfig. Note that there is a one-letter difference in this command, depending on your Operating System.

Here’s the code:

```
1  >>> import subprocess
2  >>> cmd = ['ifconfig']
3  >>> process = subprocess.Popen(
4      cmd,
5      stdout=subprocess.PIPE,
6      encoding='utf-8',
7      )
8  >>> data = process.communicate()
9  >>> print(data[0])
10 >>> 
10: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
11     options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
12     inet 127.0.0.1 netmask 0xff000000
13     inet6 ::1 prefixlen 128
14     inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
15     nd6 options=201<PERFORMNUD,DAD>
16 gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
17 stf0: flags=0<> mtu 1280
18 XHC20: flags=0<> mtu 0
19 # ----- truncated -----
```

This code is set up a little differently than the last one. Let's go over each piece in more detail.

The first thing to note is that you set the `stdout` parameter to a `subprocess.PIPE`. That allows you to capture anything that the process sends to `stdout`. You also set the encoding to `utf-8`. The reason you do that is to make the output a little easier to read, since the `subprocess.Popen` call returns bytes by default rather than strings.

The next step is to call `communicate()` which will capture the data from the process and return it. The `communicate()` method returns both `stdout` and `stderr`, so you will get a tuple. You didn't capture `stderr` here, so that will be `None`.

Finally, you print out the data. The string is fairly long, so the output is truncated here.

Let's move on and learn how you might read and write with `subprocess`!

Reading and Writing with `stdin` and `stdout`

Let's pretend that your task for today is to write a Python program that checks the currently running processes on your Linux server and prints out the ones that are running with Python.

You can get a list of currently running processes using `ps -ef`. Normally you would use that command and "pipe" it to `grep`, another Linux command-line utility used for searching for strings in files.

Here is the complete Linux command you could use:

```
1 ps -ef | grep python
```

However, you want to translate that command into Python using the `subprocess` module.

Here is one way you can do that:

```
1 import subprocess
2
3 cmd = ['ps', '-ef']
4 ps = subprocess.Popen(cmd, stdout=subprocess.PIPE)
5
6 cmd = ['grep', 'python']
7 grep = subprocess.Popen(
8         cmd,
9         stdin=ps.stdout,
10        stdout=subprocess.PIPE,
11        encoding='utf-8',
12        )
13
```

```
14 ps.stdout.close()
15 output, _ = grep.communicate()
16 python_processes = output.split('\n')
17 print(python_processes)
```

This code recreates the `ps -ef` command and uses `subprocess.Popen` to call it. You capture the output from the command using `subprocess.PIPE`.

For the `grep` command you set its `stdin` to be the output of the `ps` command. You also capture the `stdout` of the `grep` command and set the encoding to `utf-8` as before.

This effectively gets the output from the `ps` command and “pipes” or feeds it into the `grep` command. Next, you `close()` the `ps` command’s `stdout` and use the `grep` command’s `communicate()` method to get output from `grep`.

To finish it up, you split the output on the newline (`\n`), which gives you a `list` of strings that should be a listing of all your active Python processes. If you don’t have any active Python processes running right now, the output will be an empty list.

You can always run `ps -ef` yourself and find something else to search for other than `python` and try that instead.

Wrapping Up

The `subprocess` module is quite versatile and gives you a rich interface to work with external processes.

In this chapter, you learned about:

- The `subprocess.run()` Function
- The `subprocess.Popen()` Class
- The `subprocess.Popen.communicate()` Function
- Reading and Writing with `stdin` and `stdout`

There is more to the `subprocess` module than what is covered here. However, you should now be able to use `subprocess` correctly. Go ahead and give it a try!

Review Questions

1. How would you launch Microsoft Notepad or your favorite text editor with Python?
2. Which method do you use to get the result from a process?
3. How do you get `subprocess` to return strings instead of bytes?

Chapter 26 - Debugging Your Code with pdb

Mistakes in your code are known as “bugs”. You will make mistakes. You will make many mistakes, and that’s totally fine. Most of the time, they will be simple mistakes such as typos. But since computers are very literal, even typos prevent your code from working as intended. So they need to be fixed. The process of fixing your mistakes in programming is known as **debugging**.

The Python programming language comes with its own built-in debugger called `pdb`. You can use `pdb` on the command line or import it as a module. The name, `pdb`, is short for “Python debugger”.

Here is a link to the full documentation for `pdb`:

- <https://docs.python.org/3/library/pdb.html>

In this chapter, you will familiarize yourself with the basics of using `pdb`. Specifically, you will learn the following:

- Starting `pdb` in the REPL
- Starting `pdb` on the Command Line
- Stepping Through Code
- Adding Breakpoints in `pdb`
- Creating a Breakpoint with `set_trace()`
- Using the built-in `breakpoint()` Function
- Getting Help

While `pdb` is handy, most Python editors have debuggers with more features. You will find the debugger in PyCharm or WingIDE to have many more features, such as auto-complete, syntax highlighting, and a graphical call stack.

A call stack is what your debugger will use to keep track of function and method calls. When possible, you should use the debugger that is included with your Python IDE as it tends to be a little easier to understand.

However, there are times where you may not have your Python IDE, for example when you are debugging remotely on a server. It is those times when you will find `pdb` to be especially helpful.

Let’s get started!

Starting pdb in the REPL

The best way to start is to have some code that you want to run pdb on. Feel free to use your own code or a code example from another chapter of this book.

Or you can create the following code in a file named `debug_code.py`:

```
1 # debug_code.py
2
3 def log(number):
4     print(f'Processing {number}')
5     print(f'Adding 2 to number: {number + 2}')
6
7
8 def looper(number):
9     for i in range(number):
10         log(i)
11
12 if __name__ == '__main__':
13     looper(5)
```

There are several ways to start pdb and use it with your code. For this example, you will need to open up a terminal (or `cmd.exe` if you're a Windows user). Then navigate to the folder where you saved your code.

Now start Python in your terminal. This will give you the Python REPL where you can import your code and run the debugger, pdb. Here's how:

```
1 >>> import debug_code
2 >>> import pdb
3 >>> pdb.run('debug_code.looper(5)')
4 > <string>(1)<module>()
5 (Pdb) continue
6 Processing 0
7 Adding 2 to number: 2
8 Processing 1
9 Adding 2 to number: 3
10 Processing 2
11 Adding 2 to number: 4
12 Processing 3
13 Adding 2 to number: 5
14 Processing 4
15 Adding 2 to number: 6
```

The first two lines of code import your code and pdb. To run pdb against your code, you need to use pdb.run() and tell it what to do. In this case, you pass in debug_code.looper(5) as a string. When you do this, the pdb module will transform the string into an actual function call of debug_code.looper(5).

The next line is prefixed with (Pdb). That means you are now in the debugger. Success!

To run your code in the debugger, type continue or c for short. This will run your code until one of the following happens:

- The code raises an exception
- You get to a breakpoint (explained later on in this chapter)
- The code finishes

In this case, there were no exceptions or breakpoints set, so the code worked perfectly and finished execution!

Starting pdb on the Command Line

An alternative way to start pdb is via the command line. The process for starting pdb in this manner is similar to the previous method. You still need to open up your terminal and navigate to the folder where you saved your code.

But instead of opening Python, you will run this command:

```
1 python -m pdb debug_code.py
```

When you run pdb this way, the output will be slightly different:

```
1 > /python101code/chapter26_debugging/debug_code.py(1)<module>()
2 -> def log(number):
3 (Pdb) continue
4 Processing 0
5 Adding 2 to number: 2
6 Processing 1
7 Adding 2 to number: 3
8 Processing 2
9 Adding 2 to number: 4
10 Processing 3
11 Adding 2 to number: 5
12 Processing 4
13 Adding 2 to number: 6
14 The program finished and will be restarted
```

```
15 > /python101code/chapter26_debugging/debug_code.py(1)<module>()
16 -> def log(number):
17 (Pdb) exit
```

The 3rd line of output above has the same (**Pdb**) prompt that you saw in the previous section. When you see that prompt, you know you are now running in the debugger. To start debugging, enter the `continue` command.

The code will run successfully as before, but then you will see a new message:

```
1 The program finished and will be restarted
```

The debugger finished running through all your code and then started again from the beginning! That is handy for running your code multiple times! If you do not wish to run through the code again, you can type `exit` to quit the debugger.

Stepping Through Code

Stepping through your code is when you use your debugger to run one line of code at a time. You can use `pdb` to step through your code by using the `step` command, or `s` for short.

Following is the first few lines of output that you will see if you step through your code with `pdb`:

```
1 $ python -m pdb debug_code.py
2 > /python101code/chapter26_debugging/debug_code.py(3)<module>()
3 -> def log(number):
4 (Pdb) step
5 > /python101code/chapter26_debugging/debug_code.py(8)<module>()
6 -> def looper(number):
7 (Pdb) s
8 > /python101code/chapter26_debugging/debug_code.py(12)<module>()
9 -> if __name__ == '__main__':
10 (Pdb) s
11 > /python101code/chapter26_debugging/debug_code.py(13)<module>()
12 -> looper(5)
13 (Pdb)
```

The first command that you pass to `pdb` is `step`. Then you use `s` to step through the following two lines. You can see that both commands do exactly the same, since “`s`” is a shortcut or alias for “`step`”.

You can use the `next` (or `n`) command to continue execution until the next line within the function. If there is a function call within your function, `next` will **step over** it. What that means is that it will

call the function, execute its contents, and then continue to the **next** line in the current function. This, in effect, steps over the function.

You can use `step` and `next` to navigate your code and run various pieces efficiently.

If you want to step into the `looper()` function, continue to use `step`. On the other hand, if you don't want to run each line of code in the `looper()` function, then you can use `next` instead.

You should continue your session in `pdb` by calling `step` so that you step into `looper()`:

```
1 (Pdb) s
2 --Call--
3 > /python101code/chapter26_debugging/debug_code.py(8)looper()
4 -> def looper(number):
5 (Pdb) args
6 number = 5
```

When you step into `looper()`, `pdb` will print out `--Call--` to let you know that you called the function. Next you used the `args` command to print out all the current args in your namespace. In this case, `looper()` has one argument, `number`, which is displayed in the last line of output above. You can replace `args` with the shorter `a`.

The last command that you should know about is `jump` or `j`. You can use this command to jump to a specific line number in your code by typing `jump` followed by a space and then the line number that you wish to go to.

Now let's learn how you can add a breakpoint!

Adding Breakpoints in `pdb`

A breakpoint is a location in your code where you want your debugger to stop so you can check on variable states. What this allows you to do is to inspect the `callstack`, which is a fancy term for all variables and function arguments that are currently in memory.

If you have PyCharm or WingIDE, then they will have a graphical way of letting you inspect the callstack. You will probably be able to mouse over the variables to see what they are set to currently. Or they may have a tool that lists out all the variables in a sidebar.

Let's add a breakpoint to the last line in the `looper()` function which is **line 10**.

Here is your code again:

```
1 # debug_code.py
2
3 def log(number):
4     print(f'Processing {number}')
5     print(f'Adding 2 to number: {number + 2}')
6
7
8 def looper(number):
9     for i in range(number):
10         log(i)
11
12 if __name__ == '__main__':
13     looper(5)
```

To set a breakpoint in the pdb debugger, you can use the `break` or `b` command followed by the line number you wish to break on:

```
1 $ python3.8 -m pdb debug_code.py
2 > /python101code/chapter26_debugging/debug_code.py(3)<module>()
3 -> def log(number):
4 (Pdb) break 10
5 Breakpoint 1 at /python101code/chapter26_debugging/debug_code.py:10
6 (Pdb) continue
7 > /python101code/chapter26_debugging/debug_code.py(10)looper()
8 -> log(i)
9 (Pdb)
```

Now you can use the `args` command here to find out what the current arguments are set to. You can also print out the value of variables, such as the value of `i`, using the `print` (or `p` for short) command:

```
1 (Pdb) print(i)
2 0
```

Now let's find out how to add a breakpoint to your code!

Creating a Breakpoint with `set_trace()`

The Python debugger allows you to import the `pdb` module and add a breakpoint to your code directly, like this:

```
1 # debug_code_with_settrace.py
2
3 def log(number):
4     print(f'Processing {number}')
5     print(f'Adding 2 to number: {number + 2}')
6
7
8 def looper(number):
9     for i in range(number):
10         import pdb; pdb.set_trace()
11         log(i)
12
13 if __name__ == '__main__':
14     looper(5)
```

Now when you run this code in your terminal, it will automatically launch into pdb when it reaches the `set_trace()` function call:

```
1 $ python3.8 debug_code_with_settrace.py
2 > /python101code/chapter26_debugging/debug_code_with_settrace.py(12)looper()
3 -> log(i)
4 (Pdb)
```

This requires you to add a fair amount of extra code that you'll need to remove later. You can also have issues if you forget to add the semi-colon between the import and the `pdb.set_trace()` call.

To make things easier, the Python core developers added `breakpoint()` which is the equivalent of writing `import pdb; pdb.set_trace()`.

Let's discover how to use that next!

Using the built-in `breakpoint()` Function

Starting in **Python 3.7**, the `breakpoint()` function has been added to the language to make debugging easier. You can read all about the change here:

- <https://www.python.org/dev/peps/pep-0553/>

Go ahead and update your code from the previous section to use `breakpoint()` instead:

```
1 # debug_code_with_breakpoint.py
2
3 def log(number):
4     print(f'Processing {number}')
5     print(f'Adding 2 to number: {number + 2}')
6
7
8 def looper(number):
9     for i in range(number):
10         breakpoint()
11         log(i)
12
13 if __name__ == '__main__':
14     looper(5)
```

Now when you run this in the terminal, Pdb will be launched exactly as before.

Another benefit of using `breakpoint()` is that many Python IDEs will recognize that function and automatically pause execution. This means you can use the IDE's built-in debugger at that point to do your debugging. This is not the case if you use the older `set_trace()` method.

Getting Help

This chapter doesn't cover all the commands that are available to you in `pdb`. So to learn more about how to use the debugger, you can use the `help` command within `pdb`. It will print out the following:

```
1 (Pdb) help
2
3 Documented commands (type help <topic>):
4 ======
5 EOF  c          d          h          list      q          rv         undisplay
6 a    cl         debug      help      11        quit      s          unt
7 alias clear      disable   ignore    longlist r          source    until
8 args commands   display   interact n          restart   step      up
9 b    condition  down     j          next      return   tbreak   w
10 break cont      enable   jump     p          retval   u          whatis
11 bt   continue  exit     l          pp       run      unalias where
12
13 Miscellaneous help topics:
14 =====
15 exec  pdb
```

If you want to learn what a specific command does, you can type `help` followed by the command. Here is an example:

```
1 (Pdb) help where
2 w(here)
3     Print a stack trace, with the most recent frame at the bottom.
4     An arrow indicates the "current frame", which determines the
5     context of most commands. 'bt' is an alias for this command.
```

Go give it a try on your own!

Wrapping Up

Being able to debug your code successfully takes practice. It is great that Python provides you with a way to debug your code without installing anything else. You will find that using `breakpoint()` to enable breakpoints in your IDE is also quite handy.

In this chapter you learned about the following:

- Starting pdb in the REPL
- Starting pdb on the Command Line
- Stepping Through Code
- Creating a Breakpoint with `set_trace()`
- Adding Breakpoints in pdb
- Using the built-in `breakpoint()` Function
- Getting Help

You should go and try to use what you have learned here in your own code. Adding intentional errors to your code and then running them through your debugger is a great way to learn how things work!

Review Questions

1. What is pdb?
2. How do you use pdb to get to a specific location in your code?
3. What is a breakpoint?
4. What is a callstack?

Chapter 27 - Learning About Decorators

Python has lots of neat features built into it, some of them easier to understand than others. One feature that seems to trip beginners up is the concept of **decorators**. A decorator is a function that accepts another function as its argument. The decorator is used to add something new to the function that is passed into the decorator but usually does not modify the original function.

In this chapter, you will cover the following:

- Creating a Function
- Creating a Decorator
- Applying a Decorator with @
- Creating a Decorator for Logging
- Stacking Decorators
- Passing Arguments to Decorators
- Using a Class as a Decorator
- Python's Built-in Decorators
- Python Properties

Before digging into decorators deeply, it's a good idea to review functions briefly.

Let's get started!

Creating a Function

Functions are one of the building blocks of your code. They let you create reusable components in your application. You start creating a function by using the keyword `def` followed by the name of your function. Next, you add some parentheses and, optionally, add parameters inside of them. Finally, you add a colon to the end. After that you would add a block of code that is indented underneath the function name.

In this example, you will use the `pass` keyword to indicate that the function doesn't do anything. If you wanted to add functionality, the code would live in the indented function body instead of the `pass` statement.

```
1 def some_function(arg_one, arg_two):  
2     pass
```

This function takes in two arguments. Functions can take zero to practically any number of arguments.

These arguments can be of any data type. From strings to lists to dictionaries. What is not so commonly known is that you can also pass in another function!

Everything in Python is an object. That includes functions. Because of this fact, you can pass a function, a class, or any other Python data type into other functions and classes.

You can verify that a function is an object by doing the following:

```
1 >>> def some_function(arg_one, arg_two):  
2     ...     pass  
3     ...  
4 >>> type(some_function)  
5 <class 'function'>
```

This shows that your function is in fact an instance of `class function`. What that means is that your function can have methods and attributes.

Let's take a look:

```
1 >>> some_function.__name__  
2 'some_function'  
3 >>> some_function.__doc__  
4 >>> print(some_function.__doc__)  
5 None  
6 >>> def some_function(arg_one, arg_two):  
7     ...     """This is some_function's docstring"""  
8     ...  
9 >>> print(some_function.__doc__)  
10 This is some_function's docstring
```

Now that you know that functions are objects, let's try to create a decorator!

Creating a Decorator

A decorator is basically a nested function that accepts a function as its sole argument. Go ahead and create a function named `func_info()` that looks like this:

```
1 def func_info(func):
2     def wrapper():
3         print('Function name: ' + func.__name__)
4         print('Function docstring: ' + str(func.__doc__))
5         result = func()
6         return result
7     return wrapper
```

This function takes in another function as its argument. Inside of `func_info()`, you create another function named `wrapper()`. This inner function will print out the name of the function that was passed to the decorator as well as its docstring.

Then it will run the function and return its result. Finally the `func_info()` function will return the `wrapper()` function.

Let's create a new function called `treble()` and use your decorator on it:

```
1 def treble():
2     return 3 * 3
3
4 new_treble = func_info(treble)
5 print(new_treble())
```

Here you pass your `treble()` function to your decorator function, `func_info()`. Then you assign the result to the `new_treble` variable. Next, you call `new_treble` which will call `wrapper`. This will print a couple lines and then (finally!) call `func_info`, and return `func_info`'s result. Once the result has been returned, you use Python's `print()` function to print it out.

When you run this code, you will see the following:

```
1 Function name: treble
2 Function docstring: None
3 9
```

Your `func_info` decorator successfully identified the function that was passed to it. You can also see that `treble()` doesn't have a docstring, and it doesn't take any arguments.

The next task is to update the `treble()` function so that it does take an argument and has a docstring.

You will also update the decorator so that it accepts arguments for the function that it is decorating:

```
1 # first_decorator_updated.py
2
3 def func_info(func):
4     def wrapper(*args):
5         print('Function name: ' + func.__name__)
6         print('Function docstring: ' + str(func.__doc__))
7         result = func(*args)
8         return result
9     return wrapper
```

The first step is to add `*args` to the `wrapper` function. The second step is to add `*args` to the `func()` call. This allows the decorator to accept any number of arguments, and pass those arguments on to the decorated function.

Note that the `*args` are passed to the `wrapper` nested function and not `func_info` itself, which still only accepts a function.

Next you update `treble()` to accept a single argument that you want to triple.

```
1 def treble(a):
2     """A function that triples its input"""
3     return a * 3
4
5
6 my_treble = func_info(treble)
7 print(my_treble(5))
```

Now when you call the decorated function, `my_treble`, you pass in the value, 5.

Here is the output:

```
1 Function name: treble
2 Function docstring: A function that triples its input
3 15
```

Python actually has a shorter way to decorate a function.

Let's find out how!

Applying a Decorator with @

You can decorate a function by using a special syntax. The special syntax is the `@` sign followed with the name of the decorator. This line of code is put before the function definition that you are decorating.

Let's update the code to take advantage of this syntax:

```
1 # decorator_syntax.py
2
3 def func_info(func):
4     def wrapper(*args):
5         print('Function name: ' + func.__name__)
6         print('Function docstring: ' + str(func.__doc__))
7         result = func(*args)
8         return result
9     return wrapper
10
11 @func_info
12 def treble(a):
13     """A function that triples its input"""
14     return a * 3
15
16 print(treble(5))
```

This time around, you rewrote the code to use `@func_info` to decorate the `treble()` function. This is equivalent to `func_info = func_info(treble)`.

Using the `@`-syntax is the common way you will encounter decorators being used in Python code. Keep in mind that it's only a more convenient syntax for passing the function object that you are decorating to your decorator function. Because it makes your code sweeter to read but isn't essential, these types of syntax improvements are sometimes called "syntactic sugar".

Now let's go ahead and write a useful decorator!

Creating a Decorator for Logging

Decorators can be used for a wide range of purposes, from adding authentication in web frameworks such as Django or Flask, to logging useful information on function calls.

Python has a robust, thread-safe logging module that you can use. You might want to log the function and its arguments to a file to help you debug issues when the code fails. For example, if you have written a financial application, you will want some way to keep track of transactions. While logging isn't a secure method for that sort of thing, you could create a decorator that would write data to a secure database instead.

However, to keep things bit more simple, you will log to a file in the following example.

Here is the first function you need to create in a new file named `logging_decorator.py`:

```
1 # logging_decorator.py
2
3 import logging
4
5 def log(func):
6     """
7     Log what function is called
8     """
9     def wrapper(*args, **kwargs):
10         name = func.__name__
11         logger = logging.getLogger(name)
12         logger.setLevel(logging.INFO)
13
14         # add logging formatter and file handler
15         logging_formatter(logger, name)
16
17         logger.info(f"Running function: {name}")
18         logger.info(f"{args=}, {kwargs=}")
19         result = func(*args, **kwargs)
20         logger.info("Result: %s" % result)
21         return result
22     return wrapper
```

Note: You do not need to understand everything that is going on in this code or the `logging_formatter()` function below. This example is used because it is a common use-case for decorators.

This decorator works the same way as before: it takes in a function to process, and creates a wrapper function to handle the processing (including forwarding any positional and keyword arguments). The difference here is that you will use Python's logging module to write to a log file instead of printing to the screen. To do that, you import `logging`, create a logger via `logging.getLogger()` and set the logging level to the `info` level.



There are other levels of logging that you can use as well. However you don't need to worry about that right now. If you are really interested, then you should check out the documentation here:

- <https://docs.python.org/3/library/logging.html>

Next you call `logging_formatter()` to format your log and tell it where to save your log. Finally, you log some information about the function you are decorating.

Now you can add `logging_formatter()`:

```
1 def logging_formatter(logger, name):
2     """
3     Format logger and add file handler
4     """
5     fh = logging.FileHandler(f"{name}.log")
6     fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
7     formatter = logging.Formatter(fmt)
8     fh.setFormatter(formatter)
9     logger.addHandler(fh)
```

This code will take the name of the function that is being decorated and create a log file based on that name. Then you create a format string. This is used in the log to add a timestamp, the name of the function, the logging level name, and the logging message. Finally you set the logging formatter to use the formatting string you defined and add a file handler to the logger itself so that it writes to disk.

The last step is to add the following code to the end of your script:

```
1 @log
2 def treble(a):
3     return a * 3
4
5 if __name__ == '__main__':
6     treble(5)
```

This shows how to decorate `treble()` with your `log()` decorator.

When you run the code, you should get a file named `treble.log` in the same folder as your program that contains something similar to the following:

```
1 2020-05-04 16:15:19,980 - treble - INFO - Running function: treble
2 2020-05-04 16:15:19,980 - treble - INFO - args=(5,), kwargs={}
3 2020-05-04 16:15:19,980 - treble - INFO - Result: 15
```

This shows you the timestamp of when the log line was written. It also shows the name of the function, log level, and the messages that your logging function created.

Stacking Decorators

Most of the time, you will only use one decorator per function. But that's not a requirement. You can apply multiple decorators to a single function.

Let's look at an example:

```
1 def bold(func):
2     def wrapper():
3         return "<b>" + func() + "</b>"
4     return wrapper
5
6 def italic(func):
7     def wrapper():
8         return "<i>" + func() + "</i>"
9     return wrapper
10
11 @bold
12 @italic
13 def formatted_text():
14     return 'Python rocks!'
15
16 print(formatted_text())
```

The `bold()` and `italic()` decorators will add `` and `<i>` HTML tags to the returned value of the function that they are decorating.

The order of the decorators is important: Python applies the decorators bottom-up (i.e. closest to the function first), but calls the decorated function top-down (i.e. as it appears in the source). So in this example, `formatted_text` will be defined, then the `italic` decorator will be applied, and then the `bold`() decorator will be applied; when the decorated `format_text` function is called, `bold` will run, then `italic` will run, and finally the original `format_text` will run.

When you run the code above, you will see the following output:

```
1 <b><i>Python rocks!</i></b>
```

To give you a clearer picture of what is happening when you stack multiple decorators, let's rewrite this example so that it prints out the name of the function that each decorator is being applied to:

```
1 def bold(func):
2     print(f'You are wrapping {func.__name__} in bold')
3     def bold_wrapper():
4         return "<b>" + func() + "</b>"
5     return bold_wrapper
6
7 def italic(func):
8     print(f'You are wrapping {func.__name__} in italic')
9     def italic_wrapper():
10        return "<i>" + func() + "</i>"
11    return italic_wrapper
```

```
12
13 @bold
14 @italic
15 def formatted_text():
16     return 'Python rocks!'
17
18 print(formatted_text())
```

When you run this code, you will see this output:

```
1 You are wrapping formatted_text in italic
2 You are wrapping italic_wrapper in bold
3 <b><i>Python rocks!</i></b>
```

This demonstrates that `formatted_text()` is being wrapped by the `italic()` decorator. Then the `italic()` decorator's `italic_wrapper()` is being decorated by the `bold()` decorator.

Now let's learn how to pass an argument to a decorator!

Passing Arguments to Decorators

There are times when you might want to pass one or more arguments to the decorator itself. Let's take the `func_info()` decorator from earlier and try giving it an argument to see if it works:

```
1 # decorator_syntax_with_arguments.py
2
3 def func_info(func):
4     def wrapper(*args):
5         print('Function name: ' + func.__name__)
6         print('Function docstring: ' + str(func.__doc__))
7         result = func(*args)
8         return result
9     return wrapper
10
11 @func_info(2)
12 def treble(a: int) -> int:
13     """A function that triples its input"""
14     return a * 3
15
16 print(treble(5))
```

If you try to run this code, you will end up receiving an error:

```
1 builtins.AttributeError: 'int' object has no attribute '__name__'
```

That might be a bit confusing. The reason you are seeing this exception is that you passed an integer to the decorator. So your decorator is attempting to decorate the integer instead of the function.

Let's rewrite the code so that it looks like this:

```
1 # decorator_syntax_with_arguments.py
2
3 def func_info(arg1, arg2):
4     print('Decorator arg1 = ' + str(arg1))
5     print('Decorator arg2 = ' + str(arg2))
6
7     def the_real_decorator(function):
8
9         def wrapper(*args, **kwargs):
10             print('Function {} args: {} kwargs: {}'
11                   .format(
12                         function.__name__,
13                         str(args),
14                         str(kwargs)))
15             return function(*args, **kwargs)
16         return wrapper
17
18     return the_real_decorator
19
20 @func_info(3, 'Python')
21 def treble(number):
22     return number * 3
23
24 print(treble(5))
```

This time when you write your decorator you will need to have a function inside a function inside a function. Yes, you read that right! You have nested functions! Your `func_info()` now takes arguments, and those arguments can be used by `the_real_decorator()` as well as by `wrapper`.

The `wrapper()` function is rewritten to print out the function name, `args`, and `kwargs` all on one line. The rest of the code is pretty much the same as before except that you are returning `the_real_decorator`, which will return `wrapper`.

When you run this code, you should see the following output:

```
1 Decorator arg1 = 3
2 Decorator arg2 = Python
3 Function treble args: (5,) kwargs: {}
4 15
```

You might find that using nested functions is hard to wrap your mind around. Fortunately, there is another way to create a decorator that takes arguments.

Let's find out how!

Using a Class as a Decorator

There is a nicer way to create a decorator that can take arguments. That method is by creating a special class.

Go ahead and create a new file named `decorator_class.py` and add the following code to it:

```
1 # decorator_class.py
2
3 class info:
4
5     def __init__(self, arg1, arg2):
6         print('running __init__')
7         self.arg1 = arg1
8         self.arg2 = arg2
9         print('Decorator args: {}, {}'.format(arg1, arg2))
10
11    def __call__(self, function):
12        print('in __call__')
13
14        def wrapper(*args, **kwargs):
15            print('in wrapper()')
16            return function(*args, **kwargs)
17
18        return wrapper
19
20 @info(3, 'Python')
21 def treble(number):
22     return number * 3
23
24 print()
25 print(treble(5))
```

It is **not** normal to create a class with a name that is completely lowercase. You would normally create a class with title-case. But to be consistent with the other examples, you will name it using lowercase.

This class takes only two arguments. You could use `*args` and `**kwargs` too, but it would take more work. This example is meant to be simple and easy to follow. When this class is instantiated, it will save off the two arguments to instance variables and print out a message to let you know what they are.

The next step is to create a `__call__()` method. This special method makes the class callable. It takes in the function that you are decorating as its argument. Then inside of this special method, you nest the `wrapper()` function. Note that this function does **not** have `self` passed to it. Instead it receives arguments and keyword arguments to the function that is being decorated.

The rest of the code is pretty much the same as what you saw in the previous example.

When you run this code, you should see this output:

```
1 in __init__
2 Decorator args: 3, Python
3 in __call__
4
5 in wrapper()
6 15
```

Now that you know how to create a decorator that can take arguments both as a nested function and as a class, you are ready to learn about some of Python's built-in decorators!

Python's Built-in Decorators

Python comes with several decorators built-in. There are three that you can use without importing anything. They are helpful when defining classes.

Those three are as follows:

- `classmethod`
- `staticmethod`
- `property`

There are other decorators in Python. One popular example is in the `functools` module. It is called `wraps`, which is a nice decorator to use to maintain the original function's name and docstring. When you create a decorator, it is essentially replacing the original function with a new one, so its name and docstring can become obscured.

You can use `functools.wraps` to fix that issue.

Other popular decorators are `functools.lru_cache` and some interesting ones in `contextlib`. If you have the time, you should definitely go look up both of these libraries in the documentation.

The `classmethod` decorator is used on a class' method that does not need any instance data to do its job; because those methods do not need instance data their first argument is not `self`, it is the class instead (commonly abbreviated as `cls`, since `class` is a keyword). One common use for `classmethods` is as alternate constructors. Create a new file named `classmethod_example.py` and add the following:

```
1 # classmethod_example.py
2
3 class Time:
4     """an example 24-hour time class"""
5
6     def __init__(self, hour, minute):
7         self.hour = hour
8         self.minute = minute
9
10    def __repr__(self):
11        return "Time(%d, %d)" % (self.hour, self.minute)
12
13    @classmethod
14    def from_float(cls, moment):
15        """2.5 == 2 hours, 30 minutes, 0 seconds, 0 microseconds"""
16        hours = int(moment)
17        if hours:
18            moment = moment % hours
19        minutes = int(moment * 60)
20        return cls(hours, minutes)
21
22    def to_float(self):
23        """return self as a floating point number"""
24        return self.hour + self.minute / 60
25
26 Time(7, 30)
27 Time.from_float(5.75)
28 t = Time(10, 15)
29 t.to_float()
```

When you run this code, you should see this output:

```
1 Time(7, 30)
2 Time(5, 45)
3 10.25
```

The `staticmethod` decorator tells Python that the method does not need any class nor instance data; consequently, you do not need to specify `self` nor `cls` as the first parameter. You can still call `staticmethods` in both of the ways that you've seen in the example above: you can use the class directly or you can first create an instance.

The most useful of these three decorators is `property`. Let's find out why in the next section!

Python Properties

Python properties are a way of making a class function into an attribute. They also let you create a kind of private attribute in Python by using a getter and a setter. Python doesn't really have private methods and attributes as you can still access items that begin with one or two underscores. However, by convention, when you start an instance variable or method with one or two underscores, it is considered to be private. If you use those types of variables and methods then your code may break when those variables/methods change or go away in newer releases.

A "getter" refers to a function that you use to indirectly access an attribute that is private whereas a "setter" is a function that you would use to set a private attribute. Python doesn't really have this concept because you can access and change an attribute directly in most cases.

Let's get started by writing a class where you implement your own setter and getter methods:

```
1 class Amount:
2
3     def __init__(self):
4         # private attribute
5         self._amount = None
6
7     def get_amount(self):
8         return self._amount
9
10    def set_amount(self, value):
11        if isinstance(value, int) or isinstance(value, float):
12            self._amount = value
13        else:
14            print(f'Value must be an int or float')
15
16 if __name__ == '__main__':
17     amt = Amount()
```

```
18     print(f'The current amount is {amt.get_amount()}')
19     amt.set_amount('the')
20     print(f'The current amount is {amt.get_amount()}')
21     amt.set_amount(5.5)
22     print(f'The current amount is {amt.get_amount()}')
```

In this example, you create the private `self._amount` instance attribute and set it to `None`. To access this value, you create a getter method called `get_amount()`. Then you create a setter method called `set_amount()` that takes a value to set the amount to. The setter method will also do some basic error checking for you.

The last piece of code tests out the setter and getter. When you run this code, you will end up with the following output:

```
1 The current amount is None
2 Value must be an int or float
3 The current amount is None
4 The current amount is 5.5
```

Python makes this easier by giving you the ability to turn the getter and setter methods into a property.

Let's update the code by adding a single line at the end of the class:

```
1 class Amount:
2
3     def __init__(self):
4         # private attribute
5         self._amount = None
6
7     def get_amount(self):
8         return self._amount
9
10    def set_amount(self, value):
11        if isinstance(value, int) or isinstance(value, float):
12            self._amount = value
13        else:
14            print(f'Value must be an int or float')
15
16    amount = property(get_amount, set_amount)
17
18 if __name__ == '__main__':
19     amt = Amount()
20     print(f'The current amount is {amt.amount}')
```

```
21     amt.amount = 'the'
22     print(f'The current amount is {amt.amount}')
23     amt.amount = 5.5
24     print(f'The current amount is {amt.amount}')
```

Here you tell Python that you want to add a new property or attribute to the `Amount` class called `amount`. You create it using the `property` function, which accepts a setter, getter, and a delete method, which you can ignore for this case.

The other change here is that now you can access and set the `amount` directly rather than calling `get_amount()` and `set_amount()` methods.

You can simplify this even more by using `property` as a decorator.

Let's update the class one more time to use the decorator syntax:

```
1  class Amount:
2
3      def __init__(self):
4          # private attribute
5          self._amount = None
6
7      @property
8      def amount(self):
9          return self._amount
10
11     @amount.setter
12     def amount(self, value):
13         if isinstance(value, int) or isinstance(value, float):
14             self._amount = value
15         else:
16             print(f'Value must be an int or float')
17
18
19 if __name__ == '__main__':
20     amt = Amount()
21     print(f'The current amount is {amt.amount}')
22     amt.amount = 'the'
23     print(f'The current amount is {amt.amount}')
24     amt.amount = 5.5
25     print(f'The current amount is {amt.amount}')
```

This time around you change the `get_amount()` method into an `amount()` method and decorate it with `@property`. Then you create a setter for it by creating a second `amount()` method but you decorate with `amount.setter`. The code looks a bit odd, but it works. Give it a try and see for yourself!

One really nice benefit of property is that you can create your class using simple attributes, and later turn them into properties if you discover you need extra processing when getting or setting those variables.

Wrapping Up

Decorators are a powerful tool that can be a little confusing to implement. Once you get the hang of decorators, you will find them quite helpful in many different use cases.

In this chapter you learned about the following:

- Creating a Function
- Creating a Decorator
- Applying a Decorator with @
- Creating a Decorator for Logging
- Stacking Decorators
- Passing Arguments to Decorators
- Using a Class as a Decorator
- Python's Built-in Decorators
- Python Properties

To practice creating and using decorators, you can create some decorators for your own code or play around with the decorators in this chapter. You could write a decorator to time your code's execution for example. Have fun!

Review Questions

1. What is a decorator?
2. What special syntax do you use to apply a decorator?
3. Name at least two of Python's built-in decorators
4. What is a Python property? Why is it useful?

Chapter 28 - Assignment Expressions

Assignment expressions were added to Python in 3.8. The general idea is that an assignment expression allows you to assign to variables within an expression.

The syntax for doing this is:

```
1 NAME := expr
```

This operator has been called the “walrus operator”, although their real name is “assignment expression”. Interestingly, the CPython internals also refer to them as “named expressions”.

You can read all about assignment expressions in PEP 572:

- <https://www.python.org/dev/peps/pep-0572/>

Let's find out how to use assignment expressions!

Using Assignment Expressions

Assignment expressions are still relatively rare. However, you need to know about assignment expressions because you will probably come across them from time-to-time. PEP 572 has some good examples of assignment expressions.

```
1 # Handle a matched regex
2 if (match := pattern.search(data)) is not None:
3     ...
4
5 # A more explicit alternative to the 2-arg form of iter() invocation
6 while (value := read_next_item()) is not None:
7     ...
8
9 # Share a subexpression between a comprehension filter clause and its output
10 filtered_data = [y for x in data if (y := f(x)) is not None]
```

In these 3 examples, you are creating a variable in the expression statement itself. The first example creates the variable `match` by assigning it the result of the regex pattern search. The second example assigns the variable `value` to the result of calling a function in the `while` loop's expression. Finally, you assign the result of calling `f(x)` to the variable `y` inside of a list comprehension.

It would probably help to see the difference between code that doesn't use an assignment expression and one that does. Here's an example of reading a file in chunks:

```
1 with open(some_file) as file_obj:
2     while True:
3         chunk_size = 1024
4         data = file_obj.read(chunk_size)
5         if not data:
6             break
7         if 'something' in data:
8             # process the data somehow here
```

This code will open up a file of indeterminate size and process it 1024 bytes at a time. You will find this useful when working with very large files as it prevents you from loading the entire file into memory. If you do, you can run out of memory and cause your application or even the computer to crash.

You can shorten this code up a bit by using an assignment expression:

```
1 with open(some_file) as file_obj:
2     chunk_size = 1024
3     while data := file_obj.read(chunk_size):
4         if 'something' in data:
5             # process the data somehow here
```

Here you assign the result of the `read()` to `data` within the `while` loop's expression. This allows you to then use that variable inside of the `while` loop's code block. It also checks that some data was returned so you don't have to have the `if not data: break` stanza.

Another good example that is mentioned in PEP 572 is taken from Python's own `site.py`. Here's how the code was originally:

```
1 env_base = os.environ.get("PYTHONUSERBASE", None)
2 if env_base:
3     return env_base
```

And this is how it could be simplified by using an assignment expression:

```
1 if env_base := os.environ.get("PYTHONUSERBASE", None):
2     return env_base
```

You move the assignment into the conditional statement's expression, which shortens the code up nicely.

Now let's discover some of the situations where assignment expressions can't be used.

What You Cannot Do With Assignment Expressions

There are several cases where assignment expressions cannot be used.

One of the most interesting features of assignment expressions is that they can be used in contexts that an assignment statement cannot, such as in a `lambda` or the previously mentioned comprehension. However, they do NOT support some things that assignment statements can do. For example, you cannot do multiple target assignment:

```
1 x = y = z = 0 # Equivalent, but non-working: (x := (y := (z := 0)))
```

Another prohibited use case is using an assignment expression at the top level of an expression statement. Here is an example from PEP 572:

```
1 y := f(x) # INVALID
2 (y := f(x)) # Valid, though not recommended
```

There is a detailed list of other cases where assignment expressions are prohibited or discouraged in the PEP. You should check that document out if you plan to use assignment expressions often.

Wrapping Up

Assignment expressions are an elegant way to clean up certain parts of your code. The feature's syntax is kind of similar to type hinting a variable. Once you have the hang of one, the other should become easier to do as well.

In this chapter, you saw some real world examples of using the “walrus operator”. You also learned when assignment expressions shouldn't be used. This syntax is only available in Python 3.8 or newer, so if you happen to be forced to use an older version of Python, this feature won't be of much use to you.

Review Questions

1. What is an assignment expression?
2. How do you create an assignment expression?
3. Why would you use assignment expressions?

Chapter 29 - Profiling Your Code

Code is never perfect. You will find that when you write a lot of code, you will often need to go back to improve it. For example, let's say you create a web app and it becomes popular a few months after launch. Now that it's popular and handling a lot more requests, the app is running much slower under load.

How do you figure out what is making the web app so slow? One way to find the problem is to profile your code. Profiling code determines where the bottlenecks are in your code. Then you have to figure out how to fix the issue(s). Sometimes the fix is to rewrite the code. Other times you need to upgrade the hardware.

Python comes with its own profiling modules called `profile` and `cProfile`. Both of these modules have the same interface with `cProfile` being written in C. You will learn about `cProfile` because it doesn't have the significantly larger overhead that `profile` adds.

In this chapter, you will learn about the following:

- Profiling with `cProfile`
- Working with Profile Data Using `pstats`
- Other Profilers

Let's get started!

Learning How to Profile with `cProfile`

The first step is finding something to profile. Fortunately, it is quite easy to get started. You can import pretty much any module in Python and profile it with `cProfile`.

Let's try using Python's `secrets` module, which you can use for generating cryptographically strong random numbers:

```
1 >>> import secrets
2 >>> import cProfile
3 >>> cProfile.run("secrets.token_bytes(16)")
4     5 function calls in 0.000 seconds
5
6 Ordered by: standard name
7
8   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
9       1    0.000    0.000    0.000    0.000 <string>:1(<module>)
10      1    0.000    0.000    0.000    0.000 secrets.py:35(token_bytes)
11      1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
12      1    0.000    0.000    0.000    0.000 {built-in method posix.urandom}
13      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler\
14 ' objects}
```

Here you import `secrets` and `cProfile`. Then you profile `secrets.token_bytes(16)` by wrapping it in quotes and passing it to `cProfile.run()`. You can see the output from that call above.

Let's go over each piece of data in the output:

- `ncalls` - the call count, or the number of times something was called
- `tottime` - the total time spent in the given function
- `percall` - the quotient of the `tottime` divided by `ncalls`
- `cumtime` - the cumulative time spent in the function and all its sub-functions.
- `percall` - the second `percall` is the quotient of the `cumtime` divided by the number of primitive calls
- `filename:lineno(function)` - provides filename and line number information for each call

For those that are wondering, a “primitive” call is one that wasn’t the result of recursion.

This particular example isn’t very interesting as everything is only called once and there are no obvious bottlenecks here.

Let's write some code that you can profile to see how profiling might actually be useful!

Profiling a Python Script with `cProfile`

When you write code, you will always have some functions that are slower than others. One way to simulate slow functions is to use Python’s `time` module, which has a `sleep()` function.

Let's write a program that has several functions that `sleep()` various amounts of time called `profile_test.py`:

```
1 # profile_test.py
2
3 import time
4
5 def quick():
6     print('Running quick')
7     return 1 + 1
8
9 def average():
10    print('Running average')
11    time.sleep(0.5)
12
13 def super_slow():
14    print('Running super slowly')
15    time.sleep(2)
16
17 def main():
18     quick()
19     super_slow()
20     quick()
21     average()
22
23 if __name__ == '__main__':
24     main()
```

Here you have four functions:

- `quick()` - which doesn't `sleep()` at all
- `average()` - which sleeps for half a second
- `super_slow()` - which sleeps for 2 seconds
- `main()` - which is the main entry point of your program

The `main()` function runs all the other functions in your program. It also runs `quick()` twice.

You can profile this code the same way that you did in the previous section:

```
1  >>> import profile_test
2  >>> import cProfile
3  >>> cProfile.run("profile_test.main()")
4  Running quick
5  Running super slowly
6  Running quick
7  Running average
8      14 function calls in 2.505 seconds
9
10 Ordered by: standard name
11
12 ncalls  tottime  percall  cumtime  percall   filename:lineno(function)
13      1    0.000    0.000    2.505    2.505 <string>:1(<module>)
14      1    0.000    0.000    2.003    2.003 profile_test.py:13(super_slow)
15      1    0.000    0.000    2.505    2.505 profile_test.py:17(main)
16      2    0.000    0.000    0.000    0.000 profile_test.py:5(quick)
17      1    0.000    0.000    0.502    0.502 profile_test.py:9(average)
18      1    0.000    0.000    2.505    2.505 {built-in method builtins.exec}
19      4    0.000    0.000    0.000    0.000 {built-in method builtins.print}
20      2    2.505    1.253    2.505    1.253 {built-in method time.sleep}
21      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler\
22 ' objects}
```

This output shows that the `quick()` function was called twice. You can also see that `print()` was called four times and `time.sleep()` was called two times. Everything else was called once. Note that depending on which shell you use to run this code in, you may receive much more output. However, you should see output similar to the above if you are running using the default Python console with the standard Python installation.

The `super_slow()` function took two seconds to run while the `average()` function took half a second to run. You can see that the `main()` function took a little over 2.5 seconds to run because it called all the other functions.

By analyzing this data, you can see that the `average()` and the `super_slow()` functions are the ones that you should focus on to improve the performance of your program.

You can also profile your code on the command line like this:

```
1 $ python -m cProfile profile_test.py
2 Running quick
3 Running super slowly
4 Running quick
5 Running average
6     14 function calls in 2.505 seconds
7
8     Ordered by: standard name
9
10    ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
11        1    0.000    0.000    2.000    2.000  profile_test.py:13(super_slow)
12        1    0.000    0.000    2.505    2.505  profile_test.py:17(main)
13        1    0.000    0.000    2.505    2.505  profile_test.py:3(<module>)
14        2    0.000    0.000    0.000    0.000  profile_test.py:5(quick)
15        1    0.000    0.000    0.504    0.504  profile_test.py:9(average)
16        1    0.000    0.000    2.505    2.505  {built-in method builtins.exec}
17        4    0.000    0.000    0.000    0.000  {built-in method builtins.print}
18        2    2.505    1.252    2.505    1.252  {built-in method time.sleep}
19        1    0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler\
20 ' objects}
```

Here you call Python with the `-m` flag, which tells Python to load the module following the flag. Then you pass `cProfile` the program you want it to profile. The output from this command is nearly identical to the output that you saw before.

If you want to, you can tell `cProfile` to save the output to a file instead. When you do this, you will have extra data that is not shown in the regular output. To save to a file, you can use the `-o` flag, like this:

```
1 $ python -m cProfile -o profile_output.txt profile_test.py
```

Here you save the profiling output to a file named `profile_output.txt`. If you attempt to open this file in a text editor, you will see that it is not human readable.

To make heads or tails of the data in your new file, you will need to use Python's `pstats` module. Let's find out how to do that next!

Working with Profile Data Using `pstats`

Python has its own separate library that you can use to analyze the output from `cProfile` that is called `pstats`. You can use `pstats` to sort and filter the profiling data that you have collected.

Let's try to recreate the regular output that you get from `cProfile` using the `pstats` module by creating a file named `formatted_output.py`:

```
1 import pstats
2
3 def formatted_stats_output(path):
4     p = pstats.Stats(path)
5     stripped_dirs = p.strip_dirs()
6     sorted_stats = stripped_dirs.sort_stats('filename')
7     sorted_stats.print_stats()
8
9 if __name__ == '__main__':
10    path = 'profile_output.txt'
11    formatted_stats_output(path)
```

In this example, you import `pstats` and load the file, `profile_output.txt`, that you created in the last section into `Stats()`. You then use `strip_dirs()` to remove the paths to the files since you don't want the fully qualified paths in the output. Note that when you do this, you modify the `Stats` object and the stripped information is lost.

Next you use `sort_stats('filename')` to sort the profile stats by filename. You can sort the profile stats by using any of the following strings:

- 'calls' - call count
- 'cumulative' - cumulative time
- 'cumtime' - cumulative time
- 'file' - file name
- 'filename' - file name
- 'module' - file name
- 'ncalls' - call count
- 'pcalls' - primitive call count
- 'line' - line number
- 'name' - function name
- 'nfl' - name/file/line
- 'stdname' - standard name
- 'time' - internal time
- 'totime' - internal time

There are also `SortKey` enums for most of these that you could use instead of the strings.

Now you are ready to print it out via `print_stats()`.

Here is the output:

```
1 Tue May 19 21:00:38 2020      profile_output.txt
2
3     14 function calls in 2.502 seconds
4
5 Ordered by: file name
6
7 ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
8     2    0.000    0.000    0.000    0.000  profile_test.py:5(quick)
9     1    0.000    0.000    0.501    0.501  profile_test.py:9(average)
10    1    0.000    0.000    2.000    2.000  profile_test.py:13(super_slow)
11    1    0.000    0.000    2.502    2.502  profile_test.py:17(main)
12    1    0.000    0.000    2.502    2.502  profile_test.py:3(<module>)
13    1    0.000    0.000    2.502    2.502  {built-in method builtins.exec}
14    4    0.000    0.000    0.000    0.000  {built-in method builtins.print}
15    2    2.502    1.251    2.502    1.251  {built-in method time.sleep}
16    1    0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler\
17 ' objects}
```

It's not quite the same output, but it's really close! You should try using some of the strings mentioned above to sort the output in other ways.

If you want to filter the output, you can pass a regular expression to `print_stats()` instead of leaving it blank. A regular expression is a sequence of characters that can be used to search for something. They are in many ways their own language. Python has support for regular expressions in the `re` module.

To filter the output of the profile stats to only those that refer to `main()`, you would do the following:

```
1 sorted_stats.print_stats('\\(main')
```

The regular expression here includes the left parentheses which tells Python to match against the function name portion of the stats. If you re-run the example with the changed code, you will see the following:

```
1 Tue May 19 21:00:38 2020      profile_output.txt
2
3     14 function calls in 2.502 seconds
4
5 Ordered by: file name
6 List reduced from 9 to 1 due to restriction <'\\(main'>
7
8 ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
9     1    0.000    0.000    2.502    2.502  profile_test.py:17(main)
```

You can also tell `pstats` to print out the caller and callee information for specific functions in the stats that you have collected. For example, let's write some code to print out who calls `main()` and also what `main()` itself calls by modifying the code file from earlier:

```
1 import pstats
2
3 def formatted_stats_output(path):
4     p = pstats.Stats(path)
5     stripped_dirs = p.strip_dirs()
6     sorted_stats = stripped_dirs.sort_stats('filename')
7     sorted_stats.print_callers('\(\main')
8     sorted_stats.print_callees('\(\main')
9
10 if __name__ == '__main__':
11     path = 'profile_output.txt'
12     formatted_stats_output(path)
```

Once again you use the '\(\main' regular expression to limit the output to the `main()` function. When you run this code, you will see the following:

```
1 Ordered by: file name
2 List reduced from 9 to 1 due to restriction <'\(\main'>
3
4 Function           was called by...
5                      ncalls  tottime  cumtime
6 profile_test.py:17(main)  <-      1    0.000   2.502  profile_test.py:3(<module>)
7
8
9 Ordered by: file name
10 List reduced from 9 to 1 due to restriction <'\(\main'>
11
12 Function           called...
13                      ncalls  tottime  cumtime
14 profile_test.py:17(main)  ->      2    0.000   0.000  profile_test.py:5(quick)
15                           1    0.000   0.501  profile_test.py:9(average)
16                           1    0.000   2.000  profile_test.py:13(super_slo\
17 w)
```

The callees block shows that `main()` called `quick()`, `average()` and `super_slow()`, which demonstrates that you were able to trace the callees successfully.

Other Profilers

There are several other profiling packages for Python that you can use besides the built-in `cProfile`. Here are a few of the more popular ones:

- `line_profiler` - <https://pypi.org/project/line-profiler/>
- `memory_profiler` - <https://pypi.org/project/memory-profiler/>
- `profilehooks` - <https://pypi.org/project/profilehooks/>
- `scalene` - <https://github.com/emeryberger/scalene>
- `snakeviz` - <https://pypi.org/project/snakeviz/>

This book won't dig into all these different profilers; however, you can learn about some of these in the sequel to this book, **Python 201: Intermediate Python**.

To give you a quick overview, let's go over how each of these packages is different from `cProfile`.

line_profiler

The `line_profiler` package is designed for profiling the time that each individual line takes to execute. It comes with a command-line tool called `kernprof` that you can use in addition to adding `line_profiler` to your code.

The output that it generates is a bit more informative than what you get with `cProfile` because it will output your code along with the time it took to run each line, which can be really helpful in pinpointing which piece is causing the bottleneck(s) in your code.

memory_profiler

While `line_profiler` and `cProfile` are great for finding issues with slow-running code, they cannot find memory issues. For that you can use `memory_profiler`. The `memory_profiler` package goes line-by-line through your code and finds which line or lines are the most memory-intensive.

Its output and API is very similar to that of `line_profiler`, so if you are going to use `memory_profiler`, you may want to check out `line_profiler` too. They are both really handy for finding issues with your code!

profilehooks

The `profilehooks` package gives you a set of decorators to use on your code to help you profile the code. The output from using this package is nearly identical to what you get from `cProfile`.

scalene

The `scalene` package is both a CPU and memory profiler for Python. It claims to run much faster than any other Python profiler and has an overhead of 10-20% or less when compared with other profiler packages. You can think of `scalene` as a combination of `line_profiler` and `memory_profiler`, but faster. It does line-by-line profiling for CPU and memory, just like those other packages.

The output of this project is similar to the `line_profiler` and `memory_profiler`, but can include additional columns of information.

snakeviz

The `snakeviz` project is a browser based graphical viewer for the output of Python's `cProfile` module and an alternative to using the standard library `pstats` module. It will let you visualize the stats output that you create using `cProfile` in nice graphs or other types of visualizations.

This may help you find the poorly written code faster if you are a visual person.

Wrapping Up

Profiling your code is a good way to find out why your code isn't running as well as you had wanted. Once you know what the problem is, you can find a solution to the issue.

In this chapter, you learned about:

- Profiling with `cProfile`
- Working with Profile Data Using `pstats`
- Other Profilers

Remember, your code isn't always the problem. The computer itself may be the issue, or the internet connection, or something else entirely. The bottlenecks in programming can be quite varied.

Review Questions

1. What does "profiling" mean in a programming context?
2. Which Python library do you use to profile your code?
3. How do you extract data from saved profile statistics?

Chapter 30 - An Introduction to Testing

There are many things a good programmer needs to know how to do. Besides knowing the basic syntax of the language that you use and following proper coding conventions, **testing** your code is one of the most important things to learn in order to become a good programmer. Testing your code allows you to know that your code works the way that you intend. Good testing will allow you to modify your code in the future without breaking existing functionality.

Python has two testing libraries built-in that you will be learning about in this chapter:

- doctest
- unittest

Python also includes a `mock` sub-library for `unittest`. You won't be learning about that one here as it's beyond the scope of this book. You can use `mock` to replace parts of your system that are under test. For example, you can use `mock` to simulate a database connection so that you can still test a fake connection rather than accessing the database itself and possibly causing an issue.

In this chapter, you will learn about the following:

- Using doctest in the Terminal
- Using doctest in Your Code
- Using doctest From a Separate File
- Using unittest For Test Driven Development

Let's get started!

Using doctest in the Terminal

According to Python's documentation, the `doctest` module will search your code for "for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown."

You can use `doctest` for the following:

- To check that docstrings are correct and up-to-date
- To write tutorial documentation for a package

- To perform regression testing

Regression testing is a term that is used to describe running tests after you make changes to verify that your changes didn't break existing functionality.

If you would like to check out the doctest documentation, you will find it below:

- <https://docs.python.org/3/library/doctest.html>

To get started, you will need some code to use doctest on. You can create a file named `add_doctest.py` and add the following code to it:

```
1 # add_doctest.py
2
3 def add(a: int, b: int) -> int:
4     """
5     >>> add(1, 2)
6     3
7     >>> add(4, 5)
8     9
9     """
10    a + b
```

Your `add()` function accepts two integers and it should return an integer. You also added two tests in the docstring. The docstring shows what arguments were passed into the `add()` function as well as what should be returned.

However, there is an issue with your code. Do you see what it is? If you don't, that's okay. You're going to run doctest on this code and it will tell you what's wrong.

To run doctest, open up a terminal and navigate to where you saved your Python file. Then run the following command: `python3 -m doctest add_doctest.py`

You should see the following output:

```
1 Computer:chapter30_testing$ python3 -m doctest add_doctest.py
2 ****
3 File "/chapter30_testing/add_doctest.py", line 3, in add
4 Failed example:
5     add(1, 2)
6 Expected:
7     3
8 Got nothing
9 ****
```

```
10 File "/chapter30_testing/add_doctest.py", line 5, in add_doctest.add
11 Failed example:
12     add(4, 5)
13 Expected:
14     9
15 Got nothing
16 ****
17 1 items had failures:
18     2 of  2 in add_doctest.add
19 ***Test Failed*** 2 failures.
20 Computer:chapter30_testing$
```

Both of your tests failed. The output tells you that the reason the tests failed is that they expected a return value, but “Got nothing”. You should update your code so that it returns the addition of the two arguments:

```
1 # add_doctest_working.py
2
3 def add(a: int, b: int) -> int:
4     """
5         >>> add(1, 2)
6         3
7         >>> add(4, 5)
8         9
9         """
10    return a + b
```

Now if you run the doctest command again, you will see no output whatsoever. That means that all your tests passed. If you would like to see output even when your tests succeed, you can pass the `-v` flag to doctest, like this:

```
1 Computer:chapter30_testing$ python3 -m doctest -v add_doctest_working.py
2 Trying:
3     add(1, 2)
4 Expecting:
5     3
6 ok
7 Trying:
8     add(4, 5)
9 Expecting:
10    9
11 ok
```

```
12 1 items had no tests:  
13     add_doctest_working  
14 1 items passed all tests:  
15     2 tests in add_doctest_working.add  
16 2 tests in 2 items.  
17 2 passed and 0 failed.  
18 Test passed.  
19 Computer:chapter30_testing$
```

Now you can see that the tests passed!

You are now ready to find out how to use doctest in your code!

Using doctest in Your Code

You can run the doctest module directly in your code in addition to in the terminal. Let's copy the code from the previous example into a new file named `add_test_in_code.py` and add 3 new lines of code to the end:

```
1 # add_test_in_code.py  
2  
3 def add(a: int, b: int) -> int:  
4     """  
5         >>> add(1, 2)  
6         3  
7         >>> add(4, 5)  
8         9  
9         """  
10    return a + b  
11  
12 if __name__ == '__main__':  
13     import doctest  
14     doctest.testmod(verbose=True)
```

Those last 3 lines of code will only run if you run `add_test_in_code.py` itself, not if you import it. Inside of the `if` statement, you import `doctest` and call its `testmod()` function with `verbose` set to `True`.

Now you can run your code using `python3 add_test_in_code.py`, which will result in the following output:

```
1 Trying:  
2     add(1, 2)  
3 Expecting:  
4     3  
5 ok  
6 Trying:  
7     add(4, 5)  
8 Expecting:  
9     9  
10 ok  
11 1 items had no tests:  
12     __main__  
13 1 items passed all tests:  
14     2 tests in __main__.add  
15 2 tests in 2 items.  
16 2 passed and 0 failed.  
17 Test passed.
```

If you want to thoroughly test your code, you may find that putting all the tests in docstrings makes the code harder to read. To avoid this problem but keep testing your code, you can move your tests to another file. Let's find out how!

Using doctest From a Separate File

When you notice that your doctest docstrings are getting too verbose, it's a good idea to move them into a separate file. Let's create a new file named `doctest_external.py` and simplify the code into the following:

```
1 # doctest_external.py  
2  
3 def add(a: int, b: int) -> int:  
4     return a + b
```

Now, let's take the tests that you put in the docstring and save them in a file named `test.txt`:

```
1 The following are tests for doctest_external  
2  
3 >>> from doctest_external import add  
4 >>> add(1, 2)  
5 3  
6 >>> add(4, 5)  
7 9
```

You can add some helpful information to your text file that explains what the test file is for. The helpful information are the lines that appear before the `>>>`. This is what allowed you to remove the docstring from your `add()` function.

To run `doctest` against this code, you will execute `doctest` using the test file instead of the Python file. Here's an example:

```
1 Computer:chapter30_testing$ python3 -m doctest -v test.txt  
2 Trying:  
3     from doctest_external import add  
4 Expecting nothing  
5 ok  
6 Trying:  
7     add(1, 2)  
8 Expecting:  
9     3  
10 ok  
11 Trying:  
12     add(4, 5)  
13 Expecting:  
14     9  
15 ok  
16 1 items passed all tests:  
17     3 tests in test.txt  
18 3 tests in 1 items.  
19 3 passed and 0 failed.  
20 Test passed.
```

This code shows 3 tests passing instead of 2 because you need to import `doctest_external` before you can really test it. `doctest` will treat the import as a test too!

The `doctest` module can be used directly to execute the text file via the `testfile()` function. Here is an example that demonstrates how you could do that:

```
1 >>> import doctest
2 >>> doctest.testfile('test.txt')
3 TestResults(failed=0, attempted=3)
4 >>>
```

When you run `doctest` in this manner, it will return a `TestResults` object that shows you the number of attempted tests and the number of failures. You can see that it attempted to do 3 tests and none of them failed, which indicates that they passed successfully.

Apart from `doctest`, there are other ways that you can create tests in Python. You will learn about using a different module called `unittest` in the next section.

Using `unittest` For Test Driven Development

Python has the `unittest` module that allows you to write unit tests for your code. A unit test is a way to test the smallest piece of code that can be isolated from the rest. Considering the `add()` function from the previous section, you might write a test that adds two positive numbers, or a test that adds a negative and a positive number. However, you wouldn't test both of those scenarios in the same unit test.

A popular concept in programming is something called **Test-Driven Development** or TDD. The idea behind TDD is that you will write a failing test BEFORE you write any code. The basic concept is that you will spend time thinking about the implementation of your program and how you would test it before you start coding.

Here's an overview of the TDD process:

- **Fail:** First, you write a simple test and run it. It should fail right away since you haven't written any functional code yet.
- **Pass:** The next step is to write some code that passes the test.

Then you start the process from the top by writing a new test that should also fail initially. And so you iterate writing tests and writing code to pass the tests.

You can practice TDD by doing a **code kata**. Code katas are coding challenges that are designed to practice a specific programming concept in depth.

One of the popular programming katas is called **FizzBuzz**. This is also a popular interview question for computer programmers.

The concept behind FizzBuzz is as follows:

- Write a program that prints the numbers 1-100, each on a new line
- For each number that is a multiple of 3, print "Fizz" instead of the number

- For each number that is a multiple of 5, print “Buzz” instead of the number
- For each number that is a multiple of both 3 and 5, print “FizzBuzz” instead of the number

Let’s take this popular challenge and write a solution *and* some tests for it using Test-Driven Development.

The Fizz Test

The first step is to create a folder to hold your test. You can create a `fizzbuzz` folder for this purpose. Then add your code to that folder.

A lot of people will save their tests in sub-folder called `test` or `tests` and tell their test runner to add the top level folder to `sys.path` so that the tests can import it. A test runner is a separate program that you could use to run your tests continuously for you. You can skip that in this case.

Instead, let’s go ahead and create a test file called `test_fizzbuzz.py` inside your `fizzbuzz` folder.

Now enter the following into your Python file:

```
1 import fizzbuzz
2 import unittest
3
4 class TestFizzBuzz(unittest.TestCase):
5
6     def test_multiple_of_three(self):
7         self.assertEqual(fizzbuzz.process(6), 'Fizz')
8
9 if __name__ == '__main__':
10    unittest.main()
```

Here you import the `fizzbuzz` module, which is the code that you will be testing. Then you import Python’s `unittest` module. To use `unittest`, you will need to subclass `unittest.TestCase()`. Then you can create a series of functions that represent the tests that you want to run. In this case, you create `test_multiple_of_three()` and use `assertEqual()` to test that `fizzbuzz.process(6)` will return the string, ‘Fizz’.

If you run this test right now, you will receive a `ModuleNotFoundError` because `fizzbuzz` doesn’t exist. The fix is to create an empty `fizzbuzz.py` file in the same folder as your test file. This will only fix the `ModuleNotFoundError`, but it will allow you to run the test and see its output.

You can run the test like this:

```
1 python test_fizzbuzz.py
```

When you run this code, you will see the following output:

```
1 E
2 =====
3 ERROR: test_multiple_of_three (__main__.TestFizzBuzz)
4 -----
5 Traceback (most recent call last):
6   File "test_fizzbuzz.py", line 7, in test_multiple_of_three
7     self.assertEqual(fizzbuzz.process(6), 'Fizz')
8 AttributeError: module 'fizzbuzz' has no attribute 'process'
9
10 -----
11 Ran 1 test in 0.000s
12
13 FAILED (errors=1)
```

This tells you that your `fizzbuzz` module is missing an attribute called `process()`.

You can attempt to fix that by adding a `process()` function to your `fizzbuzz.py` file:

```
1 def process(number):
2     pass
```

This function accepts a number, but doesn't do anything else. That should solve the `AttributeError` you received. Try re-running the test and you should get the following output:

```
1 F
2 =====
3 FAIL: test_multiple_of_three (__main__.TestFizzBuzz)
4 -----
5 Traceback (most recent call last):
6   File "test_fizzbuzz.py", line 7, in test_multiple_of_three
7     self.assertEqual(fizzbuzz.process(6), 'Fizz')
8 AssertionError: None != 'Fizz'
9
10 -----
11 Ran 1 test in 0.000s
12
13 FAILED (failures=1)
```

Oops! Python returns `None` by default. That's not what this function should return though. Let's update the code again:

```
1 def process(number):
2     if number % 3 == 0:
3         return 'Fizz'
```

This time the function takes the `number` and uses the modulus operator to divide the number by 3 and check to see if there is a remainder. If there is no remainder, then you know that the number is divisible by 3 so you can return the string "Fizz".

Now when you run the test, the output should look like this:

```
1 .
2 -----
3 Ran 1 test in 0.000s
4
5 OK
```

The period on the first line above means that you ran one test and it passed.

Let's take a step back here and discuss what you just witnessed from a TDD perspective. When a test is failing, it is considered to be in a "red" state. When a test is passing, that is a "green" state. This refers to the Test Driven Development (TDD) mantra of red/green/refactor. Most developers will start a new project by creating a failing test (red). Then they will write the code to make the test pass, usually in the simplest way possible (green).

If you were using source control, you would commit your code now because it is passing. This allows you to rollback your code to a functional state should your next change break something. You will learn more about source control later on in this book.

Now you are ready to write another test!

The Buzz Test

The second test for your Fizzbuzz code will check for the correct response to multiples of five. To add a new test, you can create another method in the `TestFizzBuzz()` class:

```
1 import fizzbuzz
2 import unittest
3
4 class TestFizzBuzz(unittest.TestCase):
5
6     def test_multiple_of_three(self):
7         self.assertEqual(fizzbuzz.process(6), 'Fizz')
8
9     def test_multiple_of_five(self):
```

```
10     self.assertEqual(fizzbuzz.process(20), 'Buzz')
11
12 if __name__ == '__main__':
13     unittest.main()
```

This time around, your test will need to use a number that is only divisible by 5, but not by 3. In a working state, `fizzbuzz.process()` should return “Buzz”. When you run the test now, though, you will receive this:

```
1 F.
2 =====
3 FAIL: test_multiple_of_five (__main__.TestFizzBuzz)
4 -----
5 Traceback (most recent call last):
6   File "test_fizzbuzz.py", line 10, in test_multiple_of_five
7     self.assertEqual(fizzbuzz.process(20), 'Buzz')
8 AssertionError: None != 'Buzz'
9
10 -----
11 Ran 2 tests in 0.000s
12
13 FAILED (failures=1)
```

Oops! Right now your code uses the modulus operator to check for remainders after dividing by 3. If the number 20 has a remainder, that statement won’t run. The default return value of a function is `None`, so that is why you end up receiving the above error message.

Go ahead and update the `process()` function in your `fizzbuzz.py` file so that it looks like this:

```
1 def process(number):
2     if number % 3 == 0:
3         return 'Fizz'
4     elif number % 5 == 0:
5         return 'Buzz'
```

Now you can check for remainders with both 3 and 5. When you run the tests this time, the output should look like this:

```
1 ..
2 -----
3 Ran 2 tests in 0.000s
4
5 OK
```

Yay! Your tests passed and are now green! That means you can commit these changes to your Git repository.

Now you are ready to add a test for “FizzBuzz”!

The FizzBuzz Test

The next test that you can write will be for when you want to get “FizzBuzz” back. As you may recall, you will get FizzBuzz whenever the number is divisible by both 3 and 5. Go ahead and add a third test that does just that:

```
1 import fizzbuzz
2 import unittest
3
4 class TestFizzBuzz(unittest.TestCase):
5
6     def test_multiple_of_three(self):
7         self.assertEqual(fizzbuzz.process(6), 'Fizz')
8
9     def test_multiple_of_five(self):
10        self.assertEqual(fizzbuzz.process(20), 'Buzz')
11
12    def test_fizzbuzz(self):
13        self.assertEqual(fizzbuzz.process(15), 'FizzBuzz')
14
15 if __name__ == '__main__':
16     unittest.main()
```

For this test, `test_fizzbuzz()`, you ask your program to process the number 15. This shouldn’t work right yet, but go ahead and run the test code to check:

```
1 F..
2 =====
3 FAIL: test_fizzbuzz (__main__.TestFizzBuzz)
4 -----
5 Traceback (most recent call last):
6   File "test_fizzbuzz.py", line 13, in test_fizzbuzz
7     self.assertEqual(fizzbuzz.process(15), 'FizzBuzz')
8 AssertionError: 'Fizz' != 'FizzBuzz'
9 - Fizz
10 + FizzBuzz
11
12 -----
13 -----
14 Ran 3 tests in 0.001s
15
16 FAILED (failures=1)
```

Three tests were run with one failure. You are now back to red. This time the error is 'Fizz' != 'FizzBuzz' instead of comparing None to FizzBuzz. The reason for that is because your code checks if 15 is divisible by 3 and because it actually is, your code returns "Fizz".

Since that isn't what you want to happen, you will need to update your code to check if the number is divisible by both 3 and 5 before checking for just 3:

```
1 def process(number):
2     if number % 3 == 0 and number % 5 == 0:
3         return 'FizzBuzz'
4     elif number % 3 == 0:
5         return 'Fizz'
6     elif number % 5 == 0:
7         return 'Buzz'
```

Here you do the divisibility check for 3 and 5 first. Then you check for the other two as before.

Now if you run your tests, you should get the following output:

```
1 ...
2 -----
3 Ran 3 tests in 0.000s
4
5 OK
```

So far so good. However, you don't have the code working for returning numbers that aren't divisible by 3 or 5. Time for another test!

The Final Test

The last thing that your code needs to do is return the number when it does have a remainder when divided by 3 and 5. Let's test it in a couple of different ways:

```
1 import fizzbuzz
2 import unittest
3
4 class TestFizzBuzz(unittest.TestCase):
5
6     def test_multiple_of_three(self):
7         self.assertEqual(fizzbuzz.process(6), 'Fizz')
8
9     def test_multiple_of_five(self):
10        self.assertEqual(fizzbuzz.process(20), 'Buzz')
11
12    def test_fizzbuzz(self):
13        self.assertEqual(fizzbuzz.process(15), 'FizzBuzz')
14
15    def test_regular_numbers(self):
16        self.assertEqual(fizzbuzz.process(2), 2)
17        self.assertEqual(fizzbuzz.process(98), 98)
18
19 if __name__ == '__main__':
20     unittest.main()
```

For this test, you test normal numbers 2 and 98 with the `test_regular_numbers()` test. These numbers will always have a remainder when divided by 3 or 5, so they should just be returned as the original numbers.

When you run the tests now, you should get something like this:

```
1 ...F
2 =====
3 FAIL: test_regular_numbers (__main__.TestFizzBuzz)
4 -----
5 Traceback (most recent call last):
6   File "test_fizzbuzz.py", line 16, in test_regular_numbers
7     self.assertEqual(fizzbuzz.process(2), 2)
8 AssertionError: None != 2
9
10 -----
11 Ran 4 tests in 0.001s
```

```
12  
13 FAILED (failures=1)
```

This time you are back to comparing `None` to the number, which is what you probably suspected would be the output.

Go ahead and update the `process()` function in your `fizzbuzz` module as follows:

```
1 def process(number):  
2     if number % 3 == 0 and number % 5 == 0:  
3         return 'FizzBuzz'  
4     elif number % 3 == 0:  
5         return 'Fizz'  
6     elif number % 5 == 0:  
7         return 'Buzz'  
8     else:  
9         return number
```

That was easy! All you needed to do at this point was add an `else` statement that returns the `number`.

When you run the tests, they should all pass:

```
1 ....  
2 -----  
3 Ran 4 tests in 0.000s  
4  
5 OK
```

Good job! Your code works. You can verify that it works for all the numbers, 1-100, by adding the following to your `fizzbuzz.py` module:

```
1 if __name__ == '__main__':  
2     for i in range(1, 101):  
3         print(process(i))
```

When you run `fizzbuzz` yourself using `python fizzbuzz.py`, you should see the appropriate output that was specified at the beginning of this section.

Wrapping Up

You have only scratched the surface when it comes to testing in Python. In fact, there are entire books written on the topic and many tutorials and videos as well. In this chapter, you learned about:

- Using doctest in the Terminal
- Using doctest in Your Code
- Using doctest From a Separate File
- Using unittest For Test Driven Development

The doctest module is quite neat, but not used all that often. Of the two libraries, unittest is the one that is used the most as it offers greater flexibility. There are also third party testing packages. The most popular one is called pytest. While it's not covered in this book, you should check it out if you are interested in doing testing in Python.

You should always test your code. It helps you to be confident that your code is working the way it should. Testing will save you many headaches later on when you update your code.

Review Questions

1. How do you add tests to be used with doctest?
2. Are tests for doctest required to be in the docstring? If not, how would you execute it?
3. What is a unit test?
4. What is test driven development?

Chapter 31 - Learning About the Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain code, equations, visualizations, and formatted text. By default, Jupyter Notebook runs Python out of the box. Additionally, Jupyter Notebook supports many other programming languages via extensions. You can use the Jupyter Notebook for data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more!

In this chapter, you will learn about the following:

- Installing The Jupyter Notebook
- Creating a Notebook
- Adding Content
- Adding Markdown Content
- Adding an Extension
- Exporting Notebooks to Other Formats

This chapter is not meant to be a comprehensive tutorial on the Jupyter Notebook. Instead it will show you the basics of how to use a Notebook and why it might be useful. If you are intrigued by this technology, you might want to check out my book on the topic, **Jupyter Notebook 101**.

Let's get started!

Installing The Jupyter Notebook

Jupyter Notebook does not come with Python. You will need to install it using `pip`. If you are using **Anaconda** instead of the official Python, then Jupyter Notebook comes with Anaconda, pre-installed.

Here is how you would install Jupyter Notebook with `pip`:

```
1 python3 -m pip install jupyter
```

When you install Jupyter Notebook, it will install a lot of other dependencies. You may want to install Jupyter Notebook into a Python virtual environment. See **Chapter 21** for more information.

Once the installation is done, you are ready to create a Jupyter Notebook!

Creating a Notebook

Creating a Notebook is a fundamental concept. Jupyter Notebook operates through its own server, which comes included with your installation. To be able to do anything with Jupyter, you must first launch this **Jupyter Notebook Server** by running the following command:

```
1 jupyter notebook
```

This command will either launch your default browser or open up a new tab, depending on whether your browser is already running or not. In both cases you will soon see a new tab that points to the following URL: <http://localhost:8888/tree>. Your browser should load up to a page that looks like this:

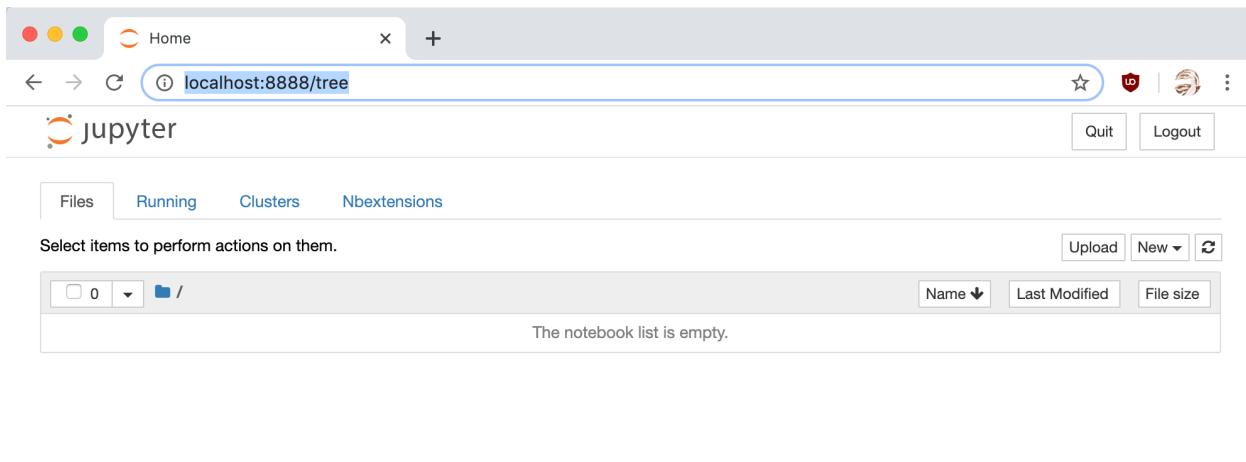


Fig. 31-1: Jupyter Notebook Server

Here you can create a Notebook by clicking the New button on the right:

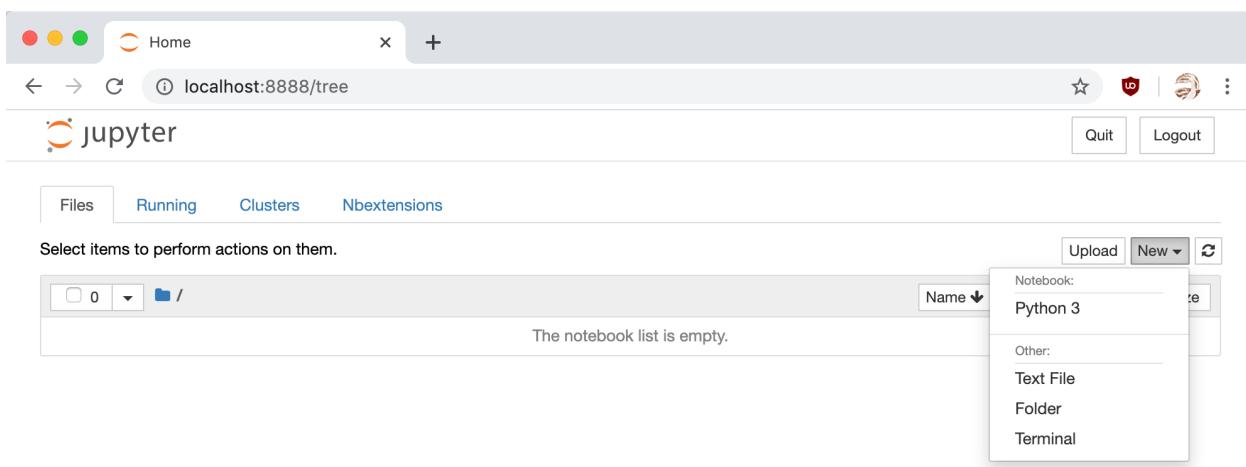


Fig. 31-2: Creating a Jupyter Notebook

You can create a Notebook with this menu as well as a text file, a folder, and an in-browser terminal session. For now, you should choose the **Python 3** option.

Having done that, a new tab will open with your new Notebook loaded:

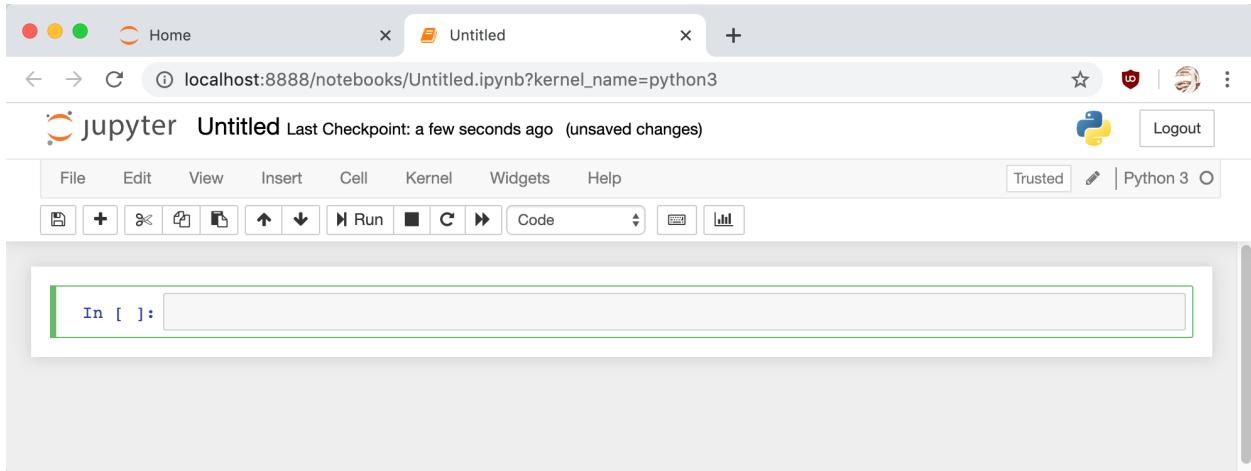


Fig. 31-3: A New Jupyter Notebook

Now let's learn about how to interact with the Notebook!

Naming Your Notebook

The top of the Notebook says that it is *Untitled*. To fix that, all you need to do is click on the word *Untitled* and an in-browser dialog will appear:

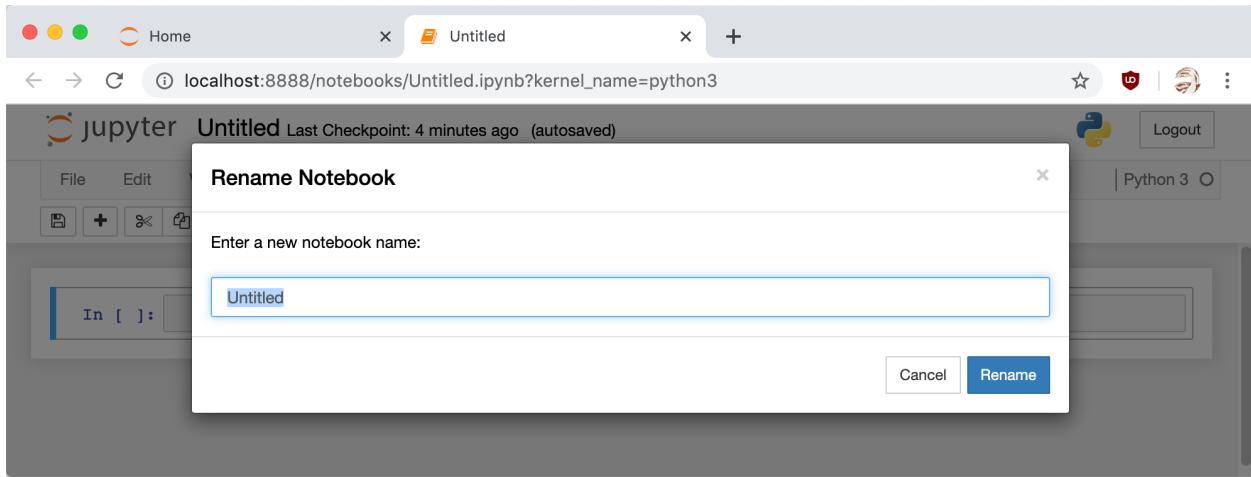


Fig. 31-4: Renaming the Notebook

When you rename the Notebook, it will also rename the file that the Notebook is saved to so that it matches the name you gave it. You can name this Notebook "Hello World".

Running Cells

Jupyter Notebook cells are where the magic happens. This is where you can create content and interactive code. By default, the Notebook will create cells in code mode. That means that it will allow you to write code in whichever kernel you chose when you created the Notebook. A kernel refers to the programming language that you chose when creating your Jupyter Notebook. You chose Python 3 when you created this Notebook, so you can write Python 3 code in the cell.

Right now the cell is empty, so it doesn't do anything at all. Let's add some code to change that:

```
1 print('Hello from Python!')
```

To execute the contents of a cell, you need to **run** that cell. After selecting the cell, there are three ways of running it:

- Clicking the **Run** button in the row of buttons along the top
- Navigating to **Cell -> Run Cells** from the Notebook menu
- Using the keyboard shortcut: **Shift+Enter**

When you run this cell, the output should look like this:

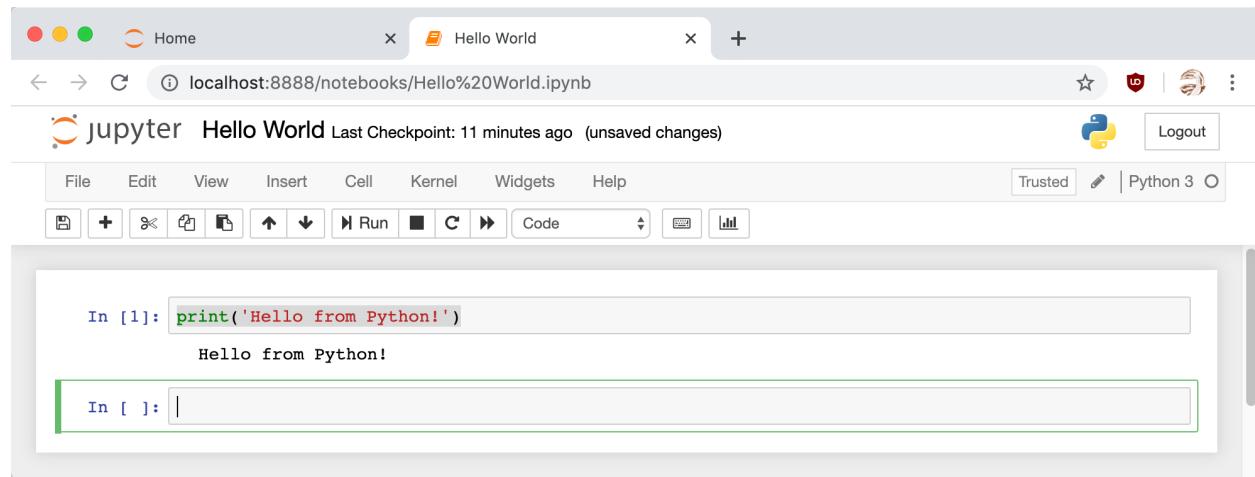


Fig. 31-5: Running a Cell

Jupyter Notebook cells remember the order in which they are run. If you run the cells out of order, you may end up with errors because you haven't imported something in the right order. However, when you **do** run the cells in order, you can write imports in one cell and still use those imports in later cells. Notebooks make it simple to keep logical pieces of the code together. In fact, you can put explanatory cells, graphs and more between the code cells and the code cells will still share with each other.

When you run a cell, there are some brackets next to the cell that will fill-in with a number. This indicates the order in which the cells were run. In this example, when you ran the first cell, the

brackets filled in with the number one. Because all code cells in a notebook operate on the same global namespace, it is important to be able to keep track of the order of execution of your code cells.

Learning About the Menus

There is a menu in the Jupyter Notebook that you can use to work with your Notebook. The menu runs along the top of the Notebook. Here are your menu options:

- File
- Edit
- View
- Insert
- Cell
- Kernel
- Widgets
- Help

Let's go over each of these menus. You don't need to know about every single option in these menus to start working with Jupyter, so this will be a high-level overview.

The **File** menu is used for opening a Notebook or creating a new one. You can also rename the Notebook here. One of the nice features of Notebooks is that you can create **Checkpoints**. Checkpoints allow you to rollback to a previous state. To create a Checkpoint, go in the *File* menu and choose the *Save and Checkpoint* option.

The **Edit** menu contains your regular cut, copy, and paste commands, which you can use on a cell level. You can also delete, split, or merge cells from here. Finally, you can use this menu to reorder the cells.

You will find that some of the options here are grayed out. The reason an item is grayed out is because that option does not apply to the currently selected cell in your Notebook. For example, if you selected a code cell, you won't be able insert an image. Try changing the cell type to Markdown to see how the options change.

The **View** menu is used for toggling the visibility of the header and the toolbar. This is also where you would go to toggle *Line Numbers* on or off.

The **Insert** menu is used for inserting cells above or below the currently selected cell.

The **Cell** menu is useful for running one cell, a group of cells or everything in the Notebook! You can change the cell type here, but you will probably find that the toolbar is more intuitive to use than the menu for that sort of thing.

Another useful feature of the **Cell** menu is that you can use it to clear the cell's output. A lot of people share their Notebooks with others. If you want to do that, it can be useful to clear out the

outputs of the cells so that your friends or colleagues can run the cells themselves and discover how they work.

The *Kernel* menu is for working with the Kernel itself. The Kernel refers to the programming language plugin. You will occasionally need to restart, reconnect or shut down your kernel. You can also change which kernel is running in your Notebook.

You won't use the Kernel menu all that often. However, when you need to do some debugging in Jupyter Notebook, it can be handy to restart the Kernel rather than restarting the entire server.

The *Widgets* menu is for clearing and saving widget state. A Widget is a way to add dynamic content to your Notebook, like a button or slider. These are written in JavaScript under the covers.

The last menu is the *Help* menu. This is where you will go to learn about the special keyboard shortcuts for your Notebook. It also provides a user interface tour and plenty of reference material that you can use to learn how to better interact with your Notebook.

Now let's learn how to create content in your Notebook!

Adding Content

You can choose between two primary types of content for your Notebooks:

- Code
- Markdown

There are technically two other cell types you can choose. One is **Raw NBConvert**, which is only intended for special use cases when using the nbconvert command line tool. This tool is used to convert your Notebook to other formats, such as PDF.

The other type is **Heading**, which actually isn't used anymore. If you choose this cell type, you will receive the following dialog:

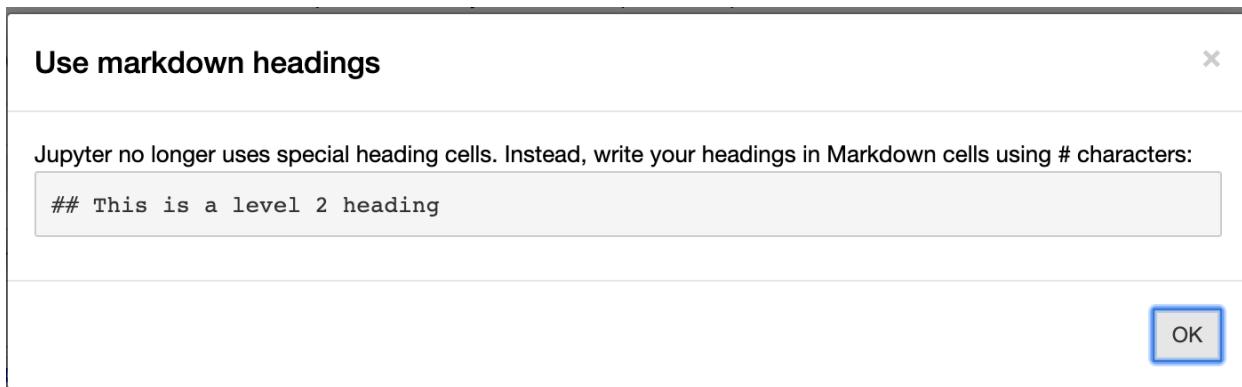


Fig. 31-6: Heading Cell Type

You have already seen how to use the default cell type, **Code**. So the next section will focus on **Markdown**.

Creating Markdown Content

The Markdown cell type allows you to format your text. You can create headings, add images and links, and format your text with italics, bold, etc.

This chapter won't cover everything you can do with Markdown, but it will teach you the basics. Let's take a look at how to do a few different things!

Formatting Your Text

If you would like to add italics to your text, you can use single underscores or single asterisks. If you would rather bold your text, then you double the number of asterisks or underscores.

Here are a couple of examples:

- 1 You can italicize like `*this*` or `_this_`
- 2
- 3 Or bold like `**this**` or `__this__`

Try setting your Notebook cell to Markdown and adding the text above to it. You will then see that the Notebook is automatically formatting the text for you:

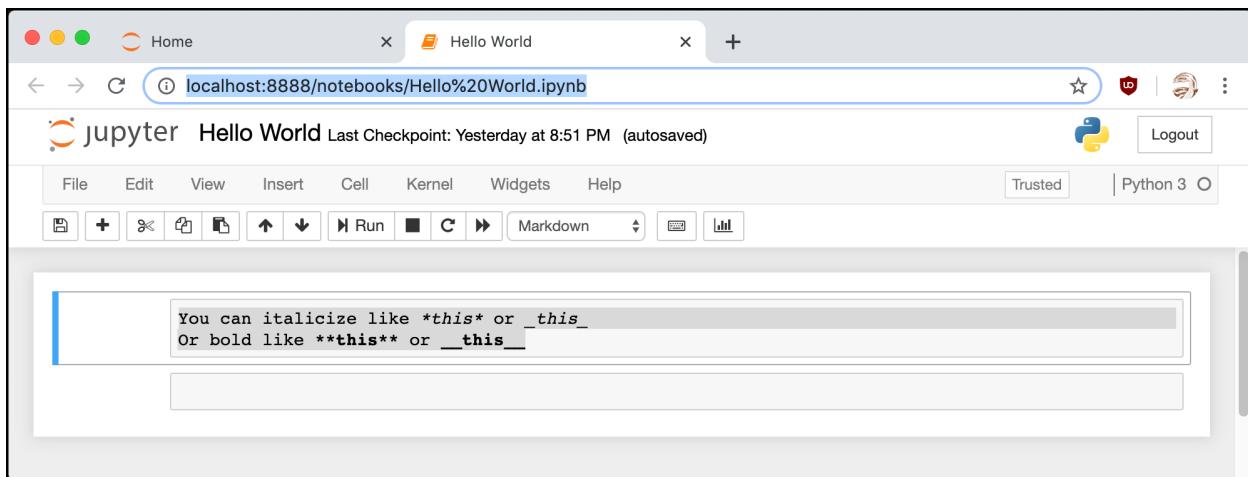


Fig. 31-7: Formatting Text with Markdown

When you run the cell, it will format the text nicely:

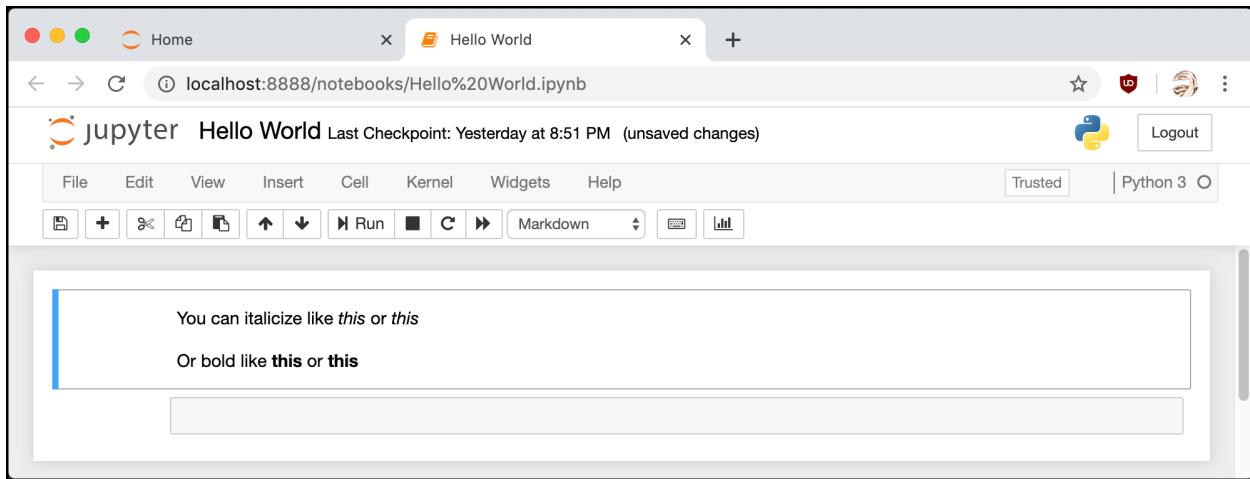


Fig. 31-8: Markdown Cell When Ran

If you need to edit the cell again, you can double-click the cell and it will go back into editing mode. Now let's find out how to add heading levels!

Using Headings

Headings are good for creating sections in your Notebook, just like they are when you are creating a web page or a document in Microsoft Word. To create headings in Markdown, you can use one or more # signs.

Here are some examples:

```
1 # Heading 1
2 ## Heading 2
3 ### Heading 3
4 ##### Heading 4
```

If you add the code above to a Markdown cell in your Notebook, it will look like this:

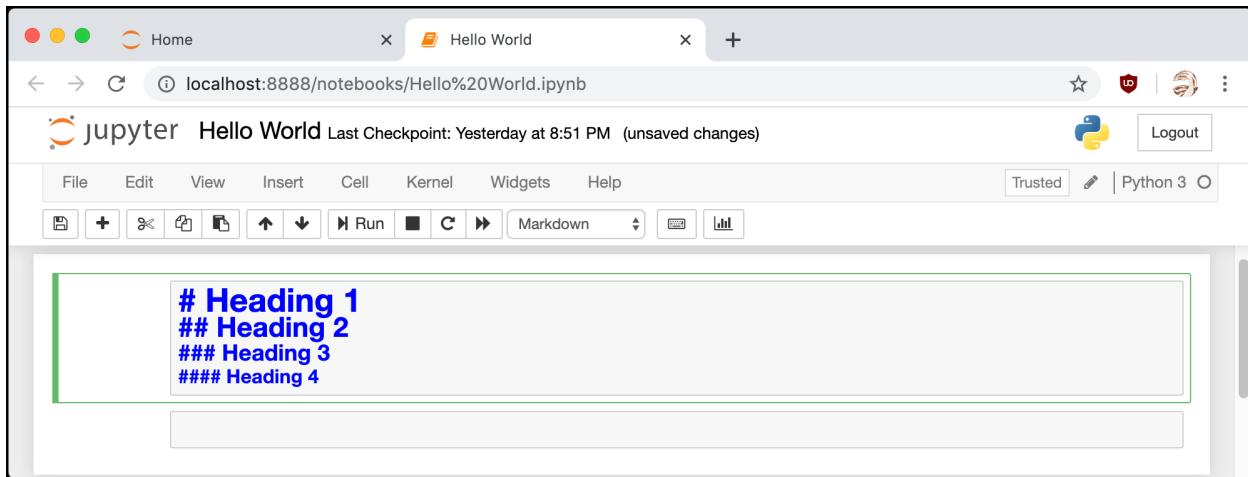


Fig. 31-9: Markdown Headings

You can see that the Notebook is already generating a type of preview for you here by shrinking the text slightly for each heading level.

When you run the cell, you will see something like the following:

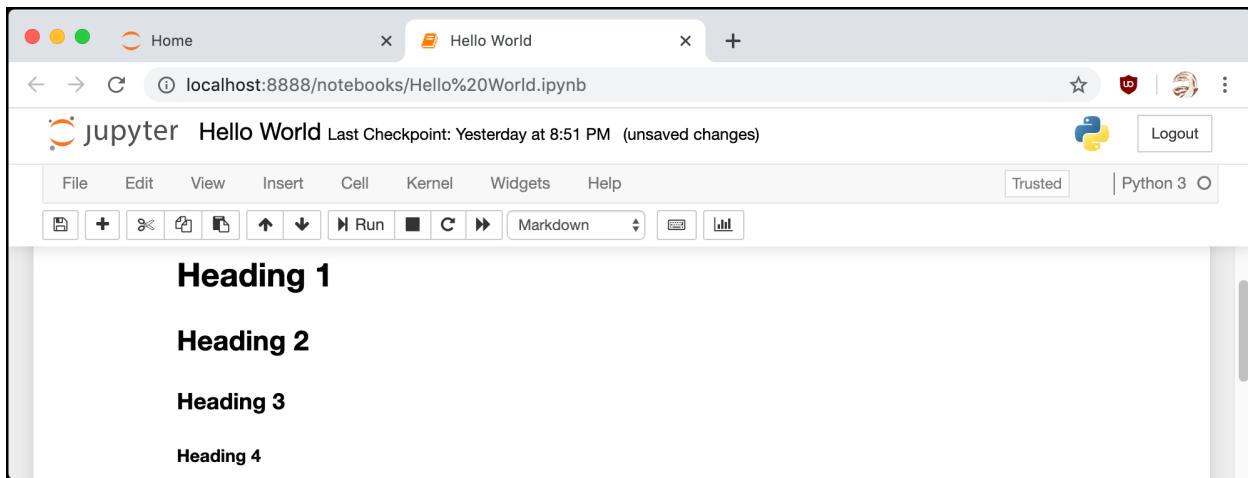


Fig. 31-10: Markdown Headings When Ran

As you can see, Jupyter nicely formats your text as different-level headings that can be helpful to structure your text.

Adding a Listing

Creating a listing or bullet points is pretty straight-forward in Markdown. To create a listing, you add an asterisk (*) or a dash (-) to the beginning of the line.

Here is an example:

```
1 * List item 1
2   * sub item 1
3   * sub item 2
4 * List item 2
5 * List item 3
```

Let's add this code to your Notebook:

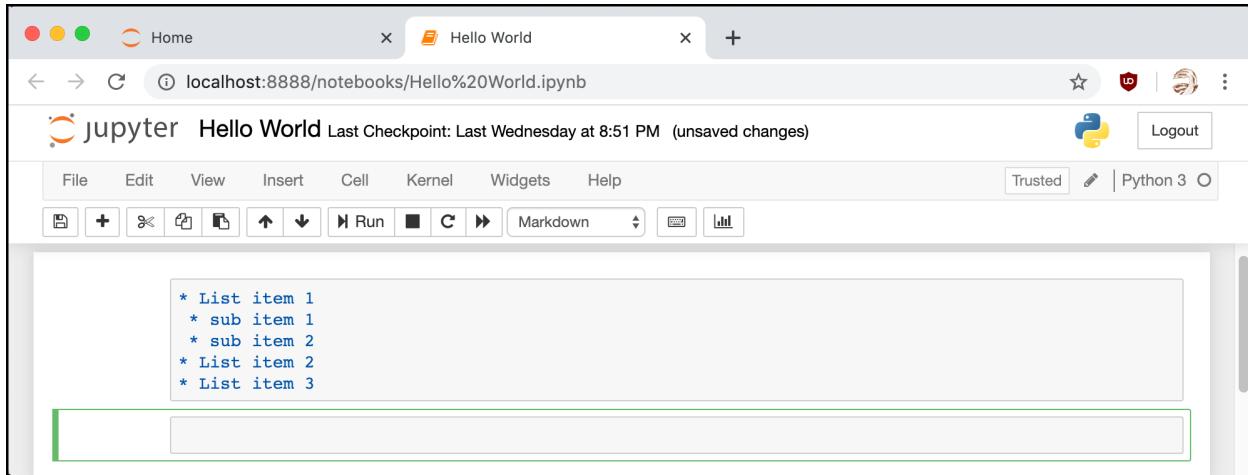


Fig. 31-11: Markdown Listings

You don't really get a preview of listings this time, so let's run the cell to see what you get:

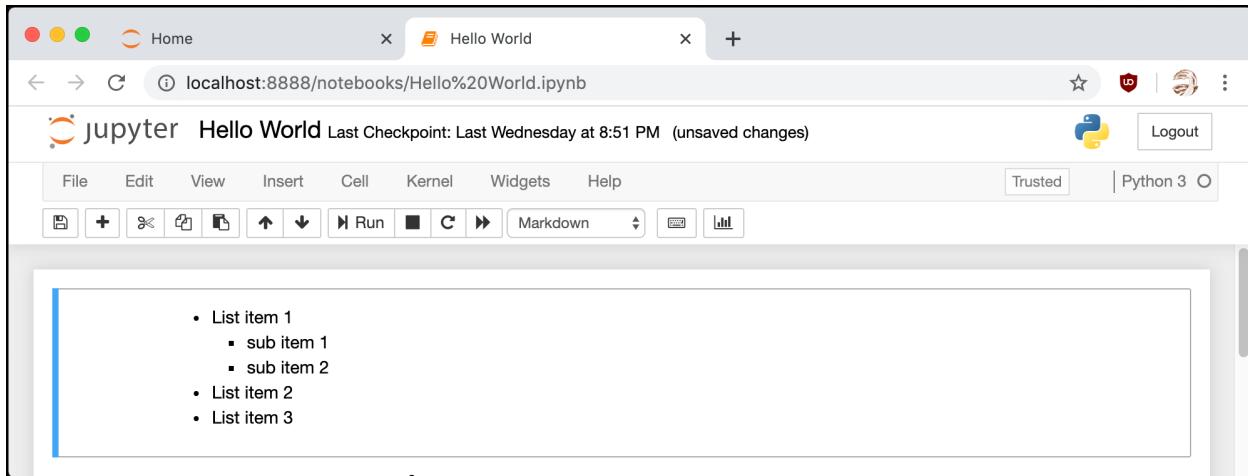


Fig. 31-12: Markdown Listings When Run

That looks pretty good! Now let's find out how to get syntax highlighting for your code!

Highlighting Code Syntax

Notebooks already allow you to show and run code and they even show syntax highlighting. However, this only works for the Kernels or languages installed in Jupyter Notebook.

If you want to show code for another language that is not installed or if you want to show syntax highlighting without giving the user the ability to run the code, then you can use Markdown for that.

To create a code block in Markdown, you would need to use 3 backticks followed by the language that you want to show. If you want to do inline code highlighting, then surround the code snippet with single backticks. However, keep in mind that inline code doesn't support syntax highlighting.

Here are two examples in the Notebook:

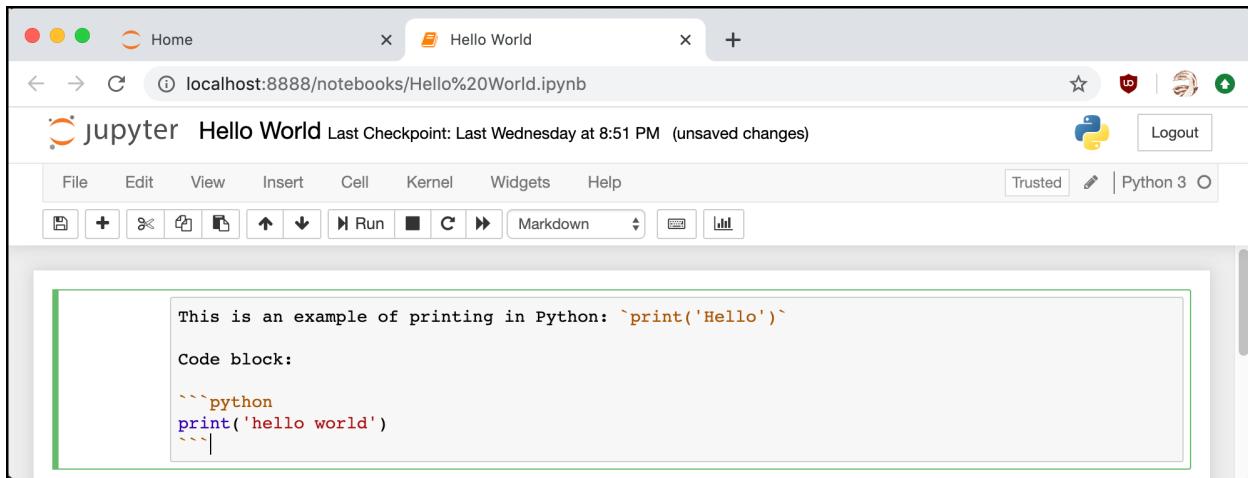


Fig. 31-13: Syntax Highlighting Examples

When you run the cell, the Notebook transforms the Markdown into the following:

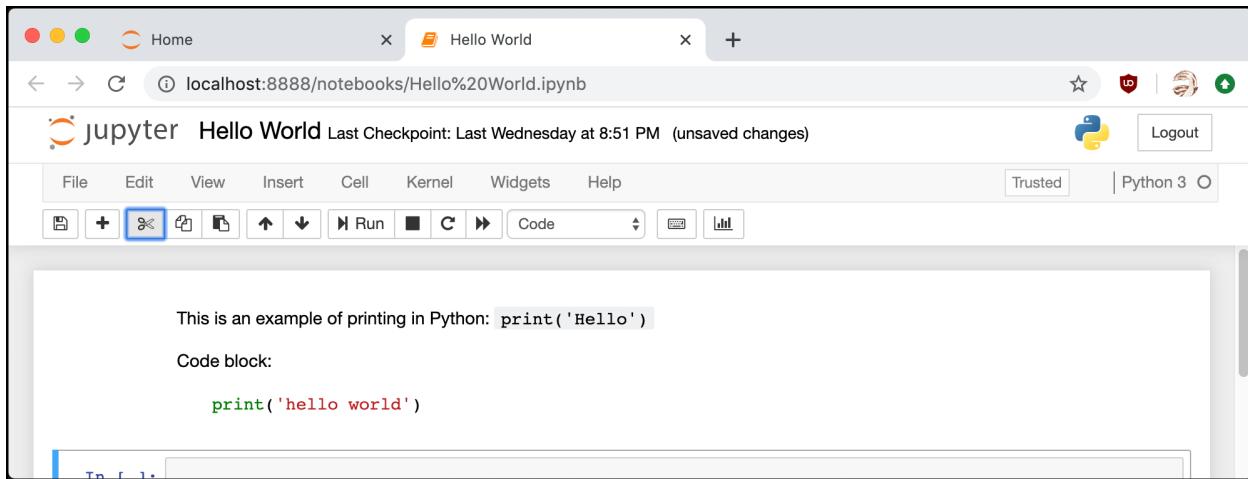


Fig. 31-14: Syntax Highlighting Examples Ran

Here you can see how the code now has syntax highlighting.

Now let's learn how to generate a hyperlink!

Creating a Hyperlink

Creating hyperlinks in Markdown is quite easy. The syntax is as follows:

```
1 [text](URL)
```

So if you wanted to link to Google, you would do this:

```
1 [Google](https://www.google.com)
```

Here is what the code looks like in the Notebook:

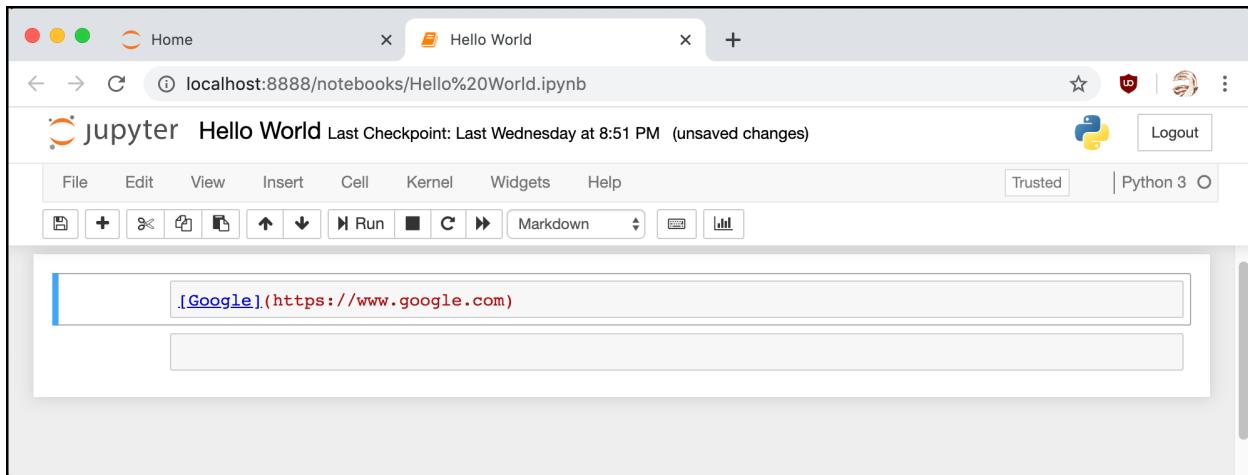


Fig. 31-15: Hyperlink Syntax

When you run the cell, you will see the Markdown turned into a regular hyperlink:

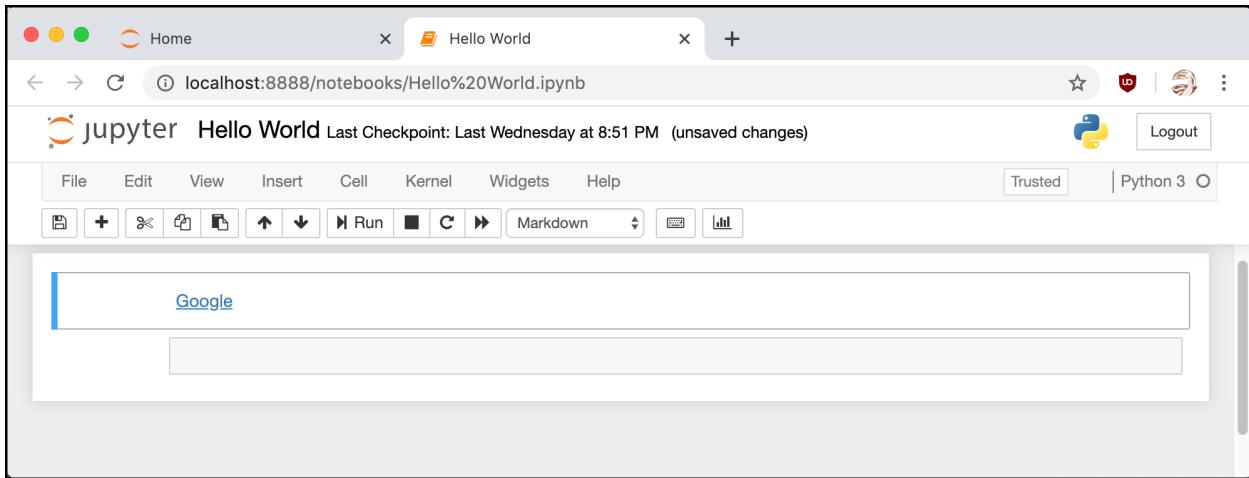


Fig. 31-16: Hyperlink Syntax When Run

As you can see, the Markdown has been transformed into a traditional hyperlink.

Let's find out about Jupyter extensions next!

Adding an Extension

Jupyter Notebook has lots of functionality right out of the box. If you need anything beyond that, you can also add new features through extensions from a large extension ecosystem. There are four different types of extensions available:

- Kernel
- IPython kernel
- Notebook
- Notebook server

Most of the time, you will want to install a Notebook extension.

An extension for Jupyter Notebook is technically a JavaScript module that will be loaded in the Notebook's front-end to add new functionality or make the Notebook look different. If you know JavaScript, you can write your own extension!

If you need to add something new to Jupyter Notebook, you should use Google to see if someone has written something that will work for you. The most popular extension is actually a large set of extensions called `jupyter_contrib_nbextensions` which you can get here:

- https://github.com/ipython-contrib/jupyter_contrib_nbextensions

Most good extensions can be installed using `pip`. For example, to install the one mentioned above, you can run this command:

```
1 $ pip install jupyter_contrib_nbextensions
```

There are a few that are not compatible with pip. In those cases, you can use Jupyter itself to install the extension:

```
1 $ jupyter nbextension install NAME_OF_EXTENSION
```

While this installs the extension for Jupyter to use, it does not make the extension active yet. You will need to enable an extension if you install it using this method before you can use it.

To enable an extension, you need to run the following command:

```
1 $ jupyter nbextension enable NAME_OF_EXTENSION
```

If you installed the extension while you were running Jupyter Notebook, you may need to restart the Kernel or the entire server to be able to use the new extension.

You may want to get the **Jupyter NbExtensions Configurator** extension to help you manage your extensions. It is a neat extension designed for enabling and disabling other extensions from within your Notebook's user interface. It also displays the extensions that you have currently installed.

Exporting Notebooks to Other Formats

After you have created an amazing Notebook, you may want to share it with other people who are not as computer savvy as you are. Jupyter Notebook supports converting the Notebooks to other formats:

- HTML
- LaTeX
- PDF
- RevealJS
- Markdown
- ReStructured Text
- Executable script

You can convert a Notebook using the nbconvert tool that was installed when you originally installed Jupyter Notebook. To use nbconvert, you can do the following:

```
1 $ jupyter nbconvert <notebook file> --to <output format>
```

Let's say you want to convert your Notebook to PDF. To do that, you would do this:

```
1 $ jupyter nbconvert my_notebook.ipynb --to pdf
```

You will see some output as it converts the Notebook into a PDF. The `nbconvert` tool will also display any warnings or errors that it encounters during the conversion. If the process finishes successfully, you will have a `my_notebook.pdf` file in the same folder as the Notebook file.

The Jupyter Notebook provides a simpler way to convert your Notebooks too. You can do so from the *File* menu within the Notebook itself. You can choose the `Download as` option to do the conversion.

Depending on the platform that you are on, you may need to install LaTeX or other dependencies to get certain export formats to work properly.

Wrapping Up

The Jupyter Notebook is a fun way to learn how to use Python or machine learning. It is a great way to organize your data so that you can share it with others. You can use it to create presentations, show your work, and run your code.

In this chapter, you learned about the following:

- Installing The Jupyter Notebook
- Creating a Notebook
- Adding Content
- Adding Markdown Content
- Adding an Extension
- Exporting Notebooks to Other Formats

You should give Jupyter Notebook a try. It's a useful coding environment and well worth your time.

Review Questions

1. What is Jupyter Notebook?
2. Name two Notebook cell types
3. What markup language do you use to format text in Jupyter Notebook?
4. How do you export a Notebook to another format?

Part III - Practical Python

It's always helpful to learn how to create real world applications. In this section of the book, you will learn how to use Python to create small applications or useful snippets that you can use in your own code.

Here is what you will be learning about:

- Chapter 32 - How to Create a Command Line Application with `argparse`
- Chapter 33 - How to Parse XML
- Chapter 34 - How to Parse JSON
- Chapter 35 - How to Scrape a Website
- Chapter 36 - How to Work with CSV files
- Chapter 37 - How to Work with a Database Using `sqlite3`
- Chapter 38 - How to Create an Excel Document
- Chapter 39 - How to Generate a PDF
- Chapter 40 - How to Create Graphs
- Chapter 41 - How to Work with Images in Python
- Chapter 42 - How to Create a GUI

Once you're done reading this section of the book, you will be able to tackle many different problems. This section of the book will give you a taste of the many things you can do with Python.

It's time to get started!

Chapter 32 - How to Create a Command-line Application with argparse

When you are creating an application, you will usually want to be able to tell your application how to do something. There are two popular methods for accomplishing this task. You can make your application accept command-line arguments or you can create a graphical user interface. Some applications support both.

Command-line interfaces are helpful when you need to run your code on a server. Most servers do not have a monitor hooked up, especially if they are Linux servers. In those cases, you might not be able to run a graphical user interface even if you wanted to.

Python comes with a built-in library called `argparse` that you can use to create a command-line interface. In this chapter, you will learn about the following:

- Parsing Arguments
- Creating Helpful Messages
- Adding Aliases
- Using Mutually Exclusive Arguments
- Creating a Simple Search Utility

There is a lot more to the `argparse` module than what will be covered in this chapter. If you would like to know more about it, you can check out the documentation here:

- <https://docs.python.org/3/library/argparse.html>

Now it's time to get started with parsing arguments from the command-line!

Parsing Arguments

Before you learn how to use `argparse`, it's good to know that there is another way to pass arguments to a Python script. You can pass any arguments to a Python script and access those arguments by using the `sys` module.

To see how that works, create a file named `sys_args.py` and enter the following code into it:

```
1 # sys_args.py
2
3 import sys
4
5 def main():
6     print('You passed the following arguments:')
7     print(sys.argv)
8
9 if __name__ == '__main__':
10    main()
```

This code imports `sys` and prints out whatever is in `sys.argv`. The `argv` attribute contains a list of everything that was passed to the script with the first item being the script itself.

Here's an example of what happens when you run this code along with a couple of sample arguments:

```
1 $ python3 sys_args.py --s 45
2 You passed the following arguments:
3 ['sys_args.py', '--s', '45']
```

The problem with using `sys.argv` is that you have no control over the arguments that can be passed to your application:

- You can't ignore arguments
- You can't create default arguments
- You can't really tell what is a valid argument at all

This is why using `argparse` is the way to go when working with Python's standard library. The `argparse` module is very powerful and useful. Let's think about a common process that a command line application follows:

- pass in a file
- do something to that file in your program
- output the result

Here is a generic example of how that might work. Go ahead and create `file_parser.py` and add the following code:

```
1 # file_parser.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser('File parser')
13     parser.add_argument('--infile', help='Input file')
14     parser.add_argument('--out', help='Output file')
15     args = parser.parse_args()
16     if args.infile:
17         file_parser(args.infile, args.out)
18
19 if __name__ == '__main__':
20     main()
```

The `file_parser()` function is where the logic for the parsing would go. For this example, it only takes in a file name and prints it back out. The `output_file` argument defaults to an empty string.

The meat of the program is in `main()` though. Here you create an instance of `argparse.ArgumentParser()` and give your parser a name. Then you add two arguments, `--infile` and `--out`. To use the parser, you need to call `parse_args()`, which will return whatever valid arguments were passed to your program. Finally, you check to see if the user used the `--infile` flag. If they did, then you run `file_parser()`.

Here is how you might run the code in your terminal:

```
1 $ python file_parser.py --infile something.txt
2 Processing something.txt
3 Finished processing
```

Here you run your script with the `--infile` flag along with a file name. This will run `main()` which in turns calls `file_parser()`.

The next step is to try your application using both command-line arguments you declared in your code:

```
1 $ python file_parser.py --infile something.txt --out output.txt
2 Processing something.txt
3 Finished processing
4 Creating output.txt
```

This time around, you get an extra line of output that mentions the output file name. This represents a branch in your code logic. When you specify an output file, you can have your code go through the process of generating that file using a new block of code or a function. If you do not specify an output file, then that block of code would not run.

When you create your command-line tool using argparse, you can easily add messages that help your users when they are unsure of how to correctly interact with your program.

Now it's time to find out how to get help from your application!

Creating Helpful Messages

The argparse library will automatically create a helpful message for your application using the information that you provided when you create each argument. Here is your code again:

```
1 # file_parser.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser('File parser')
13     parser.add_argument('--infile', help='Input file')
14     parser.add_argument('--out', help='Output file')
15     args = parser.parse_args()
16     if args.infile:
17         file_parser(args.infile, args.out)
18
19 if __name__ == '__main__':
20     main()
```

Now try running this code with the `-h` flag and you should see the following:

```
1 $ file_parser.py -h
2 usage: File parser [-h] [--infile INFILe] [--out OUT]
3
4 optional arguments:
5   -h, --help      show this help message and exit
6   --infile INFILe  Input file
7   --out OUT        Output file
```

The `help` parameter to `add_argument()` is used to create the help message above. The `-h` and `--help` options are added automatically by `argparse`. You can make your help more informative by giving it a `description` and an `epilog`.

Let's use them to improve your help messages. Start by copying the code from above into a new file named `file_parser_with_description.py`, then modify it to look like this:

```
1 # file_parser_with_description.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser(
13         'File parser',
14         description='PyParse - The File Processor',
15         epilog='Thank you for choosing PyParse!',
16         )
17     parser.add_argument('--infile', help='Input file for conversion')
18     parser.add_argument('--out', help='Converted output file')
19     args = parser.parse_args()
20     if args.infile:
21         file_parser(args.infile, args.out)
22
23 if __name__ == '__main__':
24     main()
```

Here you pass in the `description` and `epilog` arguments to `ArgumentParser`. You also update the `help` arguments to `add_argument()` to be more descriptive.

When you run this script with `-h` or `--help` after making these changes, you will see the following output:

```
1 $ python file_parser_with_description.py -h
2 usage: File parser [-h] [--infile INFILE] [--out OUT]
3
4 PyParse - The File Processor
5
6 optional arguments:
7   -h, --help      show this help message and exit
8   --infile INFILE  Input file for conversion
9   --out OUT        Converted output file
10
11 Thank you for choosing PyParse!
```

Now you can see the new description and epilog in your help output. This gives your command-line application some extra polish.

You can also disable help entirely in your application via the `add_help` argument to `ArgumentParser`. If you think that your help text is too wordy, you can disable it like this:

```
1 # file_parser_no_help.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser(
13         'File parser',
14         description='PyParse - The File Processor',
15         epilog='Thank you for choosing PyParse!',
16         add_help=False,
17     )
18     parser.add_argument('--infile', help='Input file for conversion')
19     parser.add_argument('--out', help='Converted output file')
20     args = parser.parse_args()
21     if args.infile:
22         file_parser(args.infile, args.out)
23
24 if __name__ == '__main__':
25     main()
```

By setting `add_help` to `False`, you are disabling the `-h` and `--help` flags.

You can see this demonstrated below:

```
1 $ python file_parser_no_help.py --help
2 usage: File parser [--infile INFILE] [--out OUT]
3 File parser: error: unrecognized arguments: --help
```

In the next section, you'll learn about adding aliases to your arguments!

Adding Aliases

An alias is a fancy word for using an alternate flag that does the same thing. For example, you learned that you can use both `-h` and `--help` to access your program's help message. `-h` is an alias for `--help`, and vice-versa

Look for the changes in the `parser.add_argument()` methods inside of `main()`:

```
1 # file_parser_aliases.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser(
13         'File parser',
14         description='PyParse - The File Processor',
15         epilog='Thank you for choosing PyParse!',
16         add_help=False,
17     )
18     parser.add_argument('-i', '--infile', help='Input file for conversion')
19     parser.add_argument('-o', '--out', help='Converted output file')
20     args = parser.parse_args()
21     if args.infile:
22         file_parser(args.infile, args.out)
23
24 if __name__ == '__main__':
25     main()
```

Here you change the first `add_argument()` to accept `-i` in addition to `--infile` and you also added `-o` to the second `add_argument()`. This allows you to run your code using two new shortcut flags.

Here's an example:

```
1 $ python3 file_parser_aliases.py -i something.txt -o output.txt
2 Processing something.txt
3 Finished processing
4 Creating output.txt
```

If you go looking through the `argparse` documentation, you will find that you can add aliases to subparsers too. A subparser is a way to create sub-commands in your application so that it can do other things. A good example is Docker, a virtualization or container application. It has a series of commands that you can run under `docker` as well as `docker compose` and more. Each of these commands has separate sub-commands that you can use.

Here is a typical `docker` command to run a container:

```
1 docker exec -it container_name bash
```

This will launch a container with `docker`. Whereas if you were to use `docker compose`, you would use a different set of commands. The `exec` and `compose` are examples of subparsers.

The topic of subparsers are outside the scope of this chapter. If you are interested in more details dive right into the documentation:

- <https://docs.python.org/3/library/argparse.html#sub-commands>

Using Mutually Exclusive Arguments

Sometimes you need to have your application accept some arguments but not others. For example, you might want to limit your application so that it can only create *or* delete files, but not both at once.

The `argparse` module provides the `add_mutually_exclusive_group()` method that does just that!

Change your two arguments to be mutually exclusive by adding them to a `group` object like in the example below:

```
1 # file_parser_exclusive.py
2
3 import argparse
4
5 def file_parser(input_file, output_file=''):
6     print(f'Processing {input_file}')
7     print('Finished processing')
8     if output_file:
9         print(f'Creating {output_file}')
10
11 def main():
12     parser = argparse.ArgumentParser(
13         'File parser',
14         description='PyParse - The File Processor',
15         epilog='Thank you for choosing PyParse!',
16         add_help=False,
17         )
18     group = parser.add_mutually_exclusive_group()
19     group.add_argument('-i', '--infile', help='Input file for conversion')
20     group.add_argument('-o', '--out', help='Converted output file')
21     args = parser.parse_args()
22     if args.infile:
23         file_parser(args.infile, args.out)
24
25 if __name__ == '__main__':
26     main()
```

First, you created a mutually exclusive group. Then, you added the `-i` and `-o` arguments to the group instead of to the `parser` object. Now these two arguments are mutually exclusive.

Here is what happens when you try to run your code with both arguments:

```
1 $ python3 file_parser_exclusive.py -i something.txt -o output.txt
2 usage: File parser [-i INFIL | -o OUT]
3 File parser: error: argument -o/--out: not allowed with argument -i/--infile
```

Running your code with both arguments causes your parser to show the user an error message that explains what they did wrong.

After covering all this information related to using `argparse`, you are ready to apply your new skills to create a simple search tool!

Creating a Simple Search Utility

Before starting to create an application, it is always good to figure out what you are trying to accomplish. The application you want to build in this section should be able to search for files of a specific file type. To make it more interesting, you can add an additional argument that allows you to optionally search for specific file sizes as well.

You can use Python's `glob` module for searching for file types. You can read all about this module here:

- <https://docs.python.org/3/library/glob.html>

There is also the `fnmatch` module, which `glob` itself uses. You should use `glob` for now as it is easier to use, but if you're interested in writing something more specialized, then `fnmatch` may be what you are looking for.

However, since you want to be able to optionally filter the files returned by the file size, you can use `pathlib` which includes a `glob`-like interface. The `glob` module itself does not provide file size information.

You can start by creating a file named `pysearch.py` and entering the following code:

```
1 # pysearch.py
2
3 import argparse
4 import pathlib
5
6
7 def search_folder(path, extension, file_size=None):
8     """
9         Search folder for files
10    """
11     folder = pathlib.Path(path)
12     files = list(folder.rglob(f'*.{extension}'))
13
14     if not files:
15         print(f'No files found with {extension=}')
16         return
17
18     if file_size is not None:
19         files = [
20             f
21             for f in files
22             if f.stat().st_size == file_size
23         ]
24
25     for f in files:
26         print(f'{f.name} ({f.stat().st_size} bytes)
```

```
22         if f.stat().st_size >= file_size
23     ]
24
25     print(f'{len(files)} *.{extension} files found: ')
26     for file_path in files:
27         print(file_path)
```

You start the code snippet above by importing `argparse` and `pathlib`. Next you create the `search_folder()` function which takes in three arguments:

- `path` - The folder to search within
- `extension` - The file extension to look for
- `file_size` - What file size to filter on in bytes

You turn the `path` into a `pathlib.Path` object and then use its `rglob()` method to search in the folder for the extension that the user passed in. If no files are found, you print out a meaningful message to the user and exit.

If any files are found, you check to see whether `file_size` has been set. If it was set, you use a list comprehension to filter out the files that are smaller than the specified `file_size`.

Next, you print out the number of files that were found and finally loop over these files to print out their names.

To make this all work correctly, you need to create a command-line interface. You can do that by adding a `main()` function that contains your `argparse` code like this:

```
1 def main():
2     parser = argparse.ArgumentParser(
3         'PySearch',
4         description='PySearch - The Python Powered File Searcher',
5     )
6     parser.add_argument('-p', '--path',
7                         help='The path to search for files',
8                         required=True,
9                         dest='path')
10    parser.add_argument('-e', '--ext',
11                         help='The extension to search for',
12                         required=True,
13                         dest='extension')
14    parser.add_argument('-s', '--size',
15                         help='The file size to filter on in bytes',
16                         type=int,
17                         dest='size',
```

```

18             default=None)
19
20     args = parser.parse_args()
21     search_folder(args.path, args.extension, args.size)
22
23 if __name__ == '__main__':
24     main()

```

This `ArgumentParser()` has three arguments added to it that correspond to the arguments that you pass to `search_folder()`. You make the `--path` and `--ext` arguments required while leaving the `--size` argument optional. Note that the `--size` argument is set to `type=int`, which means that you cannot pass it a string.

There is a new argument to the `add_argument()` function. It is the `dest` argument which you use to tell your argument parser where to save the arguments that are passed to them.

Here is an example run of the script:

```

1 $ python3 pysearch.py -p /Users/michael/Dropbox/python101code/chapter32_argparse -e \
2 py -s 650
3 6 *.py files found:
4 /Users/michael/Dropbox/python101code/chapter32_argparse/file_parser_aliases2.py
5 /Users/michael/Dropbox/python101code/chapter32_argparse/pysearch.py
6 /Users/michael/Dropbox/python101code/chapter32_argparse/file_parser_aliases.py
7 /Users/michael/Dropbox/python101code/chapter32_argparse/file_parser_with_description\
8 .py
9 /Users/michael/Dropbox/python101code/chapter32_argparse/file_parser_exclusive.py
10 /Users/michael/Dropbox/python101code/chapter32_argparse/file_parser_no_help.py

```

That worked quite well! Now try running it with `-s` and a string:

```

1 $ python3 pysearch.py -p /Users/michael/Dropbox/python101code/chapter32_argparse -e \
2 py -s python
3 usage: PySearch [-h] -p PATH -e EXTENSION [-s SIZE]
4 PySearch: error: argument -s/--size: invalid int value: 'python'

```

This time, you received an error because `-s` and `--size` only accept integers. Go try this code on your own machine and see if it works the way you want when you use `-s` with an integer.

Here are some ideas you can use to improve your version of the code:

- Handle the extensions better. Right now it will accept `*.py` which won't work the way you might expect
- Update the code so you can search for multiple extensions at once

- Update the code to filter on a range of file sizes (Ex. 1 MB - 5MB)

There are lots of other features and enhancements you can add to this code, such as adding error handling or unit tests.

Wrapping Up

The `argparse` module is full featured and can be used to create great, flexible command-line applications. In this chapter, you learned about the following:

- Parsing Arguments
- Creating Helpful Messages
- Adding Aliases
- Using Mutually Exclusive Arguments
- Creating a Simple Search Utility

You can do a lot more with the `argparse` module than what was covered in this chapter. Be sure to check out the documentation for full details. Now go ahead and give it a try yourself. You will find that once you get the hang of using `argparse`, you can create some really neat applications!

Review Questions

1. Which module in the standard library can you use to create a command-line application?
2. How do you add arguments to the `ArgumentParser()`?
3. How do you create helpful messages for your users?
4. Which method do you use to create a mutually exclusive group of commands?

Chapter 33 - How to Parse XML

The Extensible Markup Language, more commonly known as XML, is a markup language that is both human and machine-readable. XML remains a popular format for sharing data. The Python programming language has an XML library built-in that you can use to create, edit, or parse XML.

There are multiple sub-modules within the `xml` library. Here is a listing of the current sub-modules from the Python documentation:

- `xml.etree.ElementTree`: the ElementTree API, a simple and lightweight XML processor
- `xml.dom`: the DOM API definition
- `xml.dom.minidom`: a minimal DOM implementation
- `xml.dom.pulldom`: support for building partial DOM trees
- `xml.sax`: SAX2 base classes and convenience functions
- `xml.parsers.expat`: the Expat parser binding

The **Document Object Model (DOM)** is a way of describing the hierarchy of the nodes that make up XML. You can read about how Python works with it here:

- <https://docs.python.org/3/library/xml.dom.html>

The Simple API for XML (SAX) is another way to interface with XML from Python. The SAX parser will allow you to go step-by-step through an XML document and emit events as it goes. In contrast, the `minidom` parser will read the entire XML file into memory before you can start working on it.

The Expat parser is a non-validating XML parser and is normally used in conjunction with the SAX parser. However, if you want to have higher performance, you can use the Expat parser directly.

This chapter will focus on `xml.etree.ElementTree`. It is the easiest XML parser to use and understand within the sub-modules. You will learn how to do the following:

- Parsing XML with `ElementTree`
- Creating XML with `ElementTree`
- Editing XML with `ElementTree`
- Manipulating XML with `lxml`

Let's get started!

Parsing XML with `ElementTree`

You will probably spend more time parsing XML than you will creating or editing it. To get started, you will need some XML to parse. Here's a piece of simple XML that you should save in a file named `note.xml`:

```
1 <note_taker>
2   <note>
3     <to>Mike</to>
4     <from>Nadine</from>
5     <heading>Reminder</heading>
6     <body>Don't forget the milk</body>
7   </note>
8   <note>
9     <to>Nicole</to>
10    <from>Nadine</from>
11    <heading>Appointment</heading>
12    <body>Eye doctor</body>
13  </note>
14 </note_taker>
```

This XML represents a person's notes that they have saved for themselves. They might be reminders, or notes for research of some sort. Your job is to figure out how to parse this XML with Python using ElementTree.

To start extracting the relevant information, create a file named `parse_xml.py` and add the following code:

```
1 # parse_xml.py
2
3 from xml.etree.ElementTree import ElementTree
4
5 def parse_xml(xml_file):
6     tree = ElementTree(file=xml_file)
7     root_element = tree.getroot()
8     print(f"The root element's tag is '{root_element.tag}'")
9
10    for child_element in root_element:
11        print(f'{child_element.tag}={child_element.text}')
12        if child_element.tag == 'note':
13            for note_element in child_element:
14                print(f'{note_element.tag}={note_element.text}')
15
16
17 if __name__ == '__main__':
18     parse_xml('note.xml')
```

Here you import `ElementTree` from `xml.etree.ElementTree` and then create `parse_xml()`, which takes in a file path to an XML file. To open the file, you use `ElementTree()` directly. This returns an

`xml.etree.ElementTree.ElementTree` object, which you can extract a root element from. The XML is transformed into a tree-like structure by `ElementTree`. The root is the origin element in the XML, which in this case is `note_taker`.

XML is very similar to HTML. If you already know HTML, then you will be able to pick up XML pretty quickly. A tag in XML is created using the following syntax: `<tag_name>tag_text</tag_name>`. The `<tag_name>` is the beginning of the tag, while `tag_text` is the value of the tag. To tell XML where a tag end, you use a forward-slash with the name of the tag inside: `</tag_name>`.

The `for` loop above will iterate over the XML elements that are within the `root_element`. For each element, you will print out the tag and value of the tag. If the tag equals “note”, then you loop over that element’s children to print out all the tags and values within that element.

In other words, when you parse XML, most of the parsers that you write will be tied pretty closely to the XML’s structure.

Instead of iterating directly over the `root_element`, you could also use `root.getchildren()` to get the children from the `root_element`. Feel free to give that a try if you feel adventurous.

When you run this code, you will see the following output:

```
1 The root element's tag is 'note_taker'  
2 child_element.tag='note', child_element.text='\n      '  
3 note_element.tag='to', note_element.text='Mike'  
4 note_element.tag='from', note_element.text='Nadine'  
5 note_element.tag='heading', note_element.text='Reminder'  
6 note_element.tag='body', note_element.text="Don't forget the milk"  
7 child_element.tag='note', child_element.text='\n      '  
8 note_element.tag='to', note_element.text='Nicole'  
9 note_element.tag='from', note_element.text='Nadine'  
10 note_element.tag='heading', note_element.text='Appointment'  
11 note_element.tag='body', note_element.text='Eye doctor'
```

If you have complex XML or large XML files, than you should use `xml.etree.cElementTree` instead of `xml.etree.ElementTree`. Note the “c” in front of `ElementTree`. The `cElementTree` version is written in the C programming language, which makes it a much faster XML parser than the regular `ElementTree`. The syntax is also the same!

You can iterate over the tree object itself rather than iterating over the root. Here is an example:

```
1 # xml_tree_iterator.py
2
3 from xml.etree.cElementTree import ElementTree
4
5 def parse_xml(xml_file):
6     tree = ElementTree(file=xml_file)
7     print("Iterating using a tree iterator")
8     for elem in tree.iter():
9         print(f'{elem.tag}, {elem.text}')
10
11
12 if __name__ == '__main__':
13     parse_xml('note.xml')
```

In this case, you use the `iter()` method, which allows you to iterate over all the elements in the XML. The benefit of iterating over the tree itself is that you can force it to filter based on the tag, although here you aren't setting that argument.

This code's output is very similar to the output from your previous example:

```
1 Iterating using a tree iterator
2 elem.tag='note_taker', elem.text='\n '
3 elem.tag='note', elem.text='\n '
4 elem.tag='to', elem.text='Mike'
5 elem.tag='from', elem.text='Nadine'
6 elem.tag='heading', elem.text='Reminder'
7 elem.tag='body', elem.text="Don't forget the milk"
8 elem.tag='note', elem.text='\n '
9 elem.tag='to', elem.text='Nicole'
10 elem.tag='from', elem.text='Nadine'
11 elem.tag='heading', elem.text='Appointment'
12 elem.tag='body', elem.text='Eye doctor'
```

This looks like a good start. You now know how to extract both the tags as well as their values from any element of your XML.

Now it's time to learn how to create XML using `ElementTree`!

Creating XML with `ElementTree`

Creating XML using `ElementTree` is straightforward, but a bit tedious. The reason being that you have to create the tag and the text separately. Let's create your own XML with Python. Start off by creating a new file named `create_xml.py` and add the following code:

```
1 # create_xml.py
2
3 import xml.etree.ElementTree as ET
4
5 def create_xml(xml_file):
6     root_element = ET.Element('note_taker')
7     note_element = ET.Element('note')
8     root_element.append(note_element)
9
10    # add note sub-elements
11    to_element = ET.SubElement(note_element, 'to')
12    to_element.text = 'Mike'
13    from_element = ET.SubElement(note_element, 'from')
14    from_element.text = 'Nick'
15    heading_element = ET.SubElement(note_element, 'heading')
16    heading_element.text = 'Appointment'
17    body_element = ET.SubElement(note_element, 'body')
18    body_element.text = 'blah blah'
19
20    tree = ET.ElementTree(root_element)
21    with open(xml_file, "wb") as fh:
22        tree.write(fh)
23
24 if __name__ == '__main__':
25     create_xml('test_create.xml')
```

To create a tag, you can use `Element()` or `SubElement()`. The `Element()` class is for creating parent elements. For example, you will use it to create your root element and then append the `note` element to it. Then, to add sub-elements to the `note` element, you can use the `SubElement()` function, which takes in the parent `Element()` and the text string of the sub-element.

If an `Element()` or `SubElement()` has a value, you can set it via the `text` property (see examples above).

When you are done creating the XML, you can transform it into an XML tree structure by using `ElementTree()`. Then you can write it out to disk via `tree.write()` and passing it the open file handler. Note that when you write the file out, the XML will all be on one line.

The `ElementTree` module does not support outputting “pretty-print” XML. To do that, you would need to use a different XML library, such as the third-party `lxml` module.

After learning how to parse and create XML, you will now learn how to edit a pre-existing piece of XML!

Editing XML with ElementTree

Editing XML with Python is nice. Of course, you need some XML to edit first. You can use the original note.xml file from earlier. Here it is again for your convenience:

```
1 <?xml version="1.0" ?>
2 <note_taker>
3   <note>
4     <to>Mike</to>
5     <from>Nadine</from>
6     <heading>Reminder</heading>
7     <body>Don't forget the milk</body>
8   </note>
9   <note>
10    <to>Nicole</to>
11    <from>Nadine</from>
12    <heading>Appointment</heading>
13    <body>Eye doctor</body>
14  </note>
15 </note_taker>
```

Let's write some code to edit the "from" tag of this XML and change it to someone else for all instances of that tag. Go ahead and create a Python file named edit_xml.py.

Now enter the following code in it:

```
1 # edit_xml.py
2
3 import xml.etree.cElementTree as ET
4
5 def edit_xml(xml_file, output_file, from_person):
6     tree = ET.ElementTree(file=xml_file)
7     root = tree.getroot()
8
9     for from_element in tree.iter(tag='from'):
10         from_element.text = from_person
11
12     tree = ET.ElementTree(root)
13     with open(output_file, "wb") as f:
14         tree.write(f)
15
16 if __name__ == '__main__':
17     edit_xml('note.xml', 'output.xml', 'Guido')
```

Here you create an `edit_xml()` function that takes in an input `xml_file`, the `output_file` name and what to change the `from_person` to. Next, you open the file and transform it into a `tree` object. Then you extract the root of the XML. Then you use the tree's `iter()` function to iterate over every tag that is named "from". For each of those elements, you change its text value to `from_person`.

Finally you write the XML to disk using the passed in file name, `output_file`. The contents of the XML file will now look like this:

```
1 <note_taker>
2   <note>
3     <to>Mike</to>
4     <from>Guido</from>
5     <heading>Reminder</heading>
6     <body>Don't forget the milk</body>
7   </note>
8   <note>
9     <to>Nicole</to>
10    <from>Guido</from>
11    <heading>Appointment</heading>
12    <body>Eye doctor</body>
13  </note>
14 </note_taker>
```

You now know the basics of editing an XML file with Python's `ElementTree` module.

Let's look at using an alternative XML package from outside the standard library next!

Manipulating XML with `lxml`

There are other XML parsing libraries for Python. One of the most popular packages is `lxml`, which is a Python binding to two C libraries: `libxml2` and `libxslt`. It has an `ElementTree` implementation called `etree` that has almost the exact same API as the Python version.

The `lxml` package also has an `objectify` sub-module which allows you to turn the XML into a Python object. This can be very convenient when working with XML.

You can install `lxml` using `pip`:

```
1 $ python -m pip install lxml
```

Now that you have `lxml` installed, you can get started using it. To see a little of the power of this package, you can create a file named `parse_xml_with_lxml.py` and enter the following code:

```
1 # parse_xml_with_lxml.py
2
3 from lxml import etree, objectify
4
5 def parse_xml(xml_file):
6     with open(xml_file) as f:
7         xml = f.read()
8
9     root = objectify.fromstring(xml)
10
11    # Get an element
12    to = root.note.to
13    print(f'The {to=}')
14
15    # print out all the note element's tags and text values
16    for note in root.getchildren():
17        for note_element in note.getchildren():
18            print(f'{note_element.tag=}, {note_element.text=}')
19        print()
20
21    # modify a text value
22    print(f'Original: {root.note.to=}')
23    root.note.to = 'Guido'
24    print(f'Modified: {root.note.to=}')
25
26    # add a new element
27    root.note.new_element = "I'm new!"
28
29    # cleanup the XML before writing to disk
30    objectify.deannotate(root)
31    etree.cleanup_namespaces(root)
32    obj_xml = etree.tostring(root, pretty_print=True)
33
34    # save your xml
35    with open("lxml_output.xml", "wb") as f:
36        f.write(obj_xml)
37
38 if __name__ == '__main__':
39     parse_xml('note.xml')
```

This code is a little long, so let's go over it piece-by-piece, starting by looking at the beginning of the `parse_xml()` function:

```
1 # parse_xml_with_lxml.py
2
3 from lxml import etree, objectify
4
5 def parse_xml(xml_file):
6     with open(xml_file) as f:
7         xml = f.read()
8
9     root = objectify.fromstring(xml)
```

The first step is to import `etree` and `objectify` from `lxml`. Then you create a `parse_xml()` function. Next, you will open the `xml_file` and read its content. Then you transform the XML into a Python object using `objectify.fromstring(xml)`.

You can now access elements directly like you would with any other Python object:

```
1 # Get an element
2 to = root.note.to
3 print(f'The {to}')
```

In the above example, you extract the first “note” object’s “to” field and print it out.

Now let’s look at how to iterate over the elements next:

```
1 # print out all the note element's tags and text values
2 for note in root.getchildren():
3     for note_element in note.getchildren():
4         print(f'{note_element.tag=}, {note_element.text=}')
5     print()
```

When you use `lxml`, the primary method of iterating is by using the `getchildren()` method. This code will iterate over the `note` elements and then the children sub-elements of the `note` element.

The next snippet will show you how to change an XML element with `lxml`:

```
1 # modify a text value
2 print(f'Original: {root.note.to}')
3 root.note.to = 'Guido'
4 print(f'Modified: {root.note.to}')
```

Because `objectify` turns the XML into an object, you can assign a new value to a tag directly, as you do in the above example. This simplifies accessing and modifying the tags considerably.

If you would like to add a new element to the XML, you can do so like this:

```
1 # add a new element
2 root.note.new_element = "I'm new!"
```

This will add a new tag to the first note XML element named `new_element` and set its text value to “I’m new!”.

Finally, last few lines of code demonstrate how to write the new XML to disk:

```
1 # cleanup the XML before writing to disk
2 objectify.deannotate(root)
3 etree.cleanup_namespaces(root)
4 obj_xml = etree.tostring(root, pretty_print=True)
5
6 # save your xml
7 with open("lxml_output.xml", "wb") as f:
8     f.write(obj_xml)
```

The `deannotate()` and `cleanup_namespaces()` functions will remove some extra lines of output from the XML that `lxml` adds. To make the XML output look nice and not all on one line, you will use `etree.tostring(root, pretty_print=True)`, which indents the XML elements in a nice manner.

Finally you write the XML to disk by using the file object’s `write()` method. In the previous example, you used the tree object’s `write()` method, but here you used `objectify` to create a Python object, which means you have to write the XML to disk differently.

When you run this code, you will see the following output to stdout:

```
1 The to='Mike'
2 note_element.tag='to', note_element.text='Mike'
3 note_element.tag='from', note_element.text='Nadine'
4 note_element.tag='heading', note_element.text='Reminder'
5 note_element.tag='body', note_element.text="Don't forget the milk"
6
7 note_element.tag='to', note_element.text='Nicole'
8 note_element.tag='from', note_element.text='Nadine'
9 note_element.tag='heading', note_element.text='Appointment'
10 note_element.tag='body', note_element.text='Eye doctor'
11
12 Original: root.note.to='Mike'
13 Modified: root.note.to='Guido'
```

The XML that your code generates will look like this:

```
1 <note_taker>
2   <note>
3     <to>Guido</to>
4     <from>Nadine</from>
5     <heading>Reminder</heading>
6     <body>Don't forget the milk</body>
7     <new_element>I'm new!</new_element>
8   </note>
9   <note>
10    <to>Nicole</to>
11    <from>Nadine</from>
12    <heading>Appointment</heading>
13    <body>Eye doctor</body>
14  </note>
15 </note_taker>
```

The `lxml` library is quite nice overall and well worth your time to learn. It's also very fast and efficient to use.

Wrapping Up

Python comes with several different XML parsers in its standard library. One of the easiest to use is the `ElementTree` API. In this chapter, you learned how to do the following:

- Parsing XML with `ElementTree`
- Creating XML with `ElementTree`
- Editing XML with `ElementTree`
- Manipulating XML with `lxml`

In addition to the built-in XML parsers, there are many different 3rd party Python packages for working with XML. One of the most popular is the `lxml` package. Regardless of which XML parsing library you end up using, Python can be used effectively for all your XML needs.

Review Questions

1. What XML modules are available in Python's standard library?
2. How do you access an XML tag using `ElementTree`?
3. How do you get the root element using the `ElementTree` API?

Chapter 34 - How to Parse JSON

JavaScript Object Notation, more commonly known as JSON, is a lightweight data interchange format inspired by JavaScript object literal syntax. JSON is easy for humans to read and write. It is also easy for computers to parse and generate. JSON is used for storing and exchanging data in much the same way that XML is used.

Python has a built-in library called `json` that you can use for creating, editing and parsing JSON. You can read all about this library here:

- <https://docs.python.org/3/library/json.html>

It would probably be helpful to know what JSON looks like. Here is an example of JSON from <https://json.org>:

```
1 {"menu": {  
2     "id": "file",  
3     "value": "File",  
4     "popup": {  
5         "menuitem": [  
6             {"value": "New", "onclick": "CreateNewDoc()"},  
7             {"value": "Open", "onclick": "OpenDoc()"},  
8             {"value": "Close", "onclick": "CloseDoc()"}  
9         ]  
10    }  
11 }}
```

From Python's point of view, this JSON is a nested Python dictionary. You will find that JSON is always translated into some kind of native Python data type. In this chapter, you will learn about the following:

- Encoding a JSON String
- Decoding a JSON String
- Saving JSON to Disk
- Loading JSON from Disk
- Validating JSON with `json.tool`

JSON is a very popular format that is often used in web applications. You will find that knowing how to interact with JSON using Python is useful in your own work.

Let's get started!

Encoding a JSON String

Python's `json` module uses `dumps()` to serialize an object to a string. The "s" in `dumps()` stands for "string". It's easier to see how this works by using the `json` module in some code:

```
1  >>> import json
2  >>> j = {"menu": {
3  ...     "id": "file",
4  ...     "value": "File",
5  ...     "popup": {
6  ...         "menuitem": [
7  ...             {"value": "New", "onclick": "CreateNewDoc()"},
8  ...             {"value": "Open", "onclick": "OpenDoc()"},
9  ...             {"value": "Close", "onclick": "CloseDoc()"}
10 ...         ]
11 ...     }
12 ... }}
```

```
13  >>> json.dumps(j)
14  '{"menu": {"id": "file", "value": "File", "popup": {"menuitem": [{"value": "New", '
15  '"onclick": "CreateNewDoc()"}, {"value": "Open", "onclick": "OpenDoc()"}, '
16  '{"value": "Close", "onclick": "CloseDoc()"}]}}}
```

Here you use `json.dumps()`, which transforms the Python dictionary into a JSON string. The example's output was modified to wrap the string for print. Otherwise the string would all be on one line.

Now you're ready to learn how to write an object to disk!

Saving JSON to Disk

Python's `json` module uses the `dump()` function to serialize or encode an object as a JSON formatted stream to a file-like object. File-like objects in Python are things like file handlers or objects that you create using Python's `io` module.

Go ahead and create a file named `create_json_file.py` and add the following code to it:

```
1 # create_json_file.py
2
3 import json
4
5 def create_json_file(path, obj):
6     with open(path, 'w') as fh:
7         json.dump(obj, fh)
8
9 if __name__ == '__main__':
10    j = {"menu": {
11        "id": "file",
12        "value": "File",
13        "popup": {
14            "menuitem": [
15                {"value": "New", "onclick": "CreateNewDoc()"},
16                {"value": "Open", "onclick": "OpenDoc()"},
17                {"value": "Close", "onclick": "CloseDoc()"}
18            ]
19        }
20    }}
21    create_json_file('test.json', j)
```

In this example, you use `json.dump()`, which is for writing to a file or file-like object. It will write to the file-handler, `fh`.

Now you can learn about decoding a JSON string!

Decoding a JSON String

Decoding or deserializing a JSON string is done via the `loads()` method. `loads()` is the companion function to `dumps()`. Here is an example of its use:

```
1 >>> import json
2 >>> j_str = """{"menu": {
3 ...     "id": "file",
4 ...     "value": "File",
5 ...     "popup": {
6 ...         "menuitem": [
7 ...             {"value": "New", "onclick": "CreateNewDoc()"},
8 ...             {"value": "Open", "onclick": "OpenDoc()"},
9 ...             {"value": "Close", "onclick": "CloseDoc()"}
10 ...     ]
11 ... }}
```

```
11 ...    }
12 ...  }}
13 ...
14 >>> j_obj = json.loads(j_str)
15 >>> type(j_obj)
16 <class 'dict'>
```

Here you recreate the JSON code from earlier as a Python multi-line string. Then you load the JSON string using `json.loads()`, which converts it to a Python object. In this case, it converts the JSON to a Python dictionary.

Now you are ready to learn how to load JSON from a file!

Loading JSON from Disk

Loading JSON from a file is done using `json.load()`. Here is an example:

```
1 # load_json_file.py
2
3 import json
4
5 def load_json_file(path):
6     with open(path) as fh:
7         j_obj = json.load(fh)
8     print(type(j_obj))
9
10
11 if __name__ == '__main__':
12     load_json_file('example.json')
```

In this code, you open the passed in file as you have seen before. Then you pass the file-handler, `fh`, to `json.load()`, which will transform the JSON into a Python object.

You can also use Python's `json` module to validate JSON. You will find out how to do that next.

Validating JSON with `json.tool`

Python's `json` module provides a tool you can run on the command line to check and see if the JSON has the correct syntax. Here are a couple of examples:

```
1 $ echo '{1.2:3.4}' | python -m json.tool
2 Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
3 $ echo '{"1.2":3.4}' | python -m json.tool
4 {
5     "1.2": 3.4
6 }
```

The first call passes the string, '{1.2:3.4}' to `json.tool`, which tells you that there is something wrong with the JSON code. The second example shows you how to fix the issue. When the fixed string is passed in to `json.tool`, it will “pretty-print” the JSON back out instead of emitting an error.

Wrapping Up

The JSON format is used very often when working with web APIs and web frameworks. The Python language provides a nice tool for you to use to convert JSON to Python objects and back again in the `json` library.

In this chapter, you learned about the following:

- Encoding a JSON String
- Decoding a JSON String
- Saving JSON to Disk
- Loading JSON from Disk
- Validating JSON with `json.tool`

You now have another useful tool that you can use Python for. With a little practice, you will be working with JSON in no time!

Review Questions

1. What is JSON?
2. How do you decode a JSON string in Python?
3. How do you save JSON to disk with Python?

Chapter 35 - How to Scrape a Website

The Internet is the host of much of the world's information, both past and present. You can find history, news, comics, and much more on the Internet. As a software developer, you might want to gain access to the troves of data that exist on the Internet. Some web pages provide a free or paid **Application Programming Interface (API)** that you can use to programmatically access their data. However, most websites do not offer an API, so you must resort to **scraping** them to gain programmatic access to the information they provide.

Scraping a website refers to fetching the HTML content of a page from the Internet, parsing the HTML content, and extracting the bits and pieces that interest you. In this chapter you will learn about the following:

- Rules for Web Scraping
- Preparing to Scrape a Website
- Scraping a Website
- Downloading a File

There are several web scraping packages for Python. The most popular are **Beautiful Soup** and **Scrapy**. This chapter will focus on Beautiful Soup.

Let's get started!

Rules for Web Scraping

Most websites have rules regarding their content. Sometimes it's just copyright information that you will need to abide by, but here are some tips to keep in mind:

- Always check the terms and conditions on a website **before** you scrape from them. Violating the terms can land you in legal trouble!
- Commercial websites usually have limits on how often you can scrape and what you can scrape
- Your application can access a site much faster than a human can, so don't access a site too often in a short amount of time. This can cause a website to slow down or fail and may be illegal
- Websites change constantly, so you can expect your scraper to fail some day too
- When scraping data, you need to realize that you will get a lot of data you don't care about. Be prepared to do a lot of data cleaning to extract the information that is relevant for you.

Now let's get set up so you can start scraping!

Preparing to Scrape a Website

Beautiful Soup is the most popular web scraping package. It is not included with Python, so you will need to install it with pip:

```
1 pip install beautifulsoup4
```

Beautiful Soup needs something to parse, which means you need to have a way to download a web page. You can do that using any of the following:

- `urllib` - comes with Python
- `requests` - a popular 3rd party Python package
- `httpx` - another popular 3rd party Python package

The latter two are easier to use than Python's own library, but that also means that you have to go through an extra step of installing one or more new packages. For the purposes of this chapter, you will use `urllib`. However, if you need to use authentication with a web page before you can download, then you should look at one of those other packages as they will make that much easier.

There is one crucial tip to keep in mind when it comes to scraping a web page: **Your web browser can help you.** Most web browsers come with developer tools built-in that you can use to inspect websites. The path to open up those tools is slightly different across browsers, though. Let's look at how that works by opening up my blog in Google Chrome:

<https://www.blog.pythonlibrary.org>

Then right-click anywhere on the web page and choose the **Inspect** option:

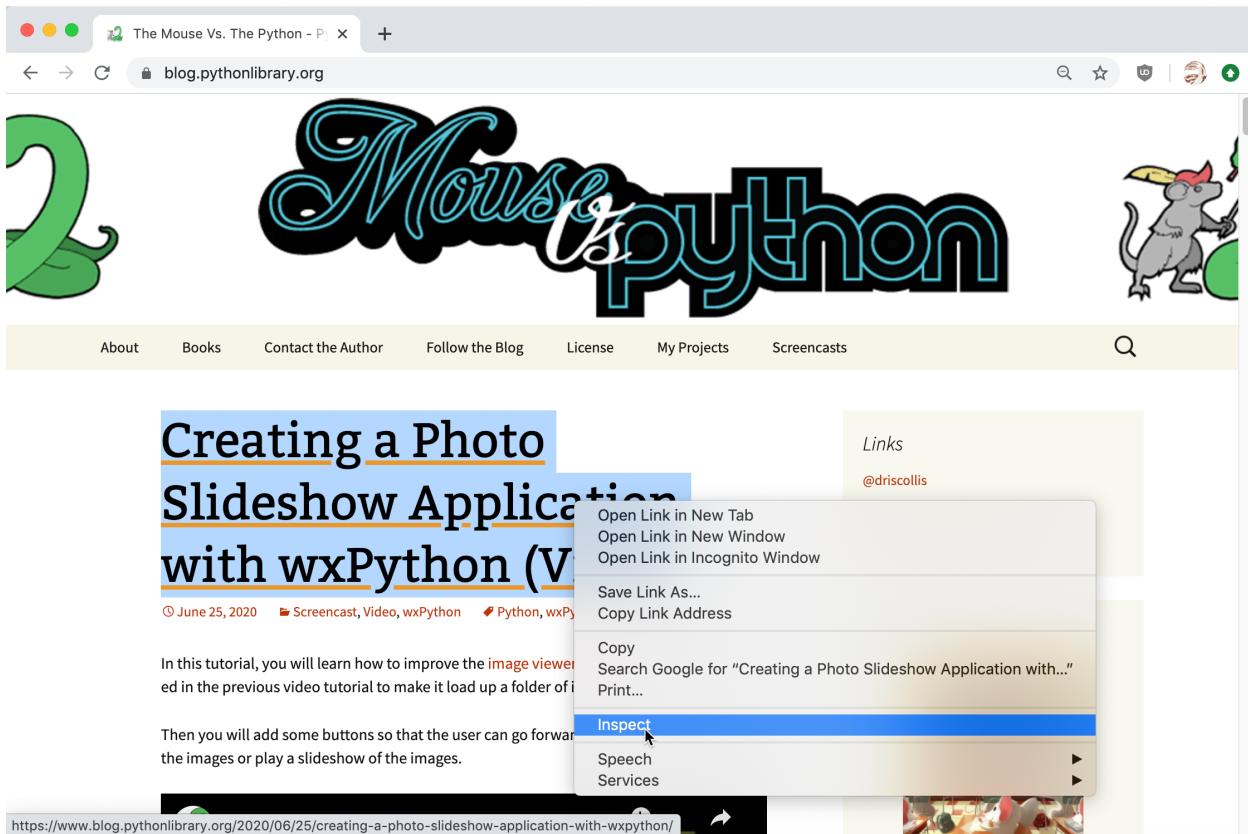


Fig. 35-1: Inspect with Google Chrome

Alternatively, you can also open up the developer tools through the menu *View → Developer → Developer Tools*. After choosing to inspect an item on the web page, Google Chrome will open up a sidebar on the right of your browser that will look something like this:

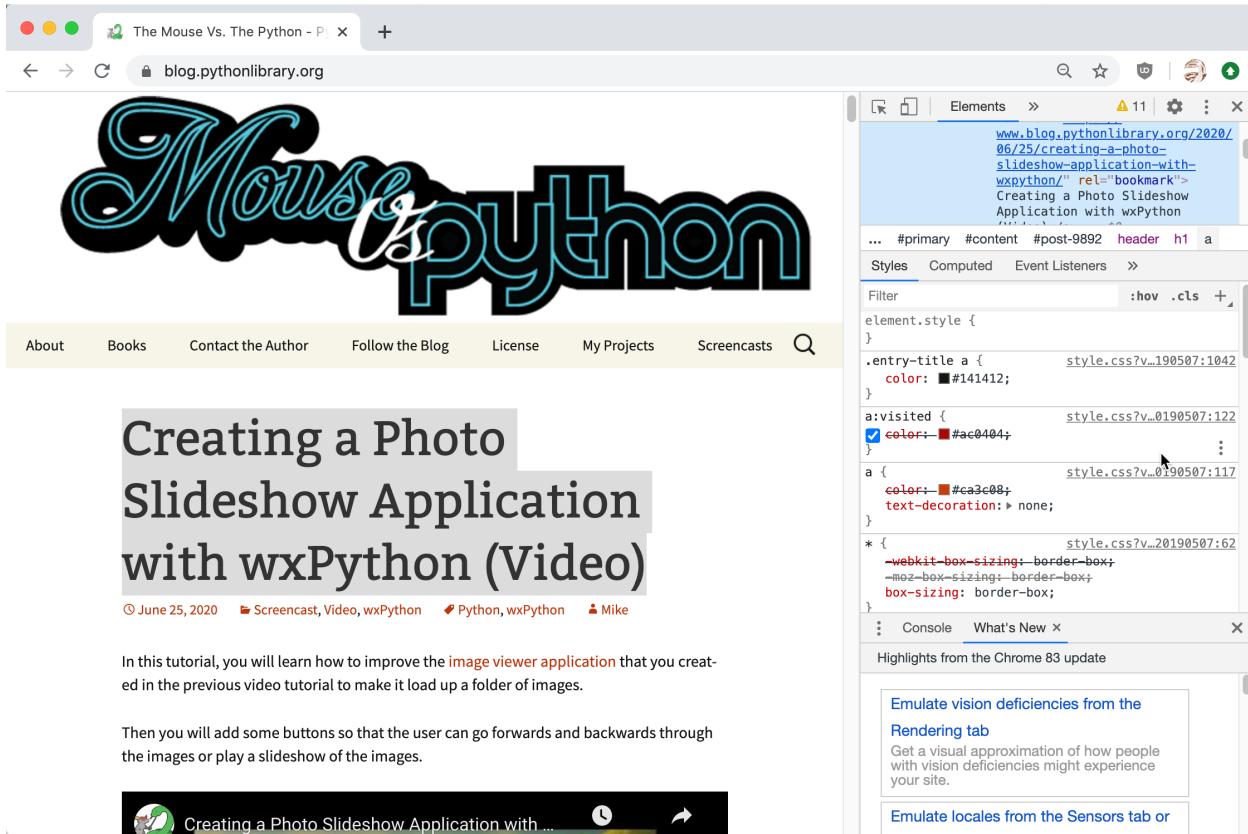


Fig. 35-2: Inspect Sidebar with Google Chrome

You can now select elements within the sidebar and your browser will highlight the relevant parts of your site on the left.

Mozilla Firefox has a very similar interface except that instead of a sidebar, it appears along the bottom of your browser. Try both of these tools and familiarize yourself with their functions. You will see that developer tools are very similar to each other, no matter which modern browser you are working with. Once you've gained some understanding of how your browser's developer tools work and what they offer, you'll be able to use them to scrape a website much more effectively.

After understanding what tools you can use to inspect and learn about your website's structure, you are ready to start scraping it.

Scraping a Website

Let's pretend that you have been tasked with getting all the current titles and links to my blog. This is a common task when you are building a website that aggregates data from other sites.

The first step in the process is to figure out how to download the main page's HTML. Here is some example code:

```
1 import urllib.request  
2  
3 url = 'https://www.blog.pythonlibrary.org'  
4 with urllib.request.urlopen(url) as response:  
5     html = response.read()
```

This short code snippet will make a request to my server, fetch the website's HTML, and store it in the variable, `html`. This is a nice little script, but not very reusable. You can take this code snippet and turn it into a function. Open up a new file named `scraper.py` and add the following:

```
1 # scraper.py  
2  
3 import urllib.request  
4  
5 def download_html(url):  
6     with urllib.request.urlopen(url) as response:  
7         html = response.read()  
8     return html  
9  
10 if __name__ == '__main__':  
11     url = 'https://www.blog.pythonlibrary.org'  
12     html = download_html(url)
```

When you run this code, it should return the HTML of the main page of my blog. If you get an `SSL: CERTIFICATE_VERIFY_FAILED` on macOS, then you will need to go where you installed Python and run the `Install Certificates.command` to fix that issue. You can read more about resolving this issue at the following link:

- <https://stackoverflow.com/q/42098126>

If you use the techniques from the previous section to inspect the title of an article from my blog, you will see that they are contained in “`h1`” tags. With that in mind, you can update the `scraper.py` program to import `BeautifulSoup` and search for all the “`h1`” tags:

```
1 # scraper.py
2
3 import urllib.request
4 from bs4 import BeautifulSoup
5
6
7 def download_html(url):
8     with urllib.request.urlopen(url) as response:
9         html = response.read()
10    return html
11
12 def scraper(url):
13     html = download_html(url)
14     soup = BeautifulSoup(html, 'html.parser')
15
16     title_links = soup.findAll('h1')
17     articles = {}
18     for link in title_links:
19         if link.a:
20             articles[link.a['href']] = link.text.strip()
21
22     for article in articles:
23         print(f'{articles[article]} - {article}')
24
25
26 if __name__ == '__main__':
27     url = 'https://www.blog.pythonlibrary.org'
28     scraper(url)
```

In this code, you add a new import for BeautifulSoup and a new function: `scraper()`. You use `scraper()` to call `download_html()` and then parse the HTML with `BeautifulSoup()`. Next, you use `findAll()` to search for all the “h1” tags. This returns a `ResultSet` object, which you can iterate over.

If an item in the result set has the attribute “a”, this means that the “h1” title element contains a HTML link element, which looks similar to this: `Link Name`. Open up your browser’s developer tools and verify that you can see these link elements nested in some of the titles. If your code discovers an “a” element in your title, this also means that this HTML element has an HTML attribute called `href`. This is the **HTML attribute** on a link element that contains the URL value that you are interested in.

You can use that information to grab the hyperlink itself and make it into a key for your `articles` dictionary. Then set the value to the title of the article. Finally, you loop over the `articles` and print out the title and corresponding link.

Give it a try and see what the output is. If you would like a challenge, try to figure out how to scrape

all the links on the page instead of only the hyperlinks that are nested in “h1” headings.

Now let’s move on and learn how to download a file from the Internet!

Downloading a File

In the previous section you learned how to download the HTML of a web page. However, web pages host much more than HTML. They can also contain other types of content. For example, they can contain images, PDFs, Excel documents, and much more. If your browser can download a file, then there is some way for Python to do so too!

As you know, the Python programming language comes with a module named `urllib`. You can use `urllib` for downloading files. If you need to login to a website before downloading a file, it may be worth looking at a 3rd party module such as `requests` or `httpx` because they make working with credentials much easier. The `urllib` library works for these too, but it takes significantly more code.

Let’s find out how you can use the `urllib` module to download a binary file:

```
1 import urllib.request
2
3 def download_file(url):
4     urllib.request.urlretrieve(url, "code.zip")
5
6 if __name__ == '__main__':
7     url = 'http://www.blog.pythonlibrary.org/wp-content/uploads/2012/06/wxDbViewer.z\
8 ip'
9     download_file(url)
```

In this example, you use `urllib.request.urlretrieve()` to download the specified URL. This function takes as input the URL to download, as well as the path to save the content. You use `code.zip` as the name of the output file here.

There is yet another way to download a file using `urllib.request.urlopen()`. Here’s an example:

```
1 import urllib.request
2
3 def alternate_download(url):
4     with urllib.request.urlopen(url) as response:
5         data = response.read()
6         with open("code2.zip", "wb") as code:
7             code.write(data)
8
9 if __name__ == '__main__':
10    url = 'http://www.blog.pythonlibrary.org/wp-content/uploads/2012/06/wxDbViewer.z\
```

```
11 ip'  
12     alternate_download(url)
```

When you open a URL using `urllib.request.urlopen()`, it returns a file-like object that you can use to `read()` the file. Then you can create a file on your computer using Python's `open()` function and write that file out. Since this is a binary file, you will need to open the file for writing in binary mode (i.e. `wb`). You can use this method of downloading a file when you want to let the user know the progress of the download.

If you are feeling adventurous, you should try using Beautiful Soup to parse a web page for images or some other binary file type and try downloading one or more of those. Just be careful that you don't overwhelm the target website.

Wrapping Up

Web scraping is a bit of an art. You will need to learn how to navigate a website programmatically to succeed. You will also need to realize up-front that your code will not work forever as websites change often. As with any skill, it comes down to putting in the time and training your web scraping skills in order to get good at it. In this chapter, you learned about the following:

- Rules for Web Scraping
- Preparing to Scrape a Website
- Scraping a Website
- Downloading a File

When you venture deeper into web scraping, you will encounter websites that make scraping more difficult than others. Most websites today contain JavaScript code that dynamically generates the website content through code execution in your browser. This means that straight-up HTTP requests with Python, as you were using above, won't be enough to get the content that you are interested in.

In these scenarios you will need to use other tools than the ones mentioned here. For example, you might find **Selenium** useful for automating interactions on a website, and **phantomJS** for scraping a site that gets dynamically generated with JavaScript. Selenium itself can be used for web scraping too.

This chapter only scratches the surface of what you can do with web scraping and Python. Go out and start practicing some scraping on your own!

Review Questions

1. What are some popular Python web scraping packages?

2. How do you examine a web page with your browser?
3. Which Python module from the standard library do you use to download a file?

Chapter 36 - How to Work with CSV files

There are many common file types that you will need to work with as a software developer. One such format is the CSV file. CSV stands for “Comma-Separated Values” and is a text file format that uses a comma as a delimiter to separate values from one another. Each row is its own record and each value is its own field. Most CSV files have records that are all the same length.

Unfortunately, CSV is not a standardized file format, which makes using them directly more complicated, especially when the data of an individual field itself contains commas or line breaks. Some organizations use quotation marks as an attempt to solve this problem, but then the issue is shifted to what happens when you need quotation marks in that field?

A couple of the benefits of CSV files is that they are human readable, and most spreadsheet software can use them. For example, Microsoft Excel and Libre Office will happily open CSV files for you and format them into rows and columns.

Python has made creating and reading CSV files much easier via its `csv` library. It works with most CSV files out of the box and allows some customization of its readers and writers. A reader is what the `csv` module uses to parse the CSV file, while a writer is used to create/update csv files.

In this chapter, you will learn about the following:

- Reading a CSV File
- Reading a CSV File with `DictReader`
- Writing a CSV File
- Writing a CSV File with `DictWriter`

If you need more information about the `csv` module, be sure to check out the documentation here:

- <https://docs.python.org/3/library/csv.html>

Let's start learning how to work with CSV files!

Reading a CSV File

Reading CSV files with Python is pretty straight-forward once you know how to do so. The first piece of the puzzle is to have a CSV file that you want to read. For the purposes of this section, you can create one named `books.csv` and copy the following text into it:

```
1 book_title,author,publisher,pub_date,isbn
2 Python 101,Mike Driscoll, Mike Driscoll,2020,123456789
3 wxPython Recipes,Mike Driscoll,Apress,2018,978-1-4842-3237-8
4 Python Interviews,Mike Driscoll,Packt Publishing,2018,9781788399081
```

The first row of data is known as the *header* record. It explains what each field of data represents. Let's write some code to read this CSV file into Python so you can work with its content. Go ahead and create a file named `csv_reader.py` and enter the following code into it:

```
1 # csv_reader.py
2
3 import csv
4
5 def process_csv(path):
6     with open(path) as csvfile:
7         reader = csv.reader(csvfile)
8         for row in reader:
9             print(row)
10
11 if __name__ == '__main__':
12     process_csv('books.csv')
```

Here you import `csv` and create a function called `process_csv()`, which accepts the path to the CSV file as its sole argument. Then you open that file and pass it to `csv.reader()` to create a `reader` object. You can then iterate over this object line-by-line and print it out.

Here is the output you will receive when you run the code:

```
1 ['book_title', 'author', 'publisher', 'pub_date', 'isbn']
2 ['Python 101', 'Mike Driscoll', ' Mike Driscoll', '2020', '123456789']
3 ['wxPython Recipes', 'Mike Driscoll', 'Apress', '2018', '978-1-4842-3237-8']
4 ['Python Interviews', 'Mike Driscoll', 'Packt Publishing', '2018', '9781788399081']
```

Most of the time, you probably won't need to process the header row. You can skip that row by updating your code like this:

```
1 # csv_reader_no_header.py
2
3 import csv
4
5 def process_csv(path):
6     with open(path) as csvfile:
7         reader = csv.reader(csvfile)
8         # Skip the header
9         next(reader, None)
10        for row in reader:
11            print(row)
12
13 if __name__ == '__main__':
14     process_csv('books.csv')
```

Python's `next()` function will take an iterable, such as `reader`, and return the next item from the iterable. This will, in effect, skip the first row. If you run this code, you will see that the output is now missing the header row:

```
1 ['Python 101', 'Mike Driscoll', 'Mike Driscoll', '2020', '123456789']
2 ['wxPython Recipes', 'Mike Driscoll', 'Apress', '2018', '978-1-4842-3237-8']
3 ['Python Interviews', 'Mike Driscoll', 'Packt Publishing', '2018', '9781788399081']
```

The `csv.reader()` function takes in some other optional arguments that are quite useful. For example, you might have a file that uses a delimiter other than a comma. You can use the `delimiter` argument to tell the `csv` module to parse the file based on that information.

Here is an example of how you might parse a file that uses a colon as its delimiter:

```
1 reader = csv.reader(csvfile, delimiter=':')
```

You should try creating a few variations of the original data file and then read them in using the `delimiter` argument.

Let's learn about another way to read CSV files!

Reading a CSV File with DictReader

The `csv` module provides a second “reader” object you can use called the `DictReader` class. The nice thing about the `DictReader` is that when you iterate over it, each row is returned as a Python dictionary. Go ahead and create a new file named `csv_dict_reader.py` and enter the following code:

```
1 # csv_dict_reader.py
2
3 import csv
4
5 def process_csv_dict_reader(file_obj):
6     reader = csv.DictReader(file_obj)
7     for line in reader:
8         print(f'{line["book_title"]} by {line["author"]}')
9
10 if __name__ == '__main__':
11     with open('books.csv') as csvfile:
12         process_csv_dict_reader(csvfile)
```

In this code you create a `process_csv_dict_reader()` function that takes in a file object rather than a file path. Then you convert the file object into a Python dictionary using `DictReader()`. Next, you loop over the `reader` object and print out a couple fields from each record using Python's dictionary access syntax.

You can see the output from running this code below:

```
1 Python 101 by Mike Driscoll
2 wxPython Recipes by Mike Driscoll
3 Python Interviews by Mike Driscoll
```

`csv.DictReader()` makes accessing fields within records much more intuitive than the regular `csv.reader` object. Try using it on one of your own CSV files to gain additional practice.

Now, you will learn how to write a CSV file using Python's `csv` module!

Writing a CSV File

Python's `csv` module wouldn't be complete without some way to create a CSV file. In fact, Python has two ways. Let's start by looking at the first method below. Go ahead and create a new file named `csv_writer.py` and enter the following code:

```
1 # csv_writer.py
2
3 import csv
4
5 def csv_writer(path, data):
6     with open(path, 'w') as csvfile:
7         writer = csv.writer(csvfile, delimiter=',')
8         for row in data:
9             writer.writerow(row)
10
11 if __name__ == '__main__':
12     data = '''book_title,author,publisher,pub_date,isbn
13 Python 101,Mike Driscoll, Mike Driscoll,2020,123456789
14 wxPython Recipes,Mike Driscoll,Apress,2018,978-1-4842-3237-8
15 Python Interviews,Mike Driscoll,Packt Publishing,2018,9781788399081'''
16     records = []
17     for line in data.splitlines():
18         records.append(line.strip().split(','))
19     csv_writer('output.csv', records)
```

In this code, you create a `csv_writer()` function that takes two arguments:

- The path to the CSV file that you want to create
- The data that you want to write to the file

To write data to a file, you need to create a `writer()` object. You can set the delimiter to something other than commas if you want to, but to keep things consistent, this example explicitly sets it to a comma. When you are ready to write data to the `writer()`, you will use `writerow()`, which takes in a list of strings.

The code that is outside of the `csv_writer()` function takes a multiline string and transforms it into a list of lists for you.

If you would like to write all the rows in the list at once, you can use the `writerows()` function. Here is an example for that:

```
1 # csv_writer_rows.py
2
3 import csv
4
5 def csv_writer(path, data):
6     with open(path, 'w') as csvfile:
7         writer = csv.writer(csvfile, delimiter=',')
8         writer.writerows(data)
9
10 if __name__ == '__main__':
11     data = '''book_title,author,publisher,pub_date,isbn
12 Python 101,Mike Driscoll, Mike Driscoll,2020,123456789
13 wxPython Recipes,Mike Driscoll,Apress,2018,978-1-4842-3237-8
14 Python Interviews,Mike Driscoll,Packt Publishing,2018,9781788399081'''
15     records = []
16     for line in data.splitlines():
17         records.append(line.strip().split(','))
18     csv_writer('output2.csv', records)
```

Instead of looping over the data row by row, you can write the entire list of lists to the file all at once.

This was the first method of creating a CSV file. Now let's learn about the second method: the `DictWriter`!

Writing a CSV File with `DictWriter`

The `DictWriter` is the complement class of the `DictReader`. It works in a similar manner as well. To learn how to use it, create a file named `csv_dict_writer.py` and enter the following:

```
1 # csv_dict_writer.py
2
3 import csv
4
5 def csv_dict_writer(path, headers, data):
6     with open(path, 'w') as csvfile:
7         writer = csv.DictWriter(
8             csvfile,
9             delimiter=',',
10            fieldnames=headers,
11            )
12         writer.writeheader()
```

```
13     for record in data:
14         writer.writerow(record)
15
16 if __name__ == '__main__':
17     data = '''book_title,author,publisher,pub_date,isbn
18 Python 101,Mike Driscoll, Mike Driscoll,2020,123456789
19 wxPython Recipes,Mike Driscoll,Apress,2018,978-1-4842-3237-8
20 Python Interviews,Mike Driscoll,Packt Publishing,2018,9781788399081'''
21 records = []
22 for line in data.splitlines():
23     records.append(line.strip().split(','))
24 headers = records.pop(0)
25
26 list_of_dicts = []
27 for row in records:
28     my_dict = dict(zip(headers, row))
29     list_of_dicts.append(my_dict)
30
31 csv_dict_writer('output_dict.csv', headers, list_of_dicts)
```

In this example, you pass in three arguments to `csv_dict_writer()`:

- The path to the file that you are creating
- The header row (a list of strings)
- The `data` argument as a Python list of dictionaries

When you instantiate `DictWriter()`, you give it a file object, set the delimiter, and, using the `headers` parameter, tell it what the `fieldnames` are. Next, you call `writeheader()` to write that header to the file. Finally, you loop over the data as you did before and use `writerow()` to write each record to the file. However, the record is now a dictionary instead of a list.

The code outside the `csv_dict_writer()` function is used to create the pieces you need to feed to the function. Once again, you create a list of lists, but this time you extract the first row and save it off in `headers`. Then you loop over the rest of the records and turn them into a list of dictionaries.

Wrapping Up

Python's `csv` module is great! You can read and write CSV files with very few lines of code. In this chapter you learned how to do that in the following sections:

- Reading a CSV File
- Reading a CSV File with `DictReader`

- Writing a CSV File
- Writing a CSV File with `DictWriter`

There are other ways to work with CSV files in Python. One popular method is to use the `pandas` package. Pandas is primarily used for data analysis and data science, so using it for working with CSVs seems like using a sledge hammer on a nail. Python's `csv` module is quite capable all on its own. But you are welcome to check out pandas and see how it might work for this use-case. You can read more about that project here:

- <https://pandas.pydata.org/>

If you don't work as a data scientist, you probably won't be using pandas. In that case, Python's `csv` module works fine. Go ahead and put in some more practice with Python's `csv` module to see how nice it is to work with!

Review Questions

1. How do you read a CSV file with Python's standard library?
2. If your CSV file doesn't use commas as the delimiter, how do you use the `csv` module to read it?
3. How do you write a row of CSV data using the `csv` module?

Chapter 37 - How to Work with a Database Using `sqlite3`

Software developers have to work with data. More often than not, the data that you work with will need to be available to multiple developers as well as multiple users at once. The typical solution for this type of situation is to use a database. Databases hold data in a tabular format, which means that they have labeled columns and rows of data.

Most database software require you to install complex software on your local machine or on a server you have access to. Popular database software includes Microsoft SQL Server, PostgreSQL, and MySQL, among others. For the purposes of this chapter, you will focus on a very simple one known as **SQLite**. The reason you will use SQLite is that it is a file-based database system that is included with Python. You won't need to do any configuration or additional installation. This allows you to focus on the essentials of what a database is and how it functions, while avoiding the danger of getting lost in installation and setup details.

In this chapter, you will learn about the following:

- Creating a SQLite Database
- Adding Data to Your Database
- Searching Your Database
- Editing Data in Your Database
- Deleting Data From Your Database

Let's start learning about how to use Python with a database now!

Creating a SQLite Database

There are 3rd party SQL connector packages to help you connect your Python code to all major databases. The Python standard library already comes with a `sqlite3` library built-in, which is what you will be using. This means that you won't have to install anything extra in order to work through this chapter. You can read the documentation for the `sqlite3` library here:

- <https://docs.python.org/3/library/sqlite3.html>

To start working with a database, you need to either connect to a pre-existing one or create a new one. For the purposes of this chapter, you will create a database. However, you will learn enough in this chapter to also load and interact with a pre-existing database if you want to.

SQLite supports the following types of data:

- NULL
- INTEGER
- REAL
- TEXT
- BLOB

These are the data types that you can store in this type of database. If you want to read more about how Python data types translate to SQLite data types and vice-versa, see the following link:

- <https://docs.python.org/3/library/sqlite3.html#sqlite-and-python-types>

Now it is time for you to create a database! Here is how you would create a SQLite database with Python:

```
1 import sqlite3
2
3 sqlite3.connect("library.db")
```

First, you import `sqlite3` and then you use the `connect()` function, which takes the path to the database file as an argument. If the file does not exist, the `sqlite3` module will create an empty database. Once the database file has been created, you need to add a table to be able to work with it. The basic SQL command you use for doing this is as follows:

```
1 CREATE TABLE table_name
2 (column_one TEXT, column_two TEXT, column_three TEXT)
```



Keywords in SQL are case-insensitive – so `CREATE` == Create == create. Identifiers, however, might be case-sensitive – it depends on the SQL engine being used and possibly what configuration settings are being used by that engine or by the database. If using a preexisting database, either check its documentation or just use the same case as it uses for table and field names.

You will be following the convention of KEYWORDS in UPPER-case, and identifiers in Mixed- or lower-case.

The `CREATE TABLE` command will create a table using the name specified. You follow that command with the name of each column as well as the column type. Columns can also be thought of as fields and column types as field types. The SQL code snippet above creates a three-column table where all the columns contain text. If you call this command and the table already exists in the database, you will receive an error.

You can create as many tables as the database allows. The number of rows and columns may have a limit from the database software, but most of the time you won't run into this limit.

If you combine the information you learned in the last two examples, you can create a database for storing information about books. Create a new file named `create_database.py` and enter the following code:

```
1 # create_database.py
2
3 import sqlite3
4
5 conn = sqlite3.connect("library.db")
6
7 cursor = conn.cursor()
8
9 # create a table
10 cursor.execute("""CREATE TABLE books
11             (title TEXT, author TEXT, release_date TEXT,
12              publisher TEXT, book_type TEXT)
13            """)
```

To work with a SQLite database, you need to `connect()` to it and then create a `cursor()` object from that connection. The `cursor` is what you use to send SQL commands to your database via its `execute()` function. The last line of code above will use the SQL syntax you saw earlier to create a `books` table with five fields:

- `title` - The title of the book as text
- `author` - The author of the book as text
- `release_date` - The date the book was released as text
- `publisher` - The publisher of the book as text
- `book_type` - The type of book (print, epub, PDF, etc)

Now you have a database that you can use, but it has no data. You will discover how to add data to your table in the next section!

Adding Data to Your Database

Adding data to a database is done using the `INSERT INTO` SQL commands. You use this command in combination with the name of the table that you wish to insert data into. This process will become clearer by looking at some code, so go ahead and create a file named `add_data.py`. Then add this code to it:

```
1 # add_data.py
2
3 import sqlite3
4
5 conn = sqlite3.connect("library.db")
6 cursor = conn.cursor()
7
8 # insert a record into the books table in the library database
9 cursor.execute("""INSERT INTO books
10                 VALUES ('Python 101', 'Mike Driscoll', '9/01/2020',
11                         'Mouse Vs Python', 'epub')"""
12             )
13
14 # save data
15 conn.commit()
16
17 # insert multiple records using the more secure "?" method
18 books = [('Python Interviews', 'Mike Driscoll',
19             '2/1/2018', 'Packt Publishing', 'softcover'),
20             ('Automate the Boring Stuff with Python',
21                 'Al Sweigart', '', 'No Starch Press', 'PDF'),
22             ('The Well-Grounded Python Developer',
23                 'Doug Farrell', '2020', 'Manning', 'Kindle')]
24 cursor.executemany("INSERT INTO books VALUES (?,?,?,?,?)", books)
25 conn.commit()
```

The first six lines show how to connect to the database and create the cursor as before. Then you use `execute()` to call `INSERT INTO` and pass it a series of five `VALUES`. To save that record to the database table, you need to call `commit()`.

The last few lines of code show how to commit multiple records to the database at once using `executemany()`. You pass `executemany()` a SQL statement and a list of items to use with that SQL statement. While there are other ways of inserting data, using the “?” syntax as you did in the example above is the preferred way of passing values to the cursor as it prevents SQL injection attacks.

If you'd like to learn more about SQL Injection, Wikipedia is a good place to start:

- https://en.wikipedia.org/wiki/SQL_injection

Now you have data in your table, but you don't have a way to actually view that data. You will find out how to do that next!

Searching Your Database

Extracting data from a database is done primarily with the SELECT, FROM, and WHERE keywords. You will find that these commands are not too hard to use. You should create a new file named queries.py and enter the following code into it:

```
1 import sqlite3
2
3 def get_cursor():
4     conn = sqlite3.connect("library.db")
5     return conn.cursor()
6
7 def select_all_records_by_author(cursor, author):
8     sql = "SELECT * FROM books WHERE author=?"
9     cursor.execute(sql, [author])
10    print(cursor.fetchall()) # or use fetchone()
11    print("\nHere is a listing of the rows in the table\n")
12    for row in cursor.execute("SELECT rowid, * FROM books ORDER BY author"):
13        print(row)
14
15 def select_using_like(cursor, text):
16     print("\nLIKE query results:\n")
17     sql = f"""
18         SELECT * FROM books
19         WHERE title LIKE '{text}%''''
20     cursor.execute(sql)
21     print(cursor.fetchall())
22
23 if __name__ == '__main__':
24     cursor = get_cursor()
25     select_all_records_by_author(cursor,
26                                  author='Mike Driscoll')
27     select_using_like(cursor, text='Python')
```

This code is a little long, so we will go over each function individually. Here is the first bit of code:

```
1 import sqlite3
2
3 def get_cursor():
4     conn = sqlite3.connect("library.db")
5     return conn.cursor()
```

The `get_cursor()` function is a useful function for connecting to the database and returning the `cursor` object. You could make it more generic by passing it the name of the database you wish to open.

The next function will show you how to get all the records for a particular author in the database table:

```

1 def select_all_records_by_author(cursor, author):
2     sql = "SELECT * FROM books WHERE author=?"
3     cursor.execute(sql, [author])
4     print(cursor.fetchall()) # or use fetchone()
5     print("\nHere is a listing of the rows in the table\n")
6     for row in cursor.execute("SELECT rowid, * FROM books ORDER BY author"):
7         print(row)

```

To get all the records from a database, you would use the following SQL command: `SELECT * FROM books`. `SELECT`, by default, returns the requested fields from every record in the database table. The asterisk is a wildcard character which means “I want all the fields”. So `SELECT` and `*` combined will return all the data currently in a table. You usually do not want to do that! Tables can become quite large and trying to pull everything from it at once may adversely affect your database’s, or your computer’s, performance. Instead, you can use the `WHERE` clause to filter the `SELECT` to something more specific, and/or only select the fields you are interested in.

In this example, you filter the `SELECT` to a specific author. You are still selecting all the records, but it is unlikely for a single author to have contributed to too many rows to negatively affect performance. You then tell the cursor to `fetchall()`, which will fetch all the results from the `SELECT` call you made. You could use `fetchone()` to fetch only the first result from the `SELECT`.

The last two lines of code fetch all the entries in the `books` table along with their `rowids`, and orders the results by the author name. The output from this function looks like this:

```

1 Here is a listing of the rows in the table
2
3 (3, 'Automate the Boring Stuff with Python', 'Al Sweigart', '', 'No Starch Press', '\
4 PDF')
5 (4, 'The Well-Grounded Python Developer', 'Doug Farrell', '2020', 'Manning', 'Kindle\
6 ')
7 (1, 'Python 101', 'Mike Driscoll', '9/01/2020', 'Mouse Vs Python', 'epub')
8 (2, 'Python Interviews', 'Mike Driscoll', '2/1/2018', 'Packt Publishing', 'softcover\
9 ')

```

You can see that when you sort by `author`, it sorts using the entire string rather than by the last name. If you are looking for a challenge, you can try to figure out how you might store the data to make it possible to sort by the last name. Alternatively, you could write more complex SQL queries or process the results in Python to sort it in a nicer way.

The last function to look at is `select_using_like()`:

```

1 def select_using_like(cursor, text):
2     print("\nLIKE query results:\n")
3     sql = f"""
4         SELECT * FROM books
5         WHERE title LIKE '{text}%''''
6     cursor.execute(sql)
7     print(cursor.fetchall())

```

This function demonstrates how to use the SQL command `LIKE`, which is kind of a filtered wildcard search. In this example, you tell it to look for a specific string with a percent sign following it. The percent sign is a wildcard, so it will look for any record that has a title that starts with the passed-in string.

When you run this function with the `text` set to “Python”, you will see the following output:

```

1 LIKE query results:
2
3 [('Python 101', 'Mike Driscoll', '9/01/2020', 'Mouse Vs Python', 'epub'),
4 ('Python Interviews', 'Mike Driscoll', '2/1/2018', 'Packt Publishing', 'softcover')]

```

The last few lines of code are here to demonstrate what the functions do:

```

1 if __name__ == '__main__':
2     cursor = get_cursor()
3     select_all_records_by_author(cursor,
4                                   author='Mike Driscoll')
5     select_using_like(cursor, text='Python')

```

Here you grab the `cursor` object and pass it in to the other functions. Remember, you use the `cursor` to send commands to your database. In this example, you set the `author` for `select_all_records_by_author()` and the `text` for `select_using_like()`. These functions are a good way to make your code reusable.

Now you are ready to learn how to update data in your database!

Editing Data in Your Database

When it comes to editing data in a database, you will almost always be using the following SQL commands:

- UPDATE - Used for updating a specific database table
- SET - Used to update a specific field in the database table



UPDATE, just like SELECT, works on all records in a table by default. Remember to use WHERE to limit the scope of the command!

To see how this works, create a file named `update_record.py` and add this code:

```
1 # update_record.py
2
3 import sqlite3
4
5 def update_author(old_name, new_name):
6     conn = sqlite3.connect("library.db")
7     cursor = conn.cursor()
8     sql = f"""
9         UPDATE books
10        SET author = '{new_name}'
11       WHERE author = '{old_name}'
12        """
13     cursor.execute(sql)
14     conn.commit()
15
16 if __name__ == '__main__':
17     update_author(
18         old_name='Mike Driscoll',
19         new_name='Michael Driscoll',
20     )
```

In this example, you create `update_author()` which takes in the old author name to look for and the new author name to change it to. Then you connect to the database and create the cursor as you have in the previous examples. The SQL code here tells your database that you want to update the `books` table and set the `author` field to the new name where the `author` name currently equals the old name. Finally, you `execute()` and `commit()` the changes.

To test that this code worked, you can re-run the query code from the previous section and examine the output.

Now you're ready to learn how to delete data from your database!

Deleting Data From Your Database

Sometimes data must be removed from a database. For example, if you decide to stop being a customer at a bank, you would expect them to purge your information from their database after a certain period of time had elapsed. To delete from a database, you can use the `DELETE` command.

Go ahead and create a new file named `delete_record.py` and add the following code to see how deleting data works:

```
1 # delete_record.py
2
3 import sqlite3
4
5 def delete_author(author):
6     conn = sqlite3.connect("library.db")
7     cursor = conn.cursor()
8
9     sql = f"""
10        DELETE FROM books
11        WHERE author = '{author}'
12    """
13
14     cursor.execute(sql)
15     conn.commit()
16
17 if __name__ == '__main__':
18     delete_author(author='Al Sweigart')
```

Here you create `delete_author()` which takes in the name of the author that you wish to remove from the database. The code in this example is nearly identical to the previous example except for the SQL statement itself. In the SQL query, you use `DELETE FROM` to tell the database which table to delete data from. Then you use the `WHERE` clause to tell it which field to use to select the target records. In this case, you tell your database to remove any records from the `books` table that match the `author` name.

You can verify that this code worked using the SQL query code from earlier in this chapter.

Wrapping Up

Working with databases can be a lot of work. This chapter covered only the basics of working with databases. Here you learned how to do the following:

- Creating a SQLite Database

- Adding Data to Your Database
- Searching Your Database
- Editing Data in Your Database
- Deleting Data From Your Database

If you find SQL code a bit difficult to understand, you might want to check out an “object relational mapper” package, such as **SQLAlchemy** or **SQLObject**. An object relational mapper (ORM) turns Python statements into SQL code for you so that you are only writing Python code. Sometimes you may still need to drop down to bare SQL to get the efficiency you need from the database, but these ORMs can help speed up development and make things easier.

Here are some links for those two projects:

- SQLAlchemy - <https://www.sqlalchemy.org/>
- SQLObject - <http://sqlobject.org/>

Review Questions

1. How do you create a database with the `sqlite3` library?
2. Which SQL command is used to add data to a table?
3. How do you change a field in a database with SQL?
4. What are SQL queries used for?
5. By default, how many records in a table will `DELETE` affect? How about `UPDATE` and `SELECT`?
6. The `delete_author` function above is susceptible to an SQL Injection attack. Why, and how would you fix it?

Chapter 38 - Working with an Excel Document in Python

The business world uses **Microsoft Office**. Their spreadsheet software solution, **Microsoft Excel**, is especially popular. Excel is used to store tabular data, create reports, graph trends, and much more. Before diving into working with Excel with Python, let's clarify some special terminology:

- Spreadsheet or Workbook - The file itself (.xls or .xlsx).
- Worksheet or Sheet - A single sheet of content within a Workbook. Spreadsheets can contain multiple Worksheets.
- Column - A vertical line of data that is labeled with letters, starting with "A".
- Row - A horizontal line of data labeled with numbers, starting with 1.
- Cell - A combination of Column and Row, like "A1".

In this chapter, you will be using Python to work with Excel Spreadsheets. You will learn about the following:

- Python Excel Packages
- Getting Sheets from a Workbook
- Reading Cell Data
- Iterating Over Rows and Columns
- Writing Excel Spreadsheets
- Adding and Removing Sheets
- Adding and Deleting Rows and Columns

Excel is used by most companies and universities. It can be used in many different ways and enhanced using Visual Basic for Applications (VBA). However, VBA is kind of clunky – which is why it's good to learn how to use Excel with Python.

Let's find out how to work with Microsoft Excel spreadsheets using the Python programming language now!

Python Excel Packages

You can use Python to create, read and write Excel spreadsheets. However, Python's standard library does not have support for working with Excel; to do so, you will need to install a 3rd party package. The most popular one is **OpenPyXL**. You can read its documentation [here](#):

- <https://openpyxl.readthedocs.io/en/stable/>

OpenPyXL is not your only choice. There are several other packages that support Microsoft Excel:

- xlrd - For reading older Excel (.xls) documents
- xlwt - For writing older Excel (.xls) documents
- xlwings - Works with new Excel formats and has macro capabilities

A couple years ago, the first two used to be the most popular libraries to use with Excel documents. However, the author of those packages has stopped supporting them. The xlwings package has lots of promise, but does not work on all platforms and requires that Microsoft Excel is installed.

You will be using OpenPyXL in this chapter because it is actively developed and supported. OpenPyXL doesn't require Microsoft Excel to be installed, and it works on all platforms.

You can install OpenPyXL using pip:

```
1 $ python -m pip install openpyxl
```

After the installation has completed, let's find out how to use OpenPyXL to read an Excel spreadsheet!

Getting Sheets from a Workbook

The first step is to find an Excel file to use with OpenPyXL. There is a books.xlsx file that is provided for you in this book's Github repository. You can download it by going to this URL:

- https://github.com/driscollis/python101code/tree/master/chapter38_excel

Feel free to use your own file, although the output from your own file won't match the sample output in this book.

The next step is to write some code to open the spreadsheet. To do that, create a new file named open_workbook.py and add this code to it:

```
1 # open_workbook.py
2
3 from openpyxl import load_workbook
4
5 def open_workbook(path):
6     workbook = load_workbook(filename=path)
7     print(f'Worksheet names: {workbook.sheetnames}')
8     sheet = workbook.active
9     print(sheet)
10    print(f'The title of the Worksheet is: {sheet.title}')
11
12 if __name__ == '__main__':
13     open_workbook('books.xlsx')
```

In this example, you import `load_workbook()` from `openpyxl` and then create `open_workbook()` which takes in the path to your Excel spreadsheet. Next, you use `load_workbook()` to create an `openpyxl.workbook.Workbook` object. This object allows you to access the sheets and cells in your spreadsheet. And yes, it really does have the double `workbook` in its name. That's not a typo!

The rest of the `open_workbook()` function demonstrates how to print out all the currently defined sheets in your spreadsheet, get the currently active sheet and print out the title of that sheet.

When you run this code, you will see the following output:

```
1 Worksheet names: ['Sheet 1 - Books']
2 <Worksheet "Sheet 1 - Books">
3 The title of the Worksheet is: Sheet 1 - Books
```

Now that you know how to access the sheets in the spreadsheet, you are ready to move on to accessing cell data!

Reading Cell Data

When you are working with Microsoft Excel, the data is stored in cells. You need a way to access those cells from Python to be able to extract that data. OpenPyXL makes this process straightforward.

Create a new file named `workbook_cells.py` and add this code to it:

```
1 # workbook_cells.py
2
3 from openpyxl import load_workbook
4
5 def get_cell_info(path):
6     workbook = load_workbook(filename=path)
7     sheet = workbook.active
8     print(sheet)
9     print(f'The title of the Worksheet is: {sheet.title}')
10    print(f'The value of {sheet["A2"].value}')
11    print(f'The value of {sheet["A3"].value}')
12    cell = sheet['B3']
13    print(f'{cell.value}')
14
15 if __name__ == '__main__':
16     get_cell_info('books.xlsx')
```

This code will load up the Excel file in an OpenPyXL workbook. You will grab the active sheet and then print out its title and a couple of different cell values. You can access a cell by using the sheet object followed by square brackets with the column name and row number inside of it. For example, sheet["A2"] will get you the cell at column “A”, row 2. To get the value of that cell, you use the value attribute.

Note: This code is using a new feature that was added to f-strings in Python 3.8. If you run this with an earlier version, you will receive an error.

When you run this code, you will get this output:

```
1 <Worksheet "Sheet 1 - Books">
2 The title of the Worksheet is: Sheet 1 - Books
3 The value of sheet["A2"].value='Title'
4 The value of sheet["A3"].value='Python 101'
5 cell.value='Mike Driscoll'
```

You can get additional information about a cell using some of its other attributes. Add the following function to your file and update the conditional statement at the end to run it:

```
1 def get_info_by_coord(path):
2     workbook = load_workbook(filename=path)
3     sheet = workbook.active
4     cell = sheet['A2']
5     print(f'Row {cell.row}, Col {cell.column} = {cell.value}')
6     print(f'{cell.value} is at {cell.coordinate}')
7
8 if __name__ == '__main__':
9     get_info_by_coord('books.xlsx')
```

In this example, you use the `row` and `column` attributes of the `cell` object to get the row and column information. Note that column “A” maps to “1”, “B” to “2”, etcetera. If you were to iterate over the Excel document, you could use the `coordinate` attribute to get the cell name.

When you run this code, the output will look like this:

```
1 Row 2, Col 1 = Title
2 cell.value='Title' is at cell.coordinate='A2'
```

Speaking of iterating, let’s find out how to do that next!

Iterating Over Rows and Columns

Sometimes you will need to iterate over the entire Excel spreadsheet or portions of the spreadsheet. OpenPyXL allows you to do that in a few different ways. Create a new file named `iterating_over_cells.py` and add the following code to it:

```
1 # iterating_over_cells.py
2
3 from openpyxl import load_workbook
4
5 def iterating_range(path):
6     workbook = load_workbook(filename=path)
7     sheet = workbook.active
8     for cell in sheet['A']:
9         print(cell)
10
11 if __name__ == '__main__':
12     iterating_range('books.xlsx')
```

Here you load up the spreadsheet and then loop over all the cells in column “A”. For each cell, you print out the `cell` object. You could use some of the cell attributes you learned about in the previous section if you wanted to format the output more granularly.

This what you get from running this code:

```
1 <Cell 'Sheet 1 - Books'.A1>
2 <Cell 'Sheet 1 - Books'.A2>
3 <Cell 'Sheet 1 - Books'.A3>
4 <Cell 'Sheet 1 - Books'.A4>
5 <Cell 'Sheet 1 - Books'.A5>
6 <Cell 'Sheet 1 - Books'.A6>
7 <Cell 'Sheet 1 - Books'.A7>
8 <Cell 'Sheet 1 - Books'.A8>
9 <Cell 'Sheet 1 - Books'.A9>
10 <Cell 'Sheet 1 - Books'.A10>
11 # output truncated for brevity
```

The output is truncated as it will print out quite a few cells by default. OpenPyXL provides other ways to iterate over rows and columns by using the `iter_rows()` and `iter_cols()` functions. These methods accept several arguments:

- `min_row`
- `max_row`
- `min_col`
- `max_col`

You can also add on a `values_only` argument that tells OpenPyXL to return the value of the cell instead of the cell object. Go ahead and create a new file named `iterating_over_cell_values.py` and add this code to it:

```
1 # iterating_over_cell_values.py
2
3 from openpyxl import load_workbook
4
5 def iterating_over_values(path):
6     workbook = load_workbook(filename=path)
7     sheet = workbook.active
8     for value in sheet.iter_rows(
9         min_row=1, max_row=3,
10        min_col=1, max_col=3,
11        values_only=True,
12    ):
13         print(value)
14
15 if __name__ == '__main__':
16     iterating_over_values('books.xlsx')
```

This code demonstrates how you can use the `iter_rows()` to iterate over the rows in the Excel spreadsheet and print out the values of those rows. When you run this code, you will get the following output:

```
1 ('Books', None, None)
2 ('Title', 'Author', 'Publisher')
3 ('Python 101', 'Mike Driscoll', 'Mouse vs Python')
```

The output is a Python tuple that contains the data within each column. At this point you have learned how to open spreadsheets and read data – both from specific cells, as well as through iteration. You are now ready to learn how to use OpenPyXL to **create** Excel spreadsheets!

Writing Excel Spreadsheets

Creating an Excel spreadsheet using OpenPyXL doesn't take a lot of code. You can create a spreadsheet by using the `Workbook()` class. Go ahead and create a new file named `writing_hello.py` and add this code to it:

```
1 # writing_hello.py
2
3 from openpyxl import Workbook
4
5 def create_workbook(path):
6     workbook = Workbook()
7     sheet = workbook.active
8     sheet['A1'] = 'Hello'
9     sheet['A2'] = 'from'
10    sheet['A3'] = 'OpenPyXL'
11    workbook.save(path)
12
13 if __name__ == '__main__':
14     create_workbook('hello.xlsx')
```

Here you instantiate `Workbook()` and get the active sheet. Then you set the first three rows in column “A” to different strings. Finally, you call `save()` and pass it the path to save the new document to. Congratulations! You have just created an Excel spreadsheet with Python.

Let's discover how to add and remove sheets in your Workbook next!

Adding and Removing Sheets

Many people like to organize their data across multiple Worksheets within the Workbook. OpenPyXL supports the ability to add new sheets to a `Workbook()` object via its `create_sheet()` method.

Create a new file named `creating_sheets.py` and add this code to it:

```
1 # creating_sheets.py
2
3 import openpyxl
4
5 def create_worksheets(path):
6     workbook = openpyxl.Workbook()
7     print(workbook.sheetnames)
8     # Add a new worksheet
9     workbook.create_sheet()
10    print(workbook.sheetnames)
11    # Insert a worksheet
12    workbook.create_sheet(index=1,
13                           title='Second sheet')
14    print(workbook.sheetnames)
15    workbook.save(path)
16
17 if __name__ == '__main__':
18     create_worksheets('sheets.xlsx')
```

Here you use `create_sheet()` twice to add two new Worksheets to the Workbook. The second example shows you how to set the title of a sheet and at which index to insert the sheet. The argument `index=1` means that the worksheet will be added after the first existing worksheet, since they are indexed starting at `0`.

When you run this code, you will see the following output:

```
1 ['Sheet']
2 ['Sheet', 'Sheet1']
3 ['Sheet', 'Second sheet', 'Sheet1']
```

You can see that the new sheets have been added step-by-step to your Workbook. After saving the file, you can verify that there are multiple Worksheets by opening Excel or another Excel-compatible application.

After this automated worksheet-creation process, you've suddenly got too many sheets, so let's get rid of some. There are two ways to remove a sheet. Go ahead and create `delete_sheets.py` to see how to use Python's `del` keyword for removing worksheets:

```
1 # delete_sheets.py
2
3 import openpyxl
4
5 def create_worksheets(path):
6     workbook = openpyxl.Workbook()
7     workbook.create_sheet()
8     # Insert a worksheet
9     workbook.create_sheet(index=1,
10                           title='Second sheet')
11    print(workbook.sheetnames)
12    del workbook['Second sheet']
13    print(workbook.sheetnames)
14    workbook.save(path)
15
16 if __name__ == '__main__':
17     create_worksheets('del_sheets.xlsx')
```

This code will create a new Workbook and then add two new Worksheets to it. Then it uses Python's `del` keyword to delete `workbook['Second sheet']`. You can verify that it worked as expected by looking at the print-out of the sheet list before and after the `del` command:

```
1 ['Sheet', 'Second sheet', 'Sheet1']
2 ['Sheet', 'Sheet1']
```

The other way to delete a sheet from a Workbook is to use the `remove()` method. Create a new file called `remove_sheets.py` and enter this code to learn how that works:

```
1 # remove_sheets.py
2
3 import openpyxl
4
5 def create_worksheets(path):
6     workbook = openpyxl.Workbook()
7     sheet1 = workbook.create_sheet()
8     # Insert a worksheet
9     workbook.create_sheet(index=1,
10                           title='Second sheet')
11    print(workbook.sheetnames)
12    workbook.remove(sheet1)
13    print(workbook.sheetnames)
14    workbook.save(path)
```

```
15  
16 if __name__ == '__main__':  
17     create_worksheets('remove_sheets.xlsx')
```

This time around, you hold onto a reference to the first Worksheet that you create by assigning the result to `sheet1`. Then you remove it later on in the code. Alternatively, you could also remove that sheet by using the same syntax as before, like this:

```
1 workbook.remove(workbook['Sheet1'])
```

No matter which method you choose for removing the Worksheet, the output will be the same:

```
1 ['Sheet', 'Second sheet', 'Sheet1']  
2 ['Sheet', 'Second sheet']
```

Now let's move on and learn how you can add and remove rows and columns.

Adding and Deleting Rows and Columns

OpenPyXL has several useful methods that you can use for adding and removing rows and columns in your spreadsheet. Here is a list of the four methods you will learn about in this section:

- `.insert_rows()`
- `.delete_rows()`
- `.insert_cols()`
- `.delete_cols()`

Each of these methods can take two arguments:

- `idx` - The index to insert the row or column
- `amount` - The number of rows or columns to add

To see how this works, create a file named `insert_demo.py` and add the following code to it:

```
1 # insert_demo.py
2
3 from openpyxl import Workbook
4
5 def inserting_cols_rows(path):
6     workbook = Workbook()
7     sheet = workbook.active
8     sheet['A1'] = 'Hello'
9     sheet['A2'] = 'from'
10    sheet['A3'] = 'OpenPyXL'
11    # insert a column before A
12    sheet.insert_cols(idx=1)
13    # insert 2 rows starting on the second row
14    sheet.insert_rows(idx=2, amount=2)
15    workbook.save(path)
16
17 if __name__ == '__main__':
18     inserting_cols_rows('inserting.xlsx')
```

Here you create a Worksheet and insert a new column before column “A”. Columns are indexed started at 1 while in contrast, worksheets start at 0. This effectively moves all the cells in column A to column B. Then you insert two new rows starting on row 2.

Now that you know how to insert columns and rows, it is time for you to discover how to remove them.

To find out how to remove columns or rows, create a new file named `delete_demo.py` and add this code:

```
1 # delete_demo.py
2
3 from openpyxl import Workbook
4
5 def deleting_cols_rows(path):
6     workbook = Workbook()
7     sheet = workbook.active
8     sheet['A1'] = 'Hello'
9     sheet['B1'] = 'from'
10    sheet['C1'] = 'OpenPyXL'
11    sheet['A2'] = 'row 2'
12    sheet['A3'] = 'row 3'
13    sheet['A4'] = 'row 4'
14    # Delete column A
15    sheet.delete_cols(idx=1)
```

```
16     # delete 2 rows starting on the second row
17     sheet.delete_rows(idx=2, amount=2)
18     workbook.save(path)
19
20 if __name__ == '__main__':
21     deleting_cols_rows('deleting.xlsx')
```

This code creates text in several cells and then removes column A using `delete_cols()`. It also removes two rows starting on the 2nd row via `delete_rows()`. Being able to add and remove columns and rows can be quite useful when it comes to organizing your data.

Wrapping Up

Due to the widespread use of Excel in many industries, it is an extremely useful skill to be able to interact with Excel files using Python. In this chapter, you learned about the following:

- Python Excel Packages
- Getting Sheets from a Workbook
- Reading Cell Data
- Iterating Over Rows and Columns
- Writing Excel Spreadsheets
- Adding and Removing Sheets
- Adding and Deleting Rows and Columns

OpenPyXL can do even more than what was covered here. For example, you can add formulas to cells, change fonts and apply other types of styling to cells using OpenPyXL. Read the documentation and try using OpenPyXL on some of your own spreadsheets so that you can discover its full power.

Review Questions

1. What Python package can you use to work with Microsoft Excel spreadsheets?
2. How do you open an Excel spreadsheet with Python?
3. Which class do you use to create an Excel spreadsheet with OpenPyXL?

Chapter 39 - How to Generate a PDF

The **Portable Document Format (PDF)** is a very popular way to share documents across multiple platforms. The goal of the PDF is to create a document that will look the same on multiple platforms and that will print the same (or very similar) on various printers. The format was originally developed by Adobe but has been made open-source.

Python has multiple libraries that you can use to create new PDFs or export portions of pre-existing PDFs. There are currently no Python libraries available for editing a PDF in-place. Here are a few of the packages you can use:

- **ReportLab** - used for creating PDFs
- **pdfrw** - used for splitting, merging, watermarking and rotating a PDF
- **PyPDF2 / PyPDF4** - used for splitting, merging, watermarking and rotating a PDF
- **PDFMiner** - used for extracting text from PDFs

There are many other PDF packages for Python. In this chapter, you will learn how to create a PDF using ReportLab. The ReportLab package has been around since the year 2000. It has an open-source version as well as a paid commercial version which has some extra features in it. You will be learning about the open-source version here.

In this chapter, you will learn about the following:

- Installing ReportLab
- Creating a Simple PDF with the Canvas
- Creating Drawings and Adding Images Using the Canvas
- Creating Multi-page Documents with PLATYPUS
- Creating a Table

ReportLab can generate almost any kind of report you can imagine. This chapter will not cover every feature that ReportLab has to offer, but you will learn enough about it to see how useful ReportLab can be.

Let's get started!

Installing ReportLab

You can install ReportLab using pip:

```
1 python -m pip install reportlab
```

ReportLab depends on the Pillow package, which is an image manipulation library for Python. It will be installed as well if you do not already have it on your system. Now that you have ReportLab installed, you are ready to learn how to create a simple PDF!

Creating a Simple PDF with the Canvas

There are two ways to create PDFs using the ReportLab package. The low-level method is drawing on the “canvas”. This allows you to draw at specific locations on the page. PDFs measure their size in **points** internally. There are 72 points per inch. A letter-size page is 612 x 792 points. However, the default page size is A4. There are several default page sizes that you can set or you can create your own page size.

It’s always easier to see some code so that you can understand how this will work. Create a new file named `hello_reportlab.py` and add this code:

```
1 # hello_reportlab.py
2
3 from reportlab.pdfgen import canvas
4
5 my_canvas = canvas.Canvas("hello.pdf")
6 my_canvas.drawString(100, 750, "Welcome to Reportlab!")
7 my_canvas.save()
```

This will create a PDF that is A4 sized. You create a `Canvas()` object that takes in the path to the PDF that you want to create. To add some text to the PDF, you use `drawString()`. This code tells ReportLab to start drawing the text 100 points from the left and 750 from the bottom of the page. If you were to start drawing at (0, 0), your text would appear at the bottom left of the page. You can change the location you start drawing by setting the `bottomup` canvas argument to 0.

The last line saves the PDF to disk. Don’t forget to do that or you won’t get to see your new creation!

The PDF should look something like this when you open it:

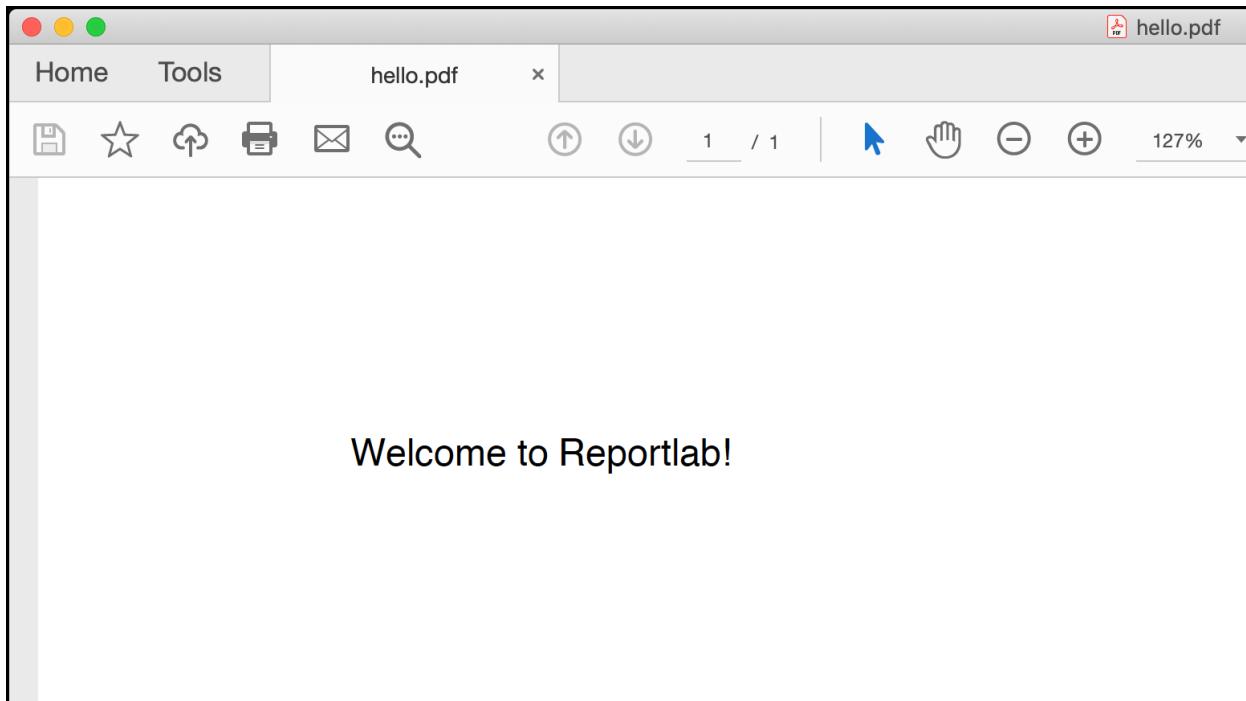


Fig. 39-1: Hello World on the Canvas

While this demonstrates how easy it is to create a PDF with ReportLab, it's kind of a boring example. You can use the canvas to draw lines, shapes and different fonts too. To learn how create a new file named `canvas_form.py` and enter this code in it:

```
1 # canvas_form.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.pdfgen import canvas
5
6 def form(path):
7     my_canvas = canvas.Canvas(path, pagesize=letter)
8     my_canvas.setLineWidth(.3)
9     my_canvas.setFont('Helvetica', 12)
10    my_canvas.drawString(30, 750, 'OFFICIAL COMMUNIQUE')
11    my_canvas.drawString(30, 735, 'OF ACME INDUSTRIES')
12    my_canvas.drawString(500, 750, "12/12/2010")
13    my_canvas.line(480, 747, 580, 747)
14    my_canvas.drawString(275, 725, 'AMOUNT OWED:')
15    my_canvas.drawString(500, 725, "$1,000.00")
16    my_canvas.line(378, 723, 580, 723)
17    my_canvas.drawString(30, 703, 'RECEIVED BY:')
18    my_canvas.line(120, 700, 580, 700)
19    my_canvas.drawString(120, 703, "JOHN DOE")
```

```
20     my_canvas.save()  
21  
22 if __name__ == '__main__':  
23     form('canvas_form.pdf')
```

Here you import the letter size from `reportlab.lib.pagesizes` which has several other sizes you could use. Then in the `form()` function, you set the `pagesize` when you instantiate `Canvas()`. Next, you use `setLineWidth()` to set the line width, which is used when you draw lines. Then you change the font to Helvetica with the font size at 12 points.

The rest of the code is a series of drawing strings at various locations with lines being drawn here and there. When you draw a `line()`, you pass in the starting coordinate (x/y positions) and the end coordinate (x/y position) and ReportLab will draw the line for you using the line width you set.

When you open the PDF, you will see the following:

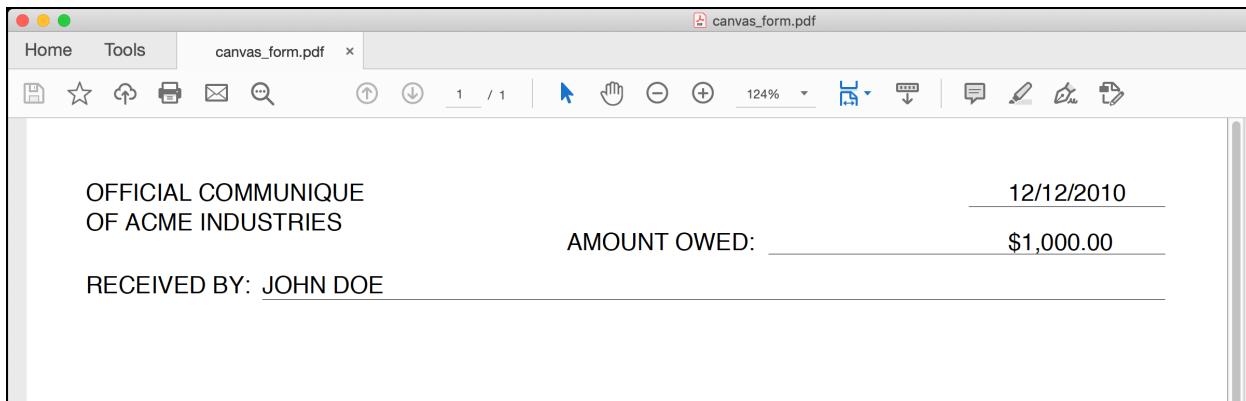


Fig. 39-2: Creating a form with ReportLab's Canvas

That looks pretty good. But what if you wanted to draw something or add a logo or some other photo to your report? Let's find out how to do that next!

Creating Drawings and Adding Images Using the Canvas

The ReportLab `Canvas()` is very flexible. It allows you to draw different shapes, use different colors, change the line widths, and more. To demonstrate some of these features, create a new file named `drawing_polygons.py` and add this code to it:

```
1 # drawing_polygons.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.pdfgen import canvas
5
6 def draw_shapes():
7     my_canvas = canvas.Canvas("drawing_polygons.pdf")
8     my_canvas.setStrokeColorRGB(0.2, 0.5, 0.3)
9     my_canvas.rect(10, 740, 100, 80, stroke=1, fill=0)
10    my_canvas.ellipse(10, 680, 100, 630, stroke=1, fill=1)
11    my_canvas.wedge(10, 600, 100, 550, 45, 90, stroke=1, fill=0)
12    my_canvas.circle(300, 600, 50)
13    my_canvas.save()
14
15 if __name__ == '__main__':
16     draw_shapes()
```

Here you create a `Canvas()` object as you have before. You can use `setStrokeColorRGB()` to change the border color using RGB values between zero and one. The next few lines of code create different shapes. For the `rect()` function, you specify the x and y start position which is the lower left-hand coordinate of the rectangle. Then you specify the width and height of the shape.

The `stroke` parameter tells ReportLab whether or not to draw the border while the `fill` parameter tells ReportLab whether or not to fill the shape with a color. All of the shapes support these two parameters.

According to the documentation for the `ellipse()`, it takes in the starting (x,y) and ending (x,y) coordinates for the enclosing rectangle for the ellipse shape.

The `wedge()` shape is similar in that you are once again specifying a series of points for an invisible rectangle that encloses the wedge shape. What you need to do is imagine that there is a circle inside of a rectangle and you are describing the size of the rectangle. The 5th argument is `startAng`, which is the starting angle of the wedge. The 6th argument is for the `extent`, which tells the wedge how far out the arc can extend.

Lastly, you create a `circle()`, which takes in the (x,y) coordinates of its center and then its radius. You skip setting the `stroke` and `fill` parameters.

When you run this code, you will end up with a PDF that looks like this:

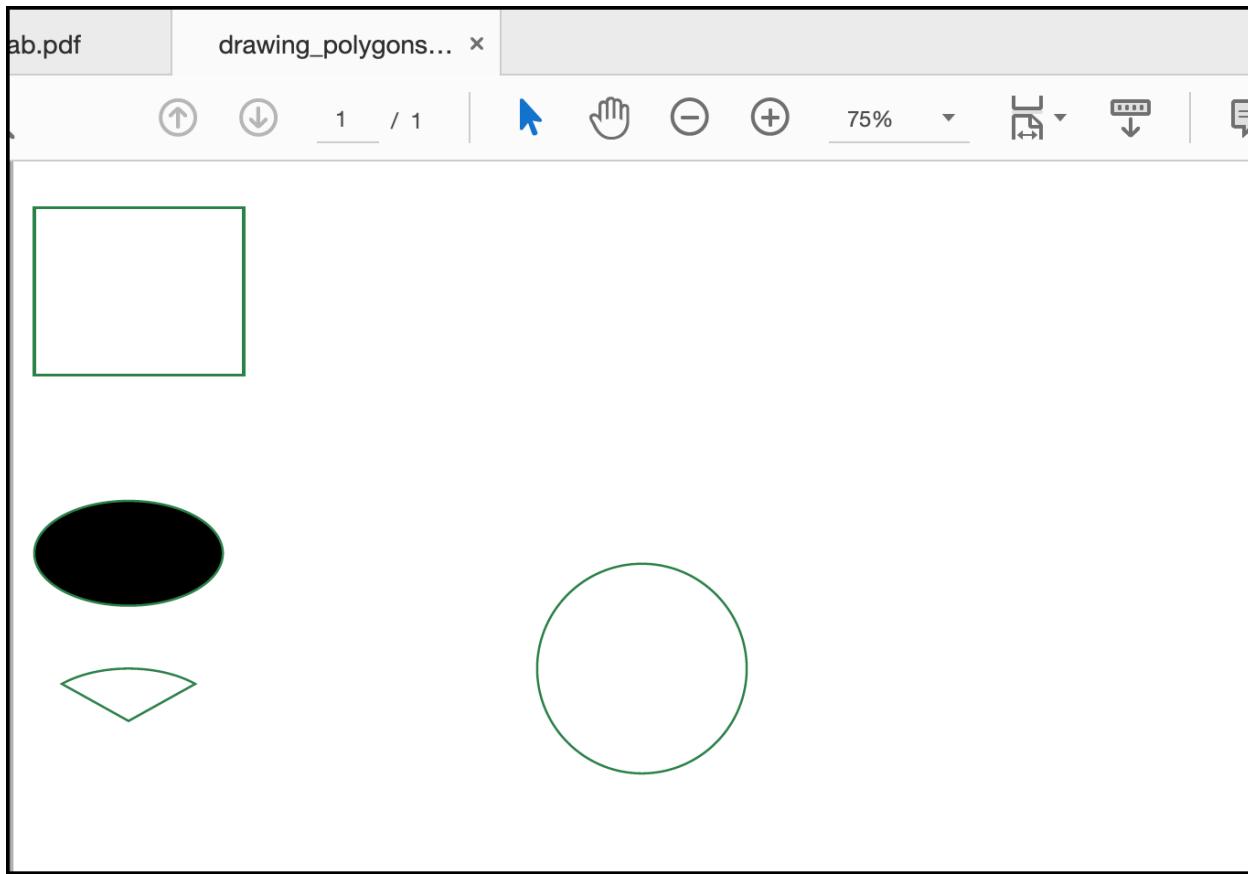


Fig. 39-3: Creating Polygons with ReportLab's Canvas

That looks pretty nice. You can play around with the values on your own and see if you can figure out how to change the shapes in different ways.

While these shapes you drew look fun, they wouldn't really look great on any professional company document. What if you want to add a company logo to your PDF report? You can do this with ReportLab by adding an image to your document. To discover how to do that, create a file named `image_on_canvas.py` and add this code to it:

```
1 # image_on_canvas.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.pdfgen import canvas
5
6
7 def add_image(image_path):
8     my_canvas = canvas.Canvas("canvas_image.pdf", pagesize=letter)
9     my_canvas.drawImage(image_path, 30, 600, width=100, height=100)
10    my_canvas.save()
11
```

```
12 if __name__ == '__main__':
13     image_path = 'snakehead.jpg'
14     add_image(image_path)
```

To draw an image on your canvas, you use the `drawImage()` method. It takes in the image path, the x and y starting positions, and the width and height you want to use for the image. This method does not maintain the aspect ratio of the image you pass in. If you set the width and height incorrectly, the image will be stretched.

When you run this code, you will end up with a PDF that looks something like this:

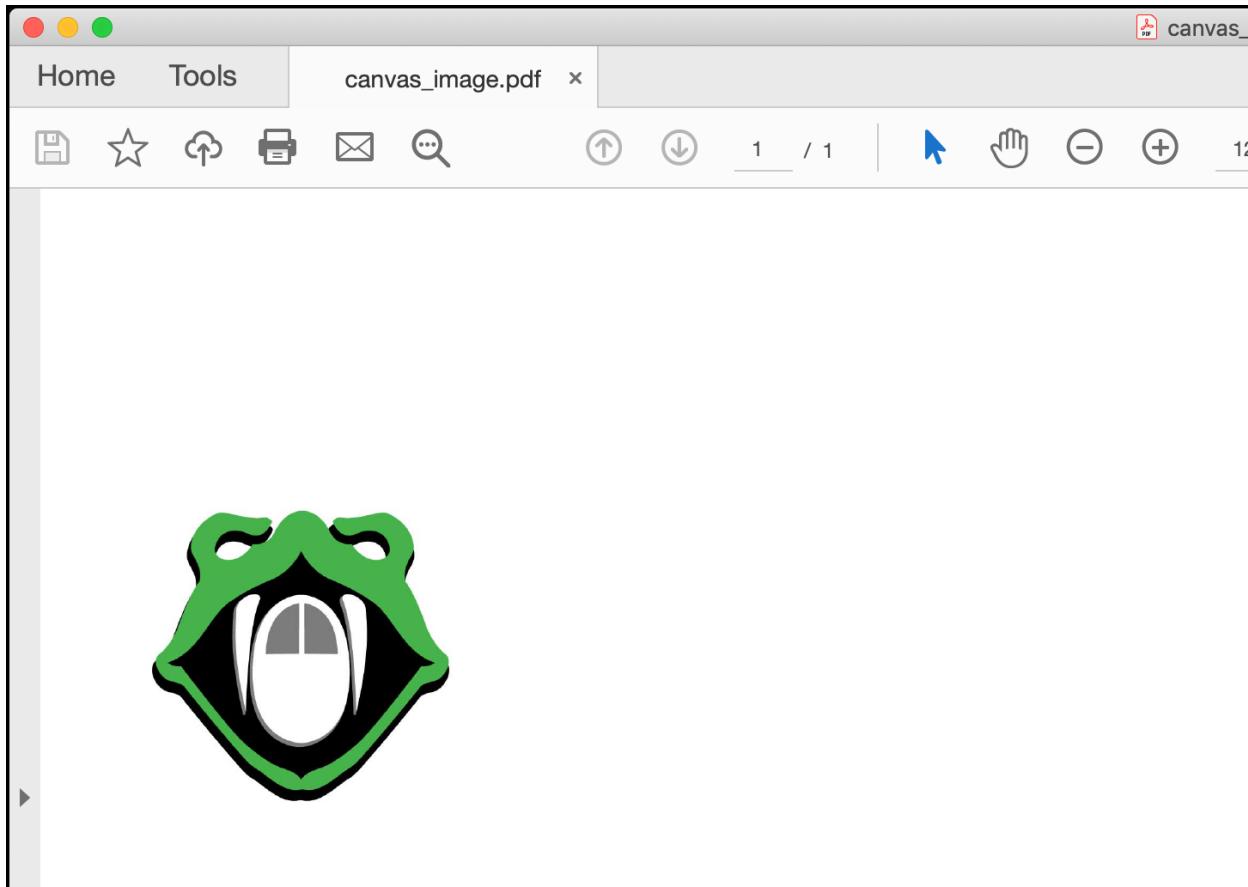


Fig. 39-4: Adding an image to ReportLab's Canvas

The `Canvas()` class is quite powerful. However, you need to keep track of where you are at on a page and tell the canvas when to create a new page. This can be difficult to do without making the code quite complex. Fortunately, there is a better way and you'll find out what that is in the next section!

Creating Multi-page Documents with PLATYPUS

ReportLab has a neat concept that they call **PLATYPUS**, which stands for “Page Layout and Typography Using Scripts”. It is a high-level layout library that ReportLab provides that makes it easier to programmatically create complex layouts with a minimum of code. PLATYPUS basically takes care of page breaking, layout, and styling for you.

There are several classes that you can use within PLATYPUS. These classes are known as **Flowables**. A Flowable has the ability to be added to a document and can break itself intelligently over multiple pages. The Flowables that you will use the most are:

- `Paragraph()` - for adding text
- `getSampleStyleSheet()` - for applying styles to Paragraphs
- `Table()` - for tabular data
- `SimpleDocTemplate()` - a document template used to hold other flowables

To see how these classes can be used, create a file named `hello_platypus.py` and add the following code:

```
1 # hello_platypus.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.platypus import SimpleDocTemplate, Paragraph
5 from reportlab.lib.styles import getSampleStyleSheet
6
7 def hello():
8     doc = SimpleDocTemplate(
9         "hello_platypus.pdf",
10        pagesize=letter,
11        rightMargin=72, leftMargin=72,
12        topMargin=72, bottomMargin=18,
13        )
14     styles = getSampleStyleSheet()
15
16     flowables = []
17
18     text = "Hello, I'm a Paragraph"
19     para = Paragraph(text, style=styles["Normal"])
20     flowables.append(para)
21
22     doc.build(flowables)
23
```

```
24 if __name__ == '__main__':
25     hello()
```

In this code, you import two new classes from `reportlab.platypus`: `SimpleDocTemplate()` and `Paragraph()`. You also import `getSampleStyleSheet()` from `reportlab.lib.styles`. Then in the `hello()` function, you create a document template object. This is where you pass the file path to the PDF that you want to create. It is analogous to the `Canvas()` class, but for PLATYPUS. You set the `pagesize` here as well and you also specify the margins. You aren't required to set up the margins, but you should know that you can, which is why it is shown here.

Then you get the sample style sheet. The `styles` variable is a `reportlab.lib.styles.StyleSheet1` object type. You can access several different styles in that stylesheet. For the purposes of this example, you use the `Normal` stylesheet.

This code creates a single `Flowable` using `Paragraph()`. The `Paragraph()` can take several different arguments. In this case, you pass in some text and what style you want to apply to the text. If you look at the code for the stylesheet, you see that you can apply various "Heading" styles to the text as well as a "Code" style and an "Italic" style, among others.

To make the document generate properly, you keep a Python list of the `Flowables`. In this example, you have a list with only one element in it: a `Paragraph()`. Instead of calling `save()` to create the PDF, you call `build()` and pass in the list of `Flowables`.

The PDF is now generated. It will look like this:

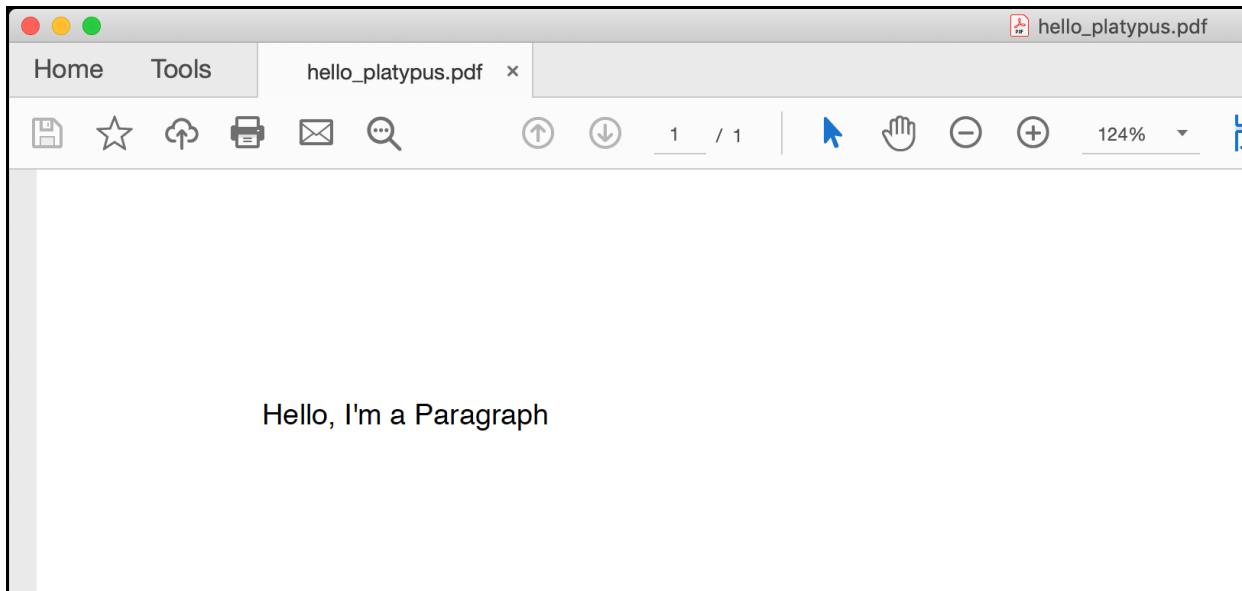


Fig. 39-5: Adding Text Using PLATYPUS

That's neat! But it's still kind of boring since it only has one element in it.

To see how useful using the PLATYPUS framework is, you will create a document with dozens of `Flowables`. Go ahead and create a new file named `platypus_multipage.py` and add this code:

```
1 # platypus_multipage.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.lib.styles import getSampleStyleSheet
5 from reportlab.lib.units import inch
6 from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer
7
8
9 def create_document():
10     doc = SimpleDocTemplate(
11         "platypus_multipage.pdf",
12         pagesize=letter,
13         )
14     styles = getSampleStyleSheet()
15     flowables = []
16     spacer = Spacer(1, 0.25*inch)
17
18     # Create a lot of content to make a multipage PDF
19     for i in range(50):
20         text = 'Paragraph #{}'.format(i)
21         para = Paragraph(text, styles["Normal"])
22         flowables.append(para)
23         flowables.append(spacer)
24
25     doc.build(flowables)
26
27 if __name__ == '__main__':
28     create_document()
```

In this example, you create 50 `Paragraph()` objects. You also create a `Spacer()`, which you use to add space between the flowables. When you create a `Paragraph()`, you add it and a `Spacer()` to the list of `flowables`. You end up with 100 flowables in the list.

When you run this code, you will generate a 3-page document. The first page will begin like the following screenshot:

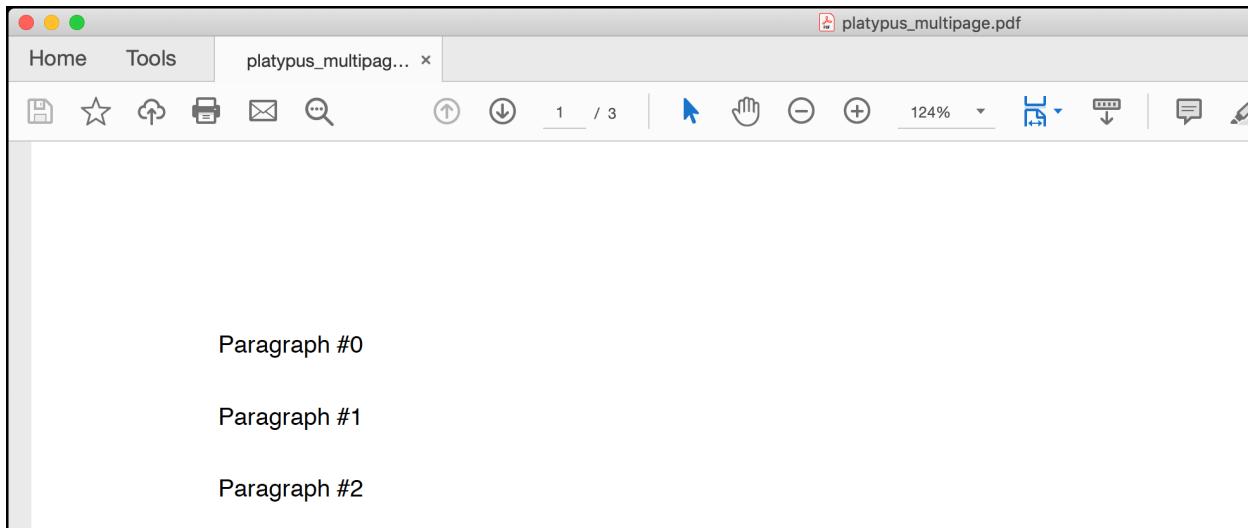


Fig. 39-6: Multipage Document with PLATYPUS

That wasn't too hard! Now let's find out how you might add a table to your PDF!

Creating a Table

One of the most complex Flowables in ReportLab is the `Table()`. It allows you to show tabular data with columns and rows. Tables allow you to put other Flowable types in each cell of the table. This allows you to create complex documents.

To get started, create a new file named `simple_table.py` and add this code to it:

```
1 # simple_table.py
2
3 from reportlab.lib.pagesizes import letter
4 from reportlab.platypus import SimpleDocTemplate, Table
5
6 def simple_table():
7     doc = SimpleDocTemplate("simple_table.pdf", pagesize=letter)
8     flowables = []
9
10    data = [
11        ['col_{}'.format(x) for x in range(1, 6)],
12        [str(x) for x in range(1, 6)],
13        ['a', 'b', 'c', 'd', 'e'],
14    ]
15
16    tbl = Table(data)
17    flowables.append(tbl)
```

```
18  
19     doc.build(flowables)  
20  
21 if __name__ == '__main__':  
22     simple_table()
```

This time you import `Table()` instead of `Paragraph()`. The rest of the imports should look familiar. To add some data to the table, you need to have a Python list of lists. The items inside the lists must be strings or Flowables. For this example, you create three rows of strings. The first row is the column row, which labels what will be in the following rows.

Next, you create the `Table()` and pass in the data, which is your list of lists. Finally, you `build()` the document as you did before. Your PDF should now have a table in it that looks like this:

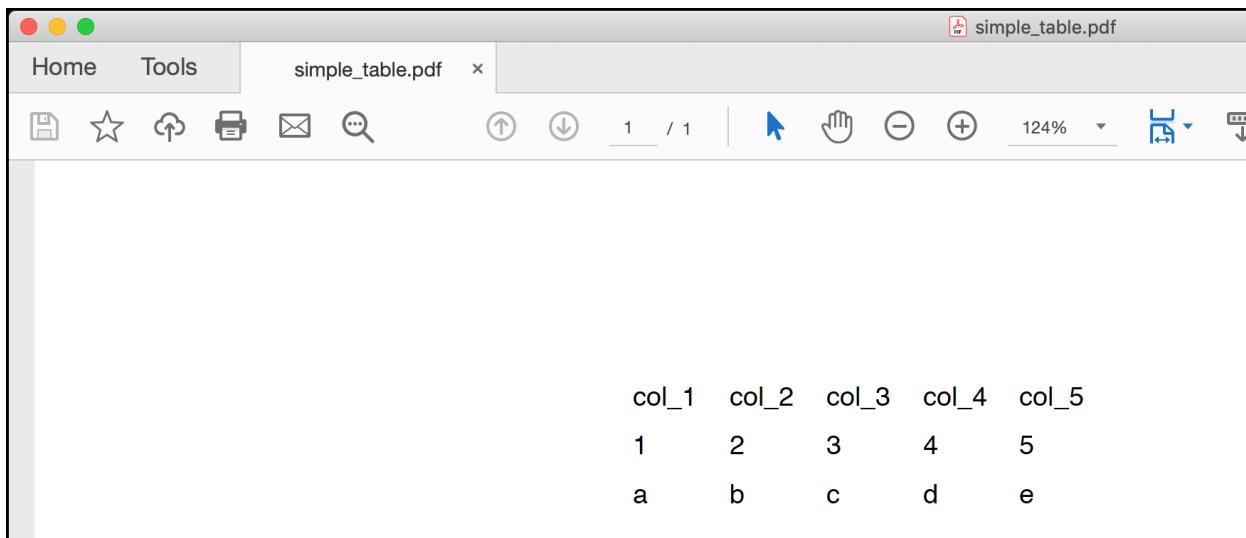


Fig. 39-7: Creating a Simple Table with PLATYPUS

A `Table()` does not have a border or a cell border turned on by default. This `Table()` has no styling applied to it at all.

Tables can have styles applied to them using a `TableStyle()`. Table styles are kind of like the stylesheets you can apply to a `Paragraph()`. To see how they work, you need to create a new file named `simple_table_with_style.py` and add the following code to it:

```
1 # simple_table_with_style.py
2
3 from reportlab.lib import colors
4 from reportlab.lib.pagesizes import letter
5 from reportlab.platypus import SimpleDocTemplate, Table, TableStyle
6
7 def simple_table_with_style():
8     doc = SimpleDocTemplate(
9         "simple_table_with_style.pdf",
10        pagesize=letter,
11        )
12    flowables = []
13
14    data = [
15        ['col_{}'.format(x) for x in range(1, 6)],
16        [str(x) for x in range(1, 6)],
17        ['a', 'b', 'c', 'd', 'e'],
18    ]
19
20    tblstyle = TableStyle([
21        ('BACKGROUND', (0, 0), (-1, 0), colors.red),
22        ('TEXTCOLOR', (0, 1), (-1, 1), colors.blue),
23    ])
24
25   tbl = Table(data)
26   tbl.setStyle(tblstyle)
27    flowables.append(tbl)
28
29    doc.build(flowables)
30
31 if __name__ == '__main__':
32     simple_table_with_style()
```

This time you add a `TableStyle()`, which is a Python list of tuples. The tuples contain the type of styling you wish to apply and which cells to apply the styling to. The first tuple states that you want to apply a red background color starting on column 0, row 0. This style should be applied through to the last column, which is specified as -1.

The second tuple applies blue color to the text, starting in column 0, row 1 though all the columns on row 1. To add the style to the table, you call `setStyle()` and pass it the `TableStyle()` instance that you created.

Note that you needed to import `colors` from `reportlab.lib` to get the two colors that you applied.

When you run this code, you will end up with the following table:

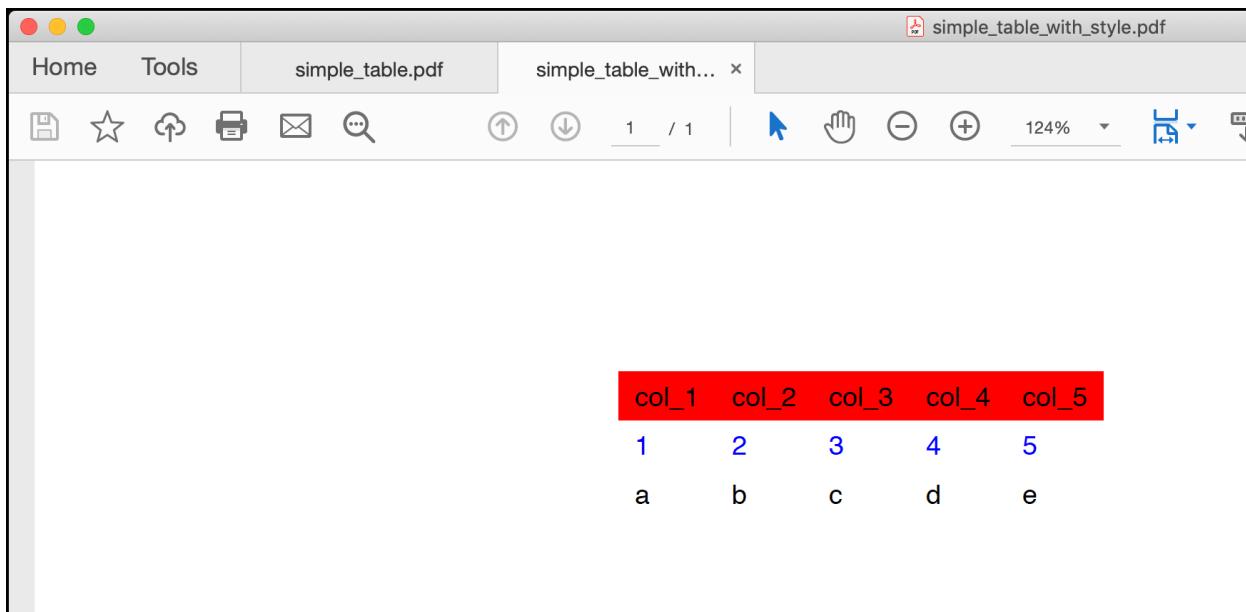


Fig. 39-8: Creating a Simple Table with a TableStyle

If you want to apply a border to the table and cells, you would add a tuple using “GRID” as the style command and then tell it which cells to apply it to.

Wrapping Up

ReportLab is the most comprehensive package available for creating PDFs with Python. In this chapter, you learned about the following topics:

- Installing ReportLab
- Creating a Simple PDF with the Canvas
- Creating Drawings and Adding Images Using the Canvas
- Creating Multi-page Documents with PLATYPUS
- Creating a Table

This chapter only scratched the surface of what you can do with ReportLab. You can use ReportLab with different types of fonts, add headers and footers, insert barcodes and much, much more. You can learn about these topics as well as other Python PDF packages in my book, **ReportLab: PDF Processing with Python**:

- <https://leanpub.com/reportlab>

Review Questions

1. What class in ReportLab do you use to draw directly on the PDF at a low-level?
2. What does PLATYPUS stand for?
3. How do you apply a stylesheet to a Paragraph?
4. Which method do you use to apply a TableStyle?

Chapter 40 - How to Create Graphs

Data visualizations are an important method of sharing your data with others. Some people refer to visualizations as plots, charts, or graphs. These names are synonymous in this chapter.

Python has many 3rd party packages that do data visualizations. In fact, there are so many that it can be somewhat overwhelming. One of the oldest and most popular is **Matplotlib**. Matplotlib is known for creating static, animated, and interactive visualizations in Python.

You can create many different types of plots and charts with Matplotlib. It also integrates well with other data science and math libraries like **NumPy** and **pandas**. You will also find that Matplotlib works with most of Python's GUI toolkits, such as Tkinter, wxPython and PyQt. Because Matplotlib is so well known, it will be the graphing package that is covered in this chapter.

You will be learning about the following topics:

- Creating a Simple Line Chart with PyPlot
- Creating a Bar Chart
- Creating a Pie Chart
- Adding Labels
- Adding Titles to Plots
- Creating a Legend
- Showing Multiple Figures

Let's start plotting with Matplotlib!

Installing Matplotlib

You will need to install Matplotlib to be able to use it. Fortunately, that is easy to do with `pip`:

```
1 python -m pip install matplotlib
```

This will install Matplotlib as well as any dependencies that it requires. Now you are ready to start graphing!

Creating a Simple Line Chart with PyPlot

Creating charts (or plots) is the primary purpose of using a plotting package. Matplotlib has a submodule called `pyplot` that you will be using to create a chart. To get started, go ahead and create a new file named `line_plot.py` and add the following code:

```
1 # line_plot.py
2
3 import matplotlib.pyplot as plt
4
5 def line_plot(numbers):
6     plt.plot(numbers)
7     plt.ylabel('Random numbers')
8     plt.show()
9
10 if __name__ == '__main__':
11     numbers = [2, 4, 1, 6]
12     line_plot(numbers)
```

Here you import `matplotlib.pyplot` as `plt`. Then you create a `line_plot()` which takes in a Python list of numbers. To plot the numbers, you use the `plot()` function. You also add a label to the y-axis. Finally, you call `show()` to display the plot.

You should now see a window that looks like this:

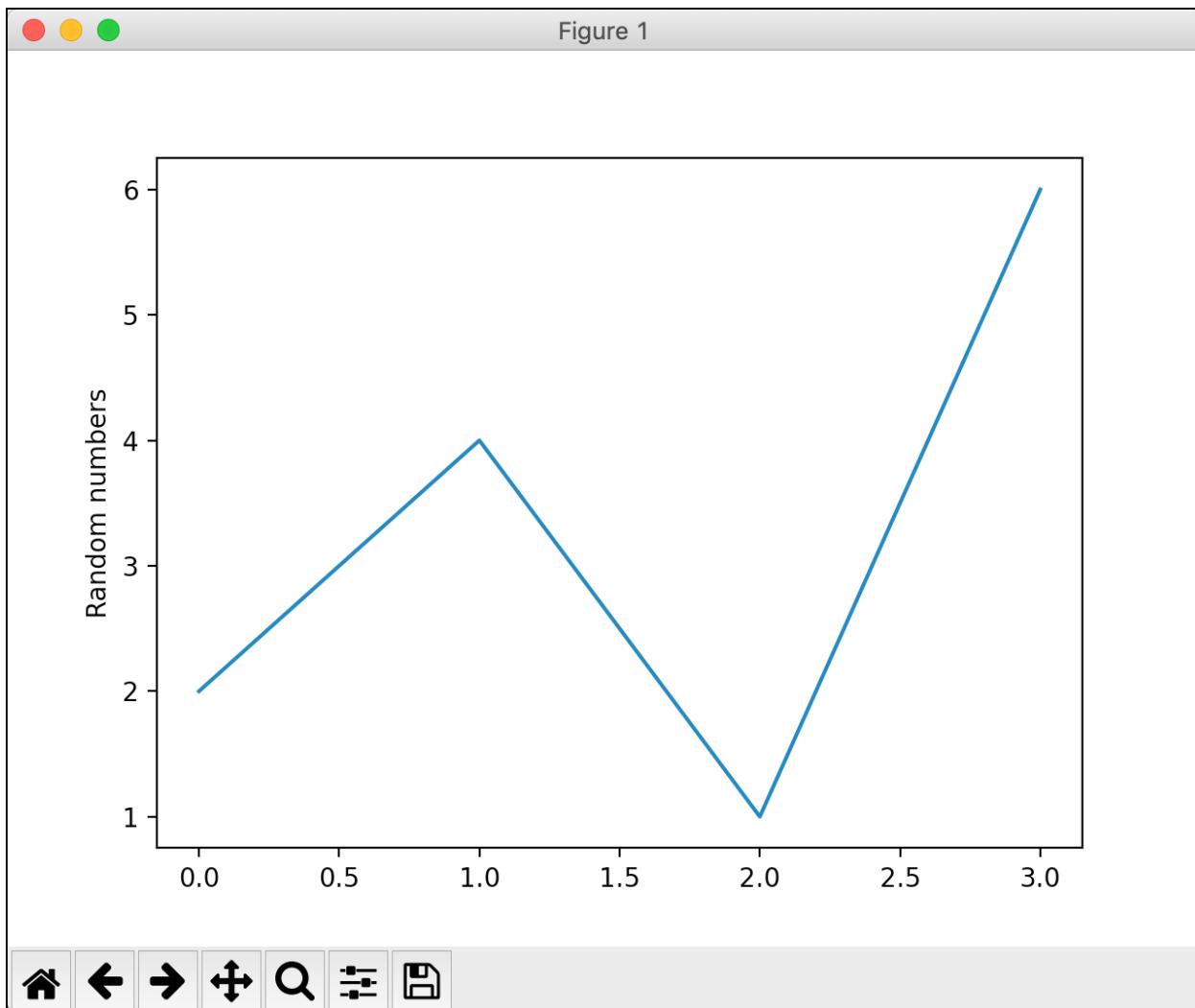


Fig. 40-1: Matplotlib Line Plot

Now you know how to create a simple line chart using Matplotlib! Now you will find out how to make a bar chart in the next section.

Creating a Bar Chart

Creating a bar chart with Matplotlib is very similar to how you created a line plot. It just takes a few extra arguments. Go ahead and create a new file named `bar_chart.py` and enter the following code into it:

```
1 # bar_chart.py
2
3 import matplotlib.pyplot as plt
4
5 def bar_chart(numbers, labels, pos):
6     plt.bar(pos, numbers, color='blue')
7     plt.xticks(ticks=pos, labels=labels)
8     plt.show()
9
10 if __name__ == '__main__':
11     numbers = [2, 1, 4, 6]
12     labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
13     pos = list(range(4))
14     bar_chart(numbers, labels, pos)
```

When you create a bar chart using `bar()`, you pass in a list of values for the x-axis. Then you pass in a list of heights for the bars. You can also optionally set a color for the bars. In this case, you set them to “blue”. Next, you set the `xticks()`, which are the tick marks that should appear along the x-axis. You also pass in a list of labels that correspond to the ticks.

Go ahead and run this code and you should see the following graph:

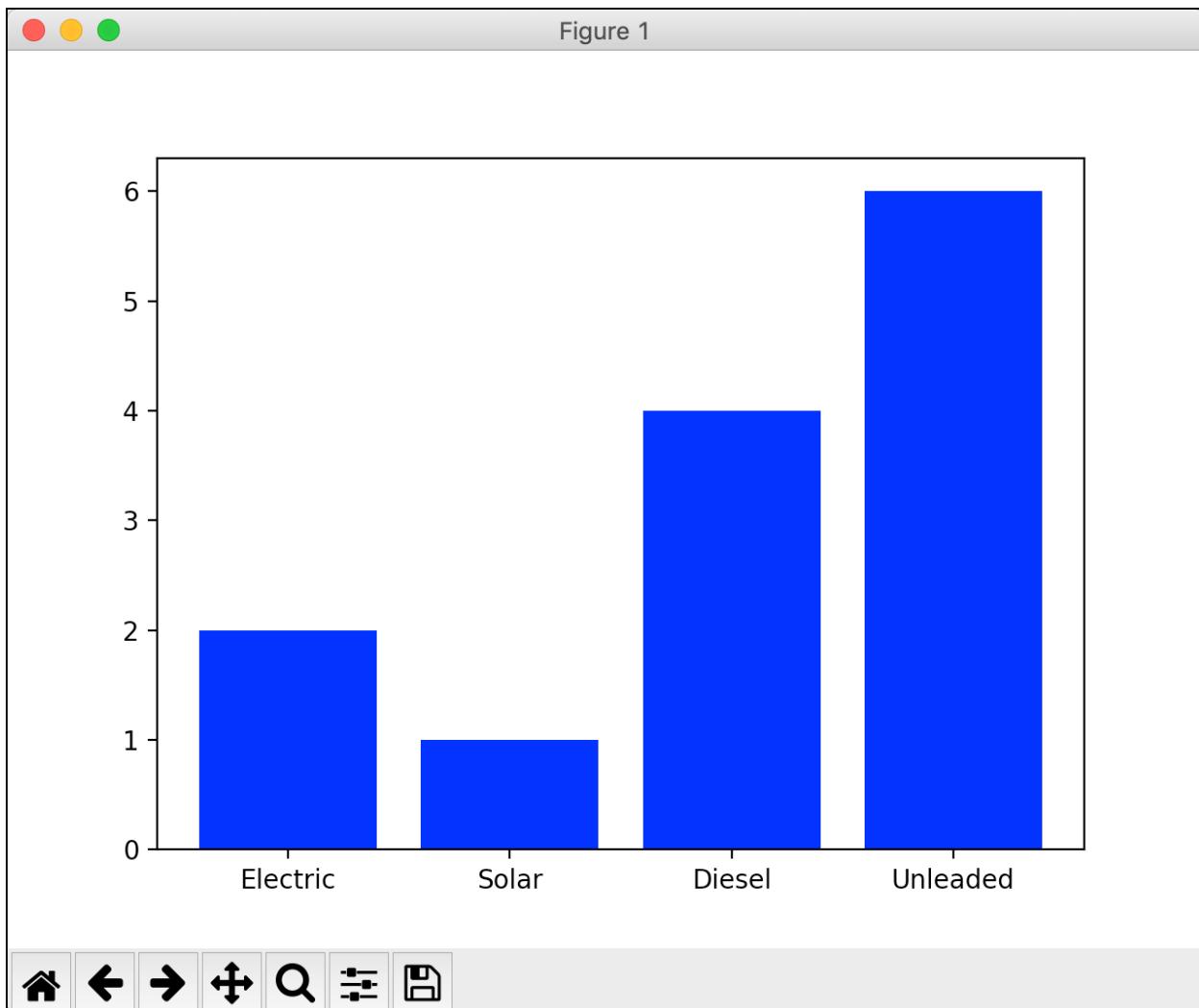


Fig. 40-2: Matplotlib Bar Chart

You can also make a horizontal bar chart with Matplotlib. All you need to do is change `bar()` to `barh()`. Create a new file named `bar_charth.py` and add this code:

```
1 # bar_charth.py
2
3 import matplotlib.pyplot as plt
4
5 def bar_charth(numbers, labels, pos):
6     plt.barh(pos, numbers, color='blue')
7     plt.yticks(ticks=pos, labels=labels)
8     plt.show()
9
10 if __name__ == '__main__':
11     numbers = [2, 1, 4, 6]
```

```
12     labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
13     pos = list(range(4))
14     bar_charth(numbers, labels, pos)
```

There is one other sneaky change here. Can you spot it? The change is that since it is now a horizontal bar chart, you will want to set the `yticks()` instead of the `xticks()` or it won't look quite right.

Once you have it all ready to go, run the code and you will see the following:

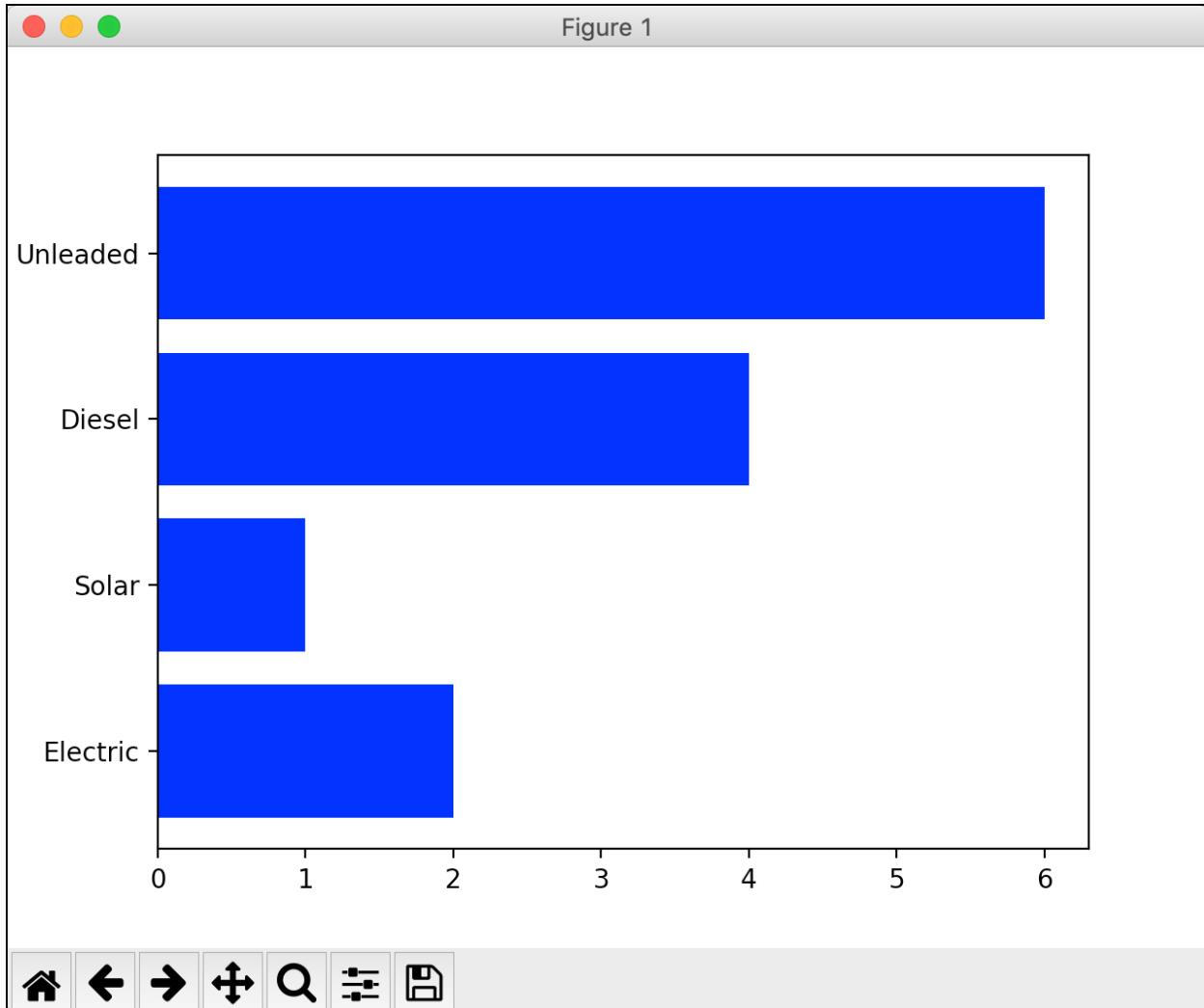


Fig. 40-3: Matplotlib Horizontal Bar Chart

That looks great and it didn't take very much code at all! Now let's find out how to create a pie chart with Matplotlib.

Creating a Pie Chart

Pie charts are a bit of a different beast. To create a pie chart, you will be using Matplotlib's `subplots()` function, which returns a `Figure` and an `Axes` object. To see how that works, create a new file named `pie_chart_plain.py` and put this code in it:

```
1 # pie_chart_plain.py
2
3 import matplotlib.pyplot as plt
4
5 def pie_chart():
6     numbers = [40, 35, 15, 10]
7     labels = ['Python', 'Ruby', 'C++', 'PHP']
8
9     fig1, ax1 = plt.subplots()
10    ax1.pie(numbers, labels=labels)
11    plt.show()
12
13 if __name__ == '__main__':
14     pie_chart()
```

In this code, you create `subplots()` and then use the `pie()` method of the `Axes` object. You pass in a list of numbers as you did before, as well as a list of labels. Then when you run the code, you will see your pie chart:

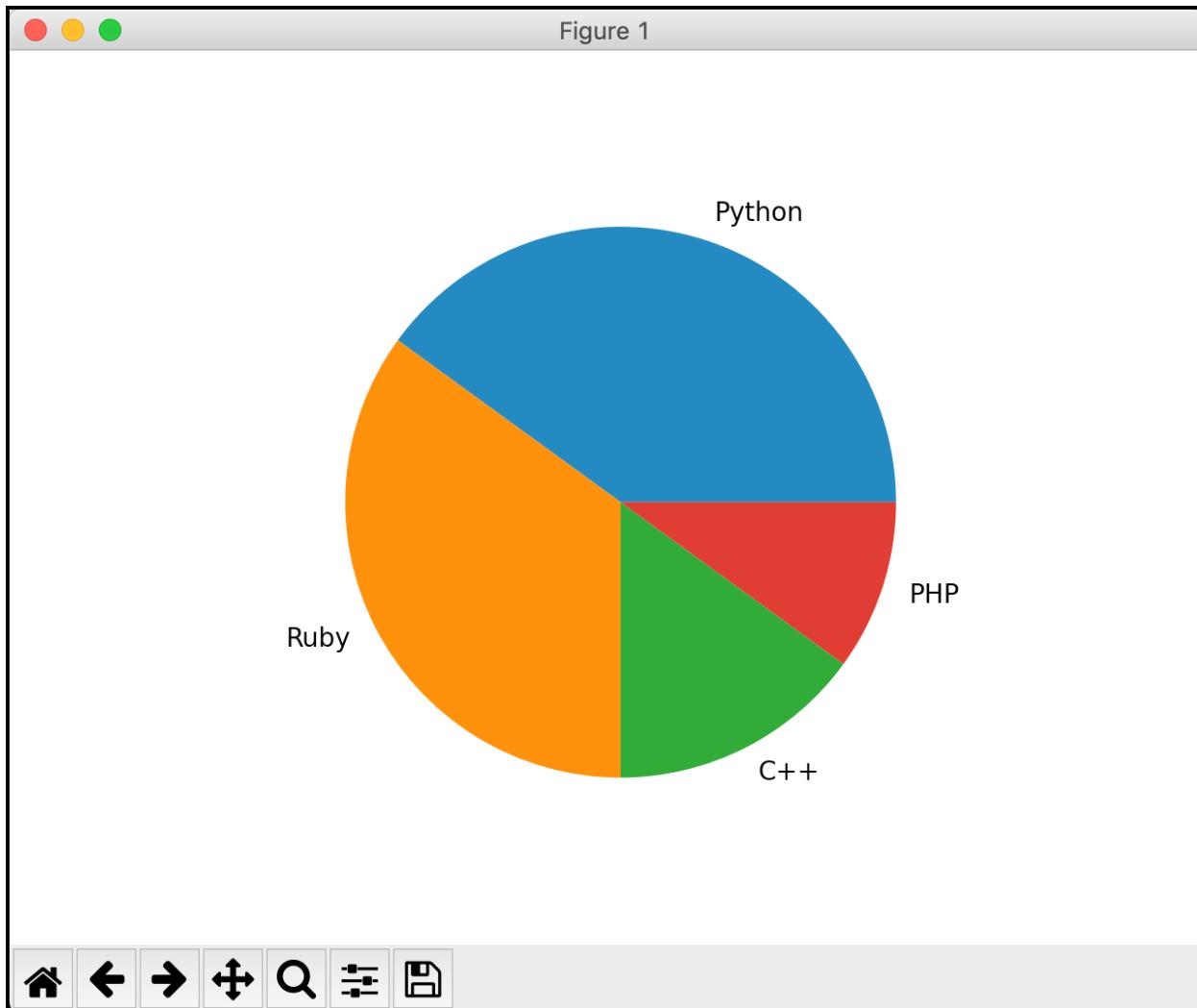


Fig. 40-4: Matplotlib Pie Chart

That's pretty nice for such a short piece of code. But you can make your pie charts look even better. Create a new file named `pie_chart_fancy.py` and add this code to see how:

```
1 # pie_chart_fancy.py
2
3 import matplotlib.pyplot as plt
4
5 def pie_chart():
6     numbers = [40, 35, 15, 10]
7     labels = ['Python', 'Ruby', 'C++', 'PHP']
8     # Explode the first slice (Python)
9     explode = (0.1, 0, 0, 0)
10
11    fig1, ax1 = plt.subplots()
```

```
12     ax1.pie(numbers, explode=explode, labels=labels,
13                 shadow=True, startangle=90,
14                 autopct='%1.1f%%')
15     ax1.axis('equal')
16     plt.show()
17
18 if __name__ == '__main__':
19     pie_chart()
```

For this example, you use the `explode` parameter to tell the pie chart to “explode” or remove a slice from the pie. In this case, you remove the first slice, which corresponds to “Python”. You also add a shadow to the pie chart. You can tell your pie chart to rotate a certain number of degrees counter-clockwise by setting the `startangle`. If you’d like to show the slice percentages, you can use `autopct`, which will use Python’s string interpolation syntax.

When you run this code, your pie chart will now look like this:

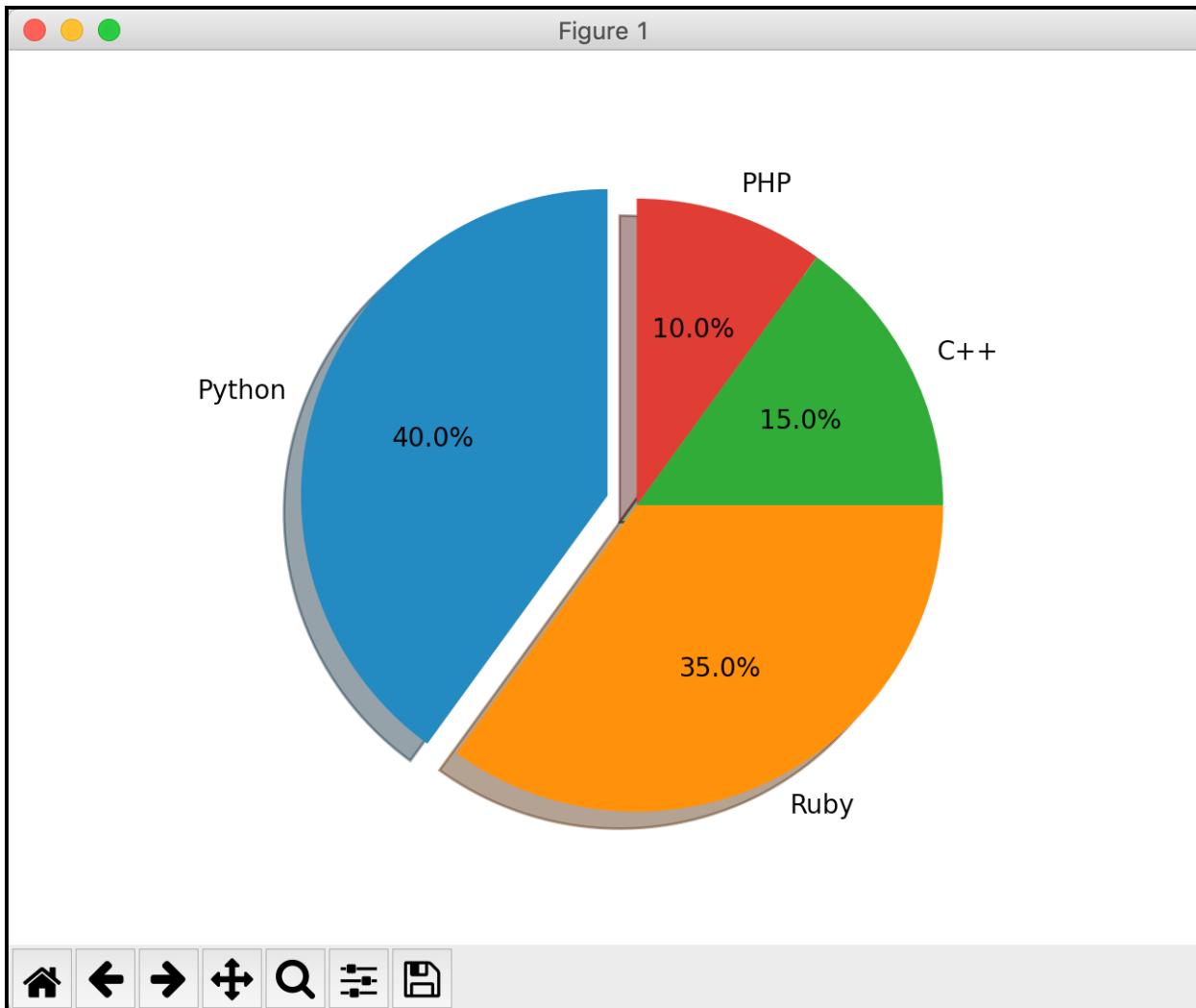


Fig. 40-5: Matplotlib Fancy Pie Chart

Isn't that neat? Your pie chart now looks much more polished! Now it's time to learn how to add labels to your other graphs!

Adding Labels

When you are graphing data, you will usually want to label the axes. You can label the x-axis by using the `xlabel()` function and you can label the y-axis by using the corresponding `ylabel()` function. To see how this works, create a file named `bar_chart_labels.py` and add this code to it:

```
1 # bar_chart_labels.py
2
3 import matplotlib.pyplot as plt
4
5 def bar_chart(numbers, labels, pos):
6     plt.bar(pos, numbers, color='blue')
7     plt.xticks(ticks=pos, labels=labels)
8     plt.xlabel('Vehicle Types')
9     plt.ylabel('Number of Vehicles')
10    plt.show()
11
12 if __name__ == '__main__':
13     numbers = [2, 1, 4, 6]
14     labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
15     pos = list(range(4))
16     bar_chart(numbers, labels, pos)
```

Here you call both `xlabel()` and `ylabel()` and set them to different strings. This adds some explanatory text underneath the graph and to the left of the graph, respectively. Here is what the result looks like:

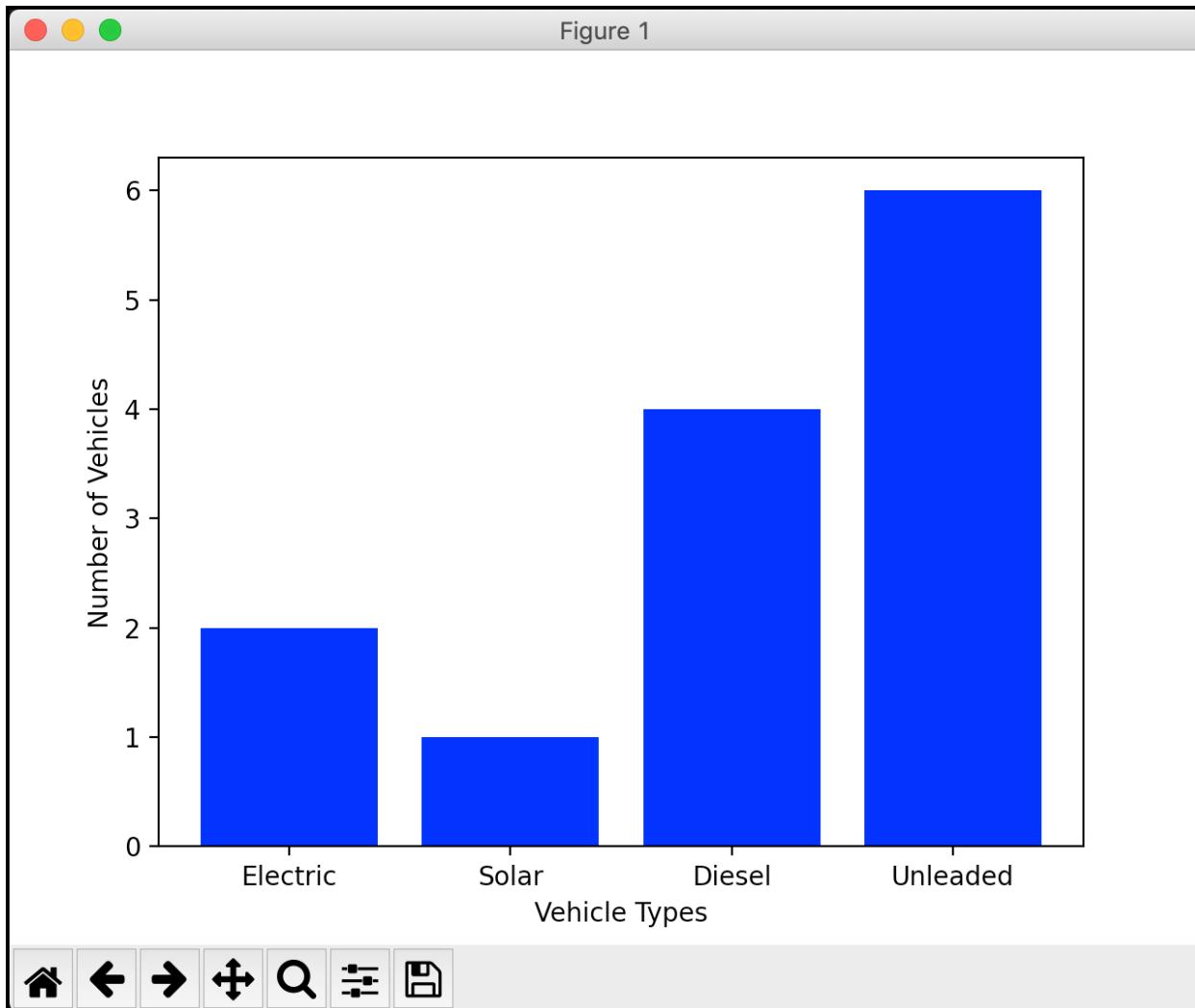


Fig. 40-6: Matplotlib Bar Chart with Labels

That looks quite nice. Your graph is easier to understand, but it is missing a title. You will learn how to do that in the next section!

Adding Titles to Plots

Adding titles to your graphs with Matplotlib is quite straightforward. In fact, all you need to do is use the `title()` function to add one. To find out how, create a new file named `bar_chart_title.py` and add this code to it:

```
1 # bar_chart_title.py
2
3 import matplotlib.pyplot as plt
4
5 def bar_chart(numbers, labels, pos):
6     plt.bar(pos, [4, 5, 6, 3], color='green')
7     plt.bar(pos, numbers, color='blue')
8     plt.xticks(ticks=pos, labels=labels)
9     plt.title('Gas Used in Various Vehicles')
10    plt.xlabel('Vehicle Types')
11    plt.ylabel('Number of Vehicles')
12    plt.show()
13
14 if __name__ == '__main__':
15     numbers = [2, 1, 4, 6]
16     labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
17     pos = list(range(4))
18     bar_chart(numbers, labels, pos)
```

The primary change here is on line 9 where you call `title()` and pass in a string. This sets the title for the graph and centers it along the top by default. You can change the location slightly by setting the `loc` parameter to “left” or “right”, but you can’t specify that the title be anywhere but the top. There is also a `fontdict` parameter that you can use for controlling the appearance of the title font.

You also add a new bar plot to the graph. This helps you see what a stacked bar plot looks like and also prepares you for the next section.

Here is what your graph looks like now:

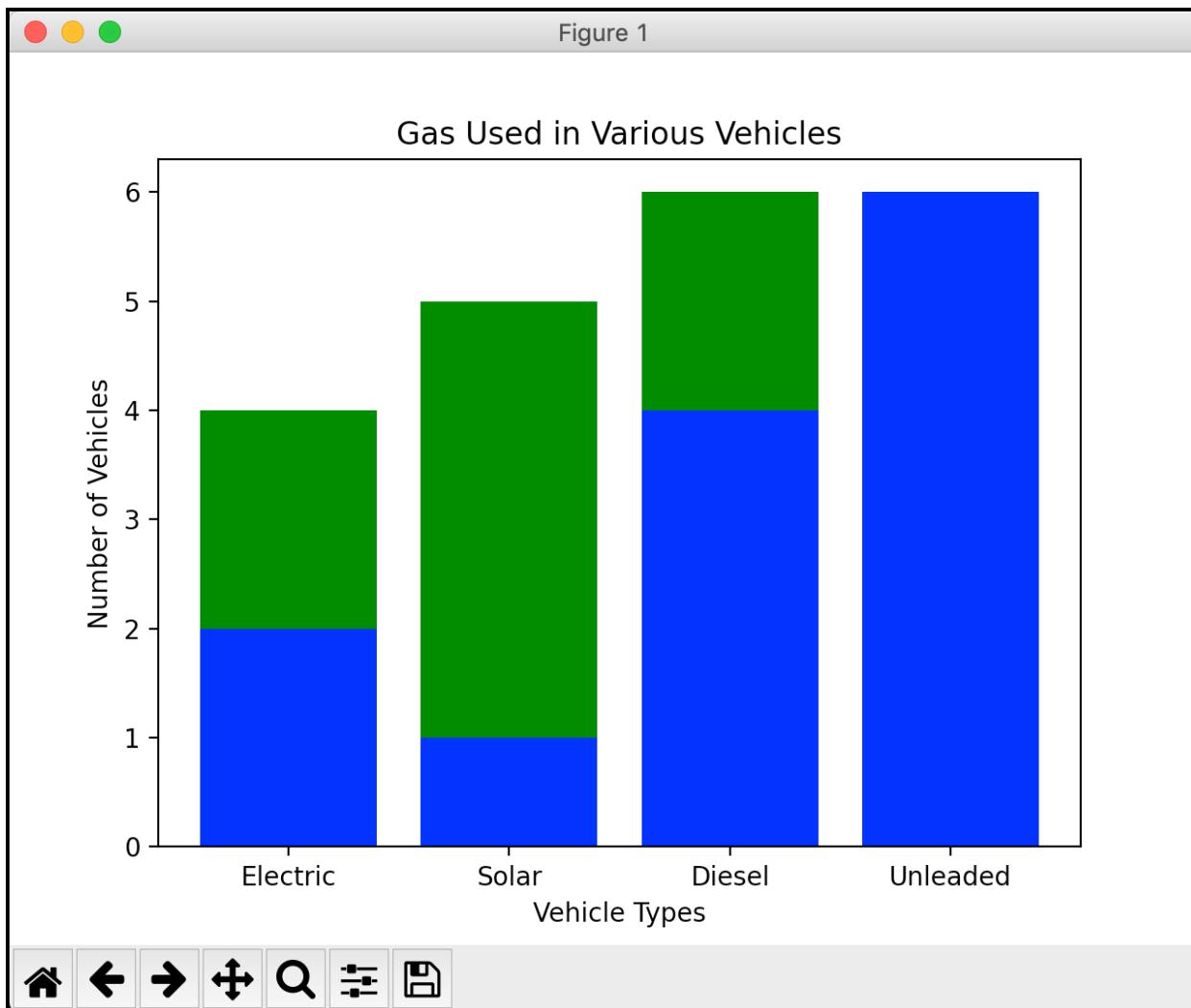


Fig. 40-7: Matplotlib Bar Chart with Title

This graph looks even better, but it's still missing something. Oh! You need a legend! Let's find out how to do that next.

Creating a Legend

Adding a legend to your Matplotlib graph is also straightforward. You will use the `legend()` function to add one. Create a new file named `bar_chart_legend.py`. Then, add this code to it:

```
1 # bar_chart_legend.py
2
3 import matplotlib.pyplot as plt
4
5 def bar_chart(numbers, labels, pos):
6     plt.bar(pos, [4, 5, 6, 3], color='green')
7     plt.bar(pos, numbers, color='blue')
8     plt.xticks(ticks=pos, labels=labels)
9     plt.xlabel('Vehicle Types')
10    plt.ylabel('Number of Vehicles')
11    plt.legend(['First Label', 'Second Label'], loc='upper left')
12    plt.show()
13
14 if __name__ == '__main__':
15     numbers = [2, 1, 4, 6]
16     labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
17     pos = list(range(4))
18     bar_chart(numbers, labels, pos)
```

Here you add a `legend()` right before you `show()` the graph. When you create a legend, you can set the labels by passing in a list of strings. The list should match the number of plots in your graph. You can also set the location of the legend by using the `loc` parameter.

When you run this code, you will see your graph updated to look like this:

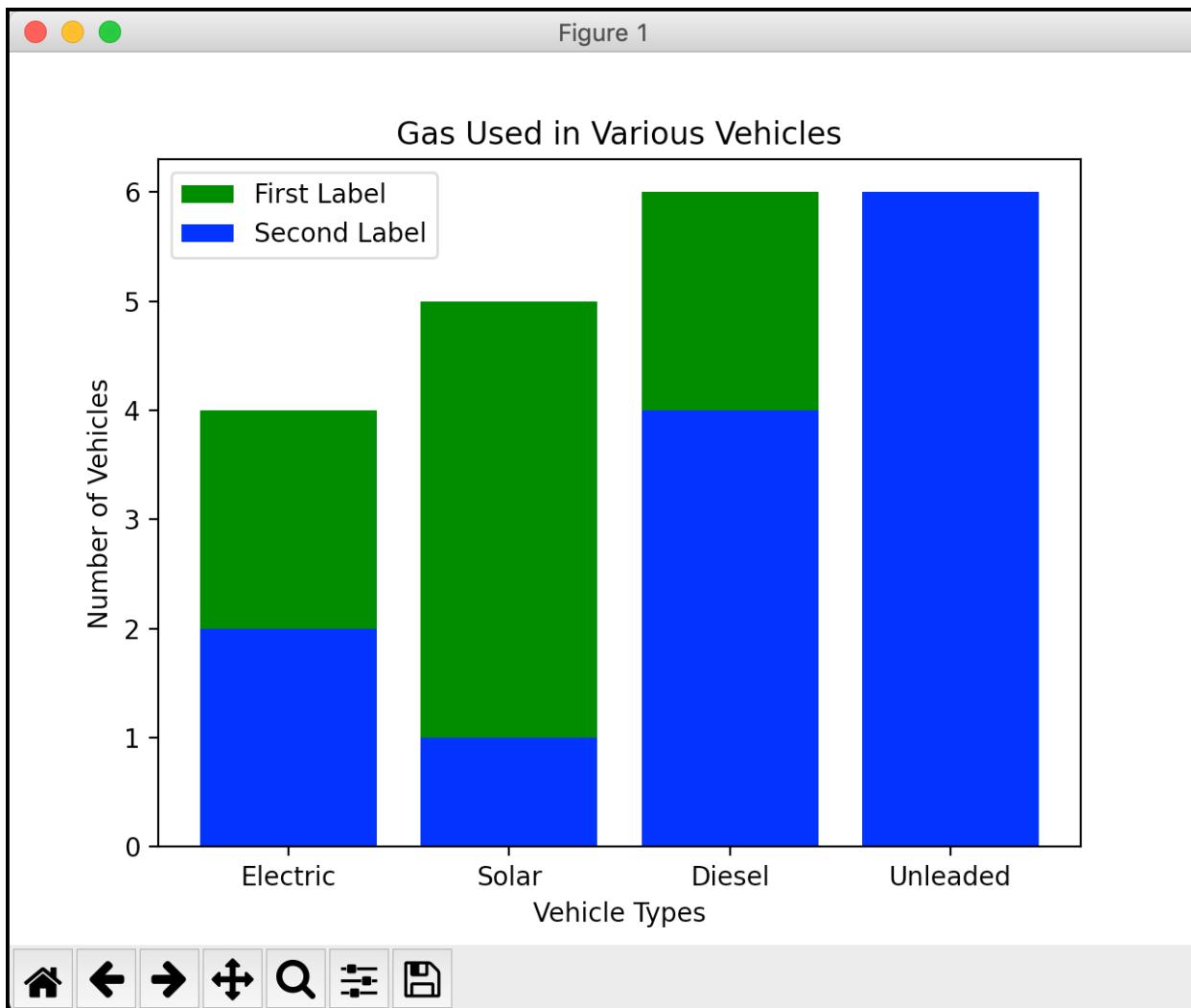


Fig. 40-8: Matplotlib Bar Chart with Legend

Now your graph has all the normal components that you would expect to have in a graph. At this point you've already seen a lot of tasks you can accomplish using Matplotlib. The last topic to learn about is how to add multiple figures with Matplotlib.

Showing Multiple Figures

Matplotlib allows you to create several plots before you show them. This lets you work with multiple datasets at once. There are several different ways you can do this. You will look at one of the simplest ways to do so.

Create a new file named `multiple_figures.py` and add this code:

```
1 # multiple_figures.py
2
3 import matplotlib.pyplot as plt
4
5 def line_plot(numbers, numbers2):
6     first_plot = plt.figure(1)
7     plt.plot(numbers)
8
9     second_plot = plt.figure(2)
10    plt.plot(numbers2)
11    plt.show()
12
13 if __name__ == '__main__':
14     numbers = [2, 4, 1, 6]
15     more_numbers = [5, 1, 10, 3]
16     line_plot(numbers, more_numbers)
```

Here you create two line plots. Before you plot, you call `figure()`, which creates a top-level container for the plots that follow after it is called. Thus the first plot is added to figure one and the second plot is added to figure 2. When you then call `show()` at the end, Matplotlib will open two windows with each graph shown separately.

Run the code and you will see the following two windows on your machine:

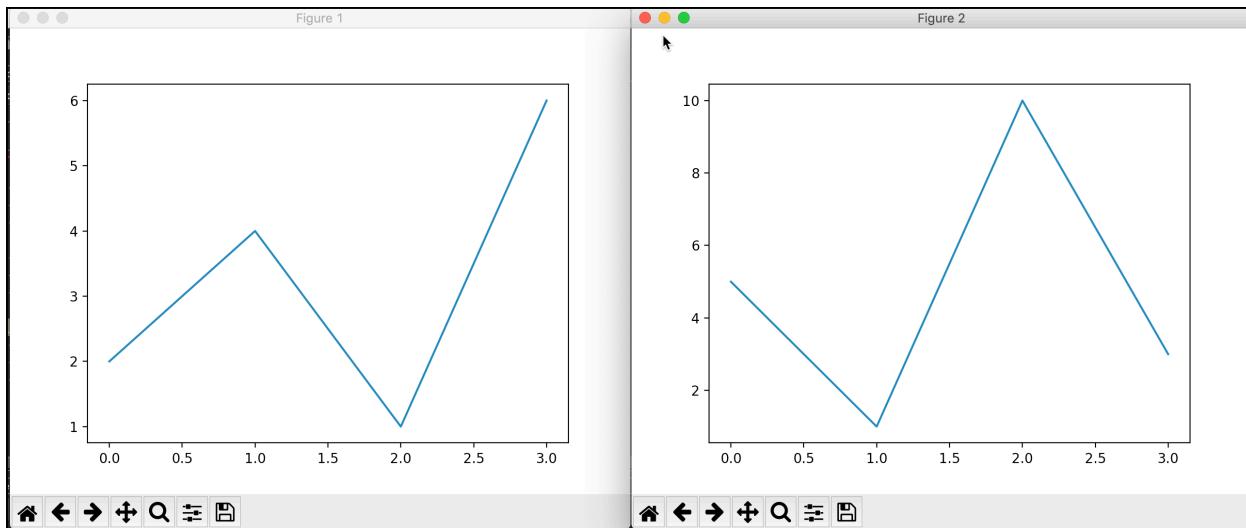


Fig. 40-10: Matplotlib Multiple Figures

Matplotlib also supports adding two or more plots to a single window. To see how that works, create another new file and name this one `multiple_plots.py`. To make things more interesting, you will use NumPy in this example to create the two plots.

Note: If you haven't already, you will need to install NumPy to get this example to work.

This example is based on one from the Matplotlib documentation:

```
1 # multiple_plots.py
2
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 def multiple_plots():
7     # Some example data to display
8     x = np.linspace(0, 2 * np.pi, 400)
9     y = np.sin(x ** 2)
10
11    fig, axs = plt.subplots(2)
12    fig.suptitle('Vertically stacked subplots')
13    axs[0].plot(x, y)
14    axs[1].plot(x, -y)
15    plt.show()
16
17 if __name__ == '__main__':
18     multiple_plots()
```

Here you create what amounts to two separate sine wave graphs. To make them both show up in the same window, you use a call to `subplots()`, which is a handy utility for creating multiple figures in a single call. You can then use the `Axes` object that it returns to plot the data you created with NumPy.

The result ends up looking like this:

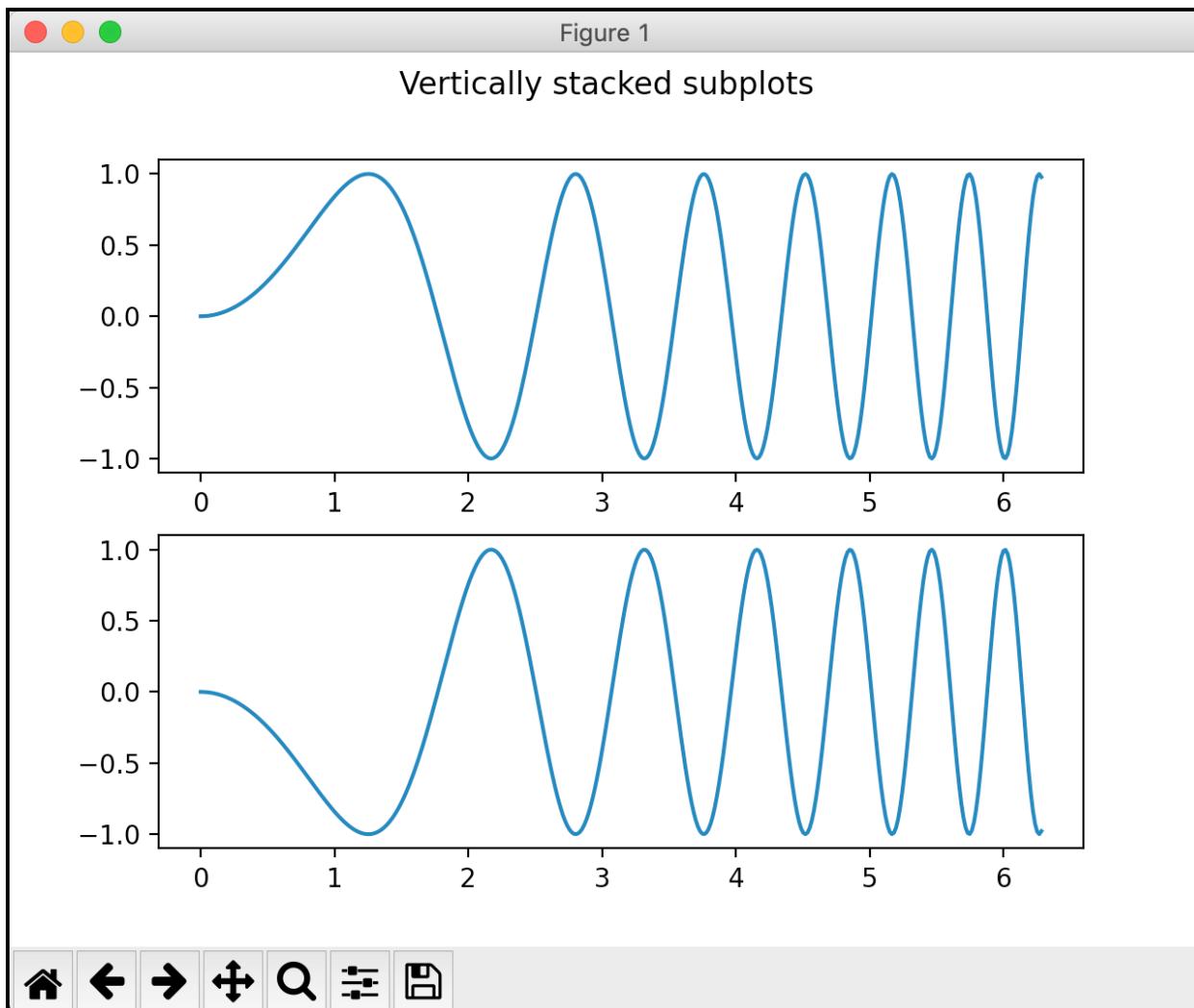


Fig. 40-10: Matplotlib Stacked Plots

If you don't want to use NumPy, you could plot the two sets of numbers from the previous example. In fact, you should try that. Go ahead and create a new file named `multiple_plots2.py` and add this code:

```
1 # multiple_plots2.py
2
3 import matplotlib.pyplot as plt
4
5 def multiple_plots():
6     numbers = [2, 4, 1, 6]
7     more_numbers = [5, 1, 10, 3]
8     fig, axs = plt.subplots(2)
9     fig.suptitle('Vertically stacked subplots')
10    axs[0].plot(numbers)
```

```
11     axs[1].plot(more_numbers)
12     plt.show()
13
14 if __name__ == '__main__':
15     multiple_plots()
```

In this code, you remove the NumPy code entirely and add the two lists of numbers from the earlier example. Then you plot them using the `Axes` object.

This results in the following stacked plot:

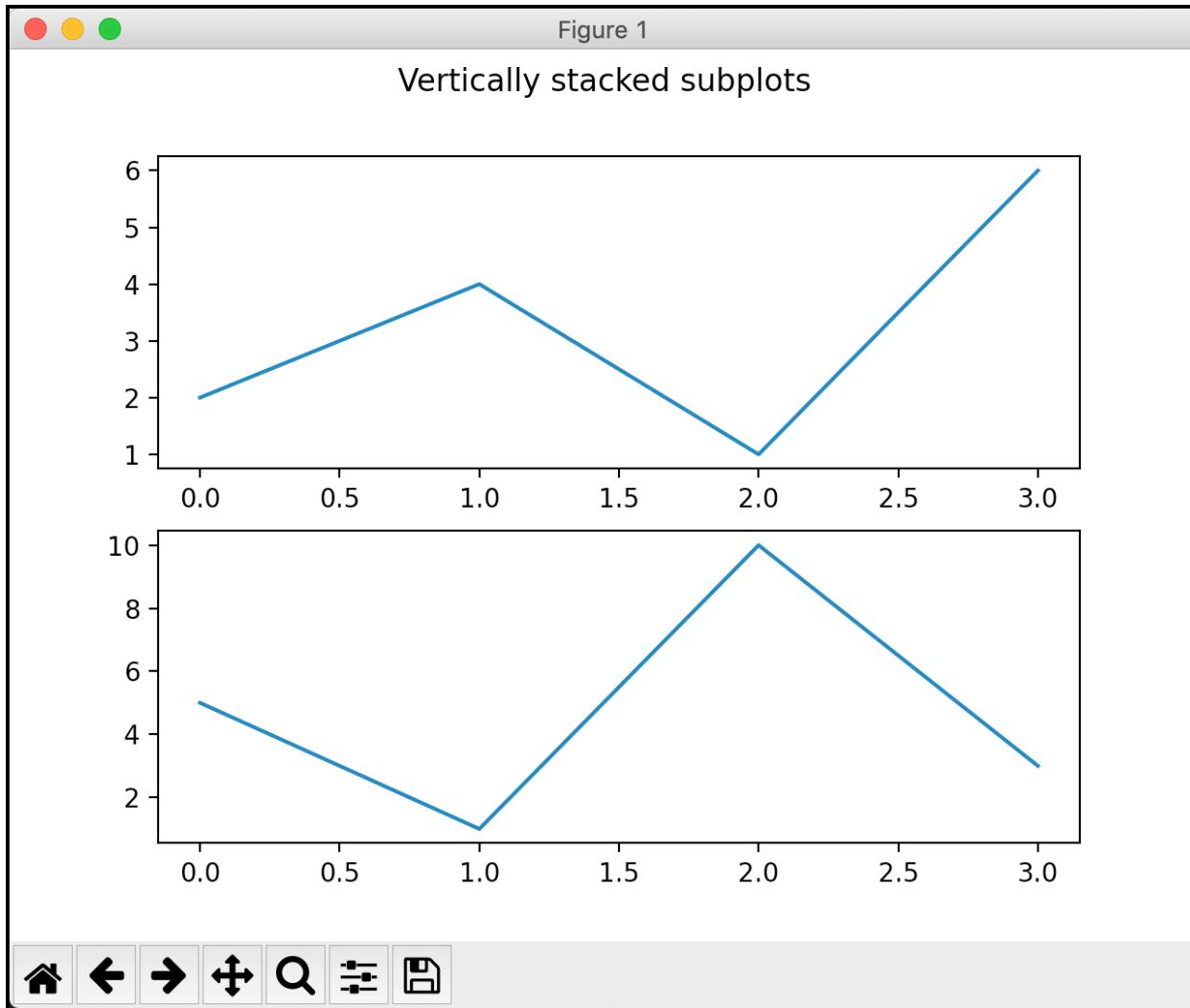


Fig. 40-11: Matplotlib Stacked Plots without NumPy

At this point, you should have a pretty good handle on how to create multiple figures and stacked plots with Matplotlib.

Wrapping Up

Matplotlib is a great package that you can use to create all kinds of neat graphs. It's amazing how few lines of code you need to write to create a useful plot from your data. In this chapter, you learned about the following topics:

- Creating a Simple Line Chart with PyPlot
- Creating a Bar Chart
- Creating a Pie Chart
- Adding Labels
- Adding Titles to Plots
- Creating a Legend
- Showing Multiple Figures

Matplotlib is very powerful and has many features that are not covered here. You can create many other types of visualizations with Matplotlib. There is a newer package called **Seaborn** that is built on top of Matplotlib and makes its graphs look even nicer. There are also many other completely separate graphing packages for Python. You will find that Python has support for almost any type of graph you can think of and probably many you didn't know existed.

Review Questions

1. Which module in Matplotlib do you use to create plots?
2. How do you add a label to the x-axis of a plot?
3. Which functions do you use to create titles and legends for plots?

Chapter 41 - How to Work with Images in Python

The **Python Imaging Library** (PIL) is a 3rd party Python package that adds image processing capabilities to your Python interpreter. It allows you to process photos and do many common image file manipulations. The current version of this software is in **Pillow**, which is a fork of the original PIL to support Python 3. Several other Python packages, such as wxPython and ReportLab, use Pillow to support loading many different image file types. You can use Pillow for several use cases including the following:

- Image processing
- Image archiving
- Batch processing
- Image display via Tkinter

In this chapter, you will learn how to do the following with Pillow:

- Opening Images
- Cropping Images
- Using Filters
- Adding Borders
- Resizing Images

As you can see, Pillow can be used for many types of image processing. The images used in this chapter are some that the author has taken himself. They are included with the code examples on Github. See the introduction for more details.

Now let's get started by installing Pillow!

Installing Pillow

Installing Pillow is easy to do with pip. Here is how you would do it after opening a terminal or console window:

```
1 python -m pip install pillow
```

Now that Pillow is installed, you are ready to start using it!

Opening Images

Pillow let's you open and view many different file types. For a full listing of the image file types that Pillow supports, see the following:

- <https://pillow.readthedocs.io/en/stable/handbook/image-file-formats.html>

You can use Pillow to open and view any of the file types mentioned in the “fully supported formats” section at the link above. The viewer is made with Tkinter and works in much the same way as Matplotlib does when it shows a graph.

To see how this works, create a new file named `open_image.py` and enter the following code:

```
1 # open_image.py
2
3 from PIL import Image
4
5 image = Image.open('jellyfish.jpg')
6 image.show()
```

Here you import `Image` from the `PIL` package. Then you use `Image.open()` to open up an image. This will return an `PIL.JpegImagePlugin.JpegImageFile` object that you can use to learn more about your image. When you run this code, you will see a window similar to the following:

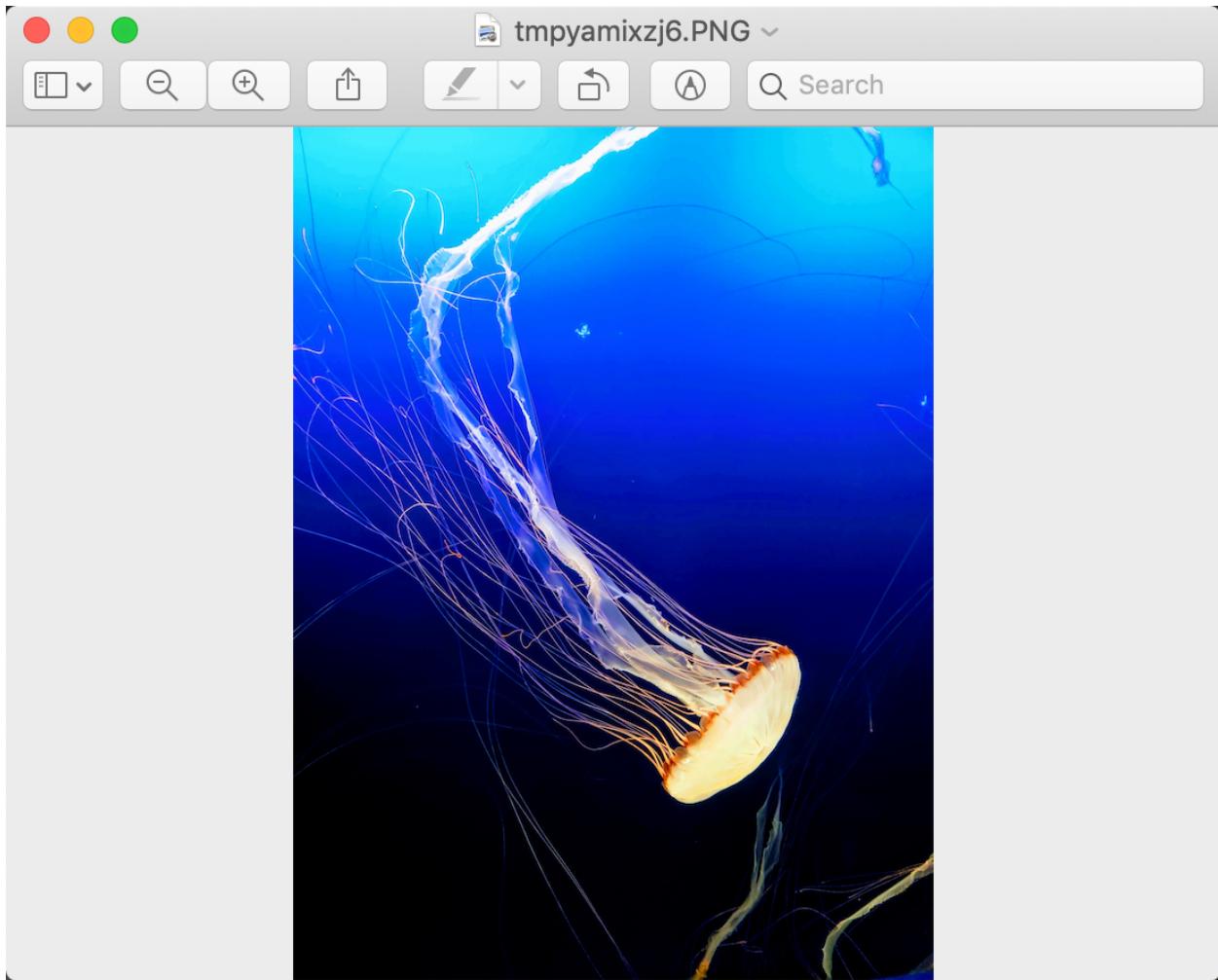


Fig. 41-1: Displaying an Image with Pillow

This is pretty handy because now you can view your images with Python without writing an entire graphical user interface. You can use Pillow to learn more about your images as well. Create a new file named `get_image_info.py` and add this code to it:

```
1 # get_image_info.py
2
3 from PIL import Image
4
5 def get_image_info(path):
6     image = Image.open(path)
7     print(f'This image is {image.width} x {image.height}')
8     exif = image._getexif()
9     print(exif)
10
11 if __name__ == '__main__':
```

```
12     get_image_info('ducks.jpg')
```

Here you get the width and height of the image using the `image` object. Then you use the `_getexif()` method to get metadata about your image. EXIF stands for “Exchangeable image file format” and is a standard that specifies the formats for images, sound, and ancillary tags used by digital cameras. The output is pretty verbose, but you can learn from that data that this particular photo was taken with a Sony 6300 camera with the following settings: “E 18-200mm F3.5-6.3 OSS LE”. The timestamp for the photo is also in the Exif information.

However, the Exif data can be altered if you use photo editing software to crop, apply filters or do other types of image manipulation. This can remove part or all of the Exif data. Try running this function on some of your own photos and see what kinds of information you can extract!

Another fun bit of information that you can extract from the image is its histogram data. The histogram of an image is a graphical representation of its tonal values. It shows you the brightness of the photo as a list of values that you could graph. Let’s use this image as an example:



Fig. 41-2: A fun butterfly

To get the histogram from this image you will use the image’s `histogram()` method. Then you will use Matplotlib (see last chapter) to graph it out. To see one way that you could do that, create a new file named `get_histogram.py` and add this code to it:

```
1 # get_histogram.py
2
3 import matplotlib.pyplot as plt
4
5 from PIL import Image
6
7 def get_image_histogram(path):
8     image = Image.open(path)
9     histogram = image.histogram()
10    plt.hist(histogram, bins=len(histogram))
11    plt.xlabel('Histogram')
12    plt.show()
13
14 if __name__ == '__main__':
15     get_image_histogram('butterfly.jpg')
```

When you run this code, you open the image as before. Then you extract the histogram from it and pass the list of values to your Matplotlib object where you call the `hist()` function. The `hist()` function takes in the list of values and the number of equal width bins in the range of values.

When you run this code, you will see the following graph:

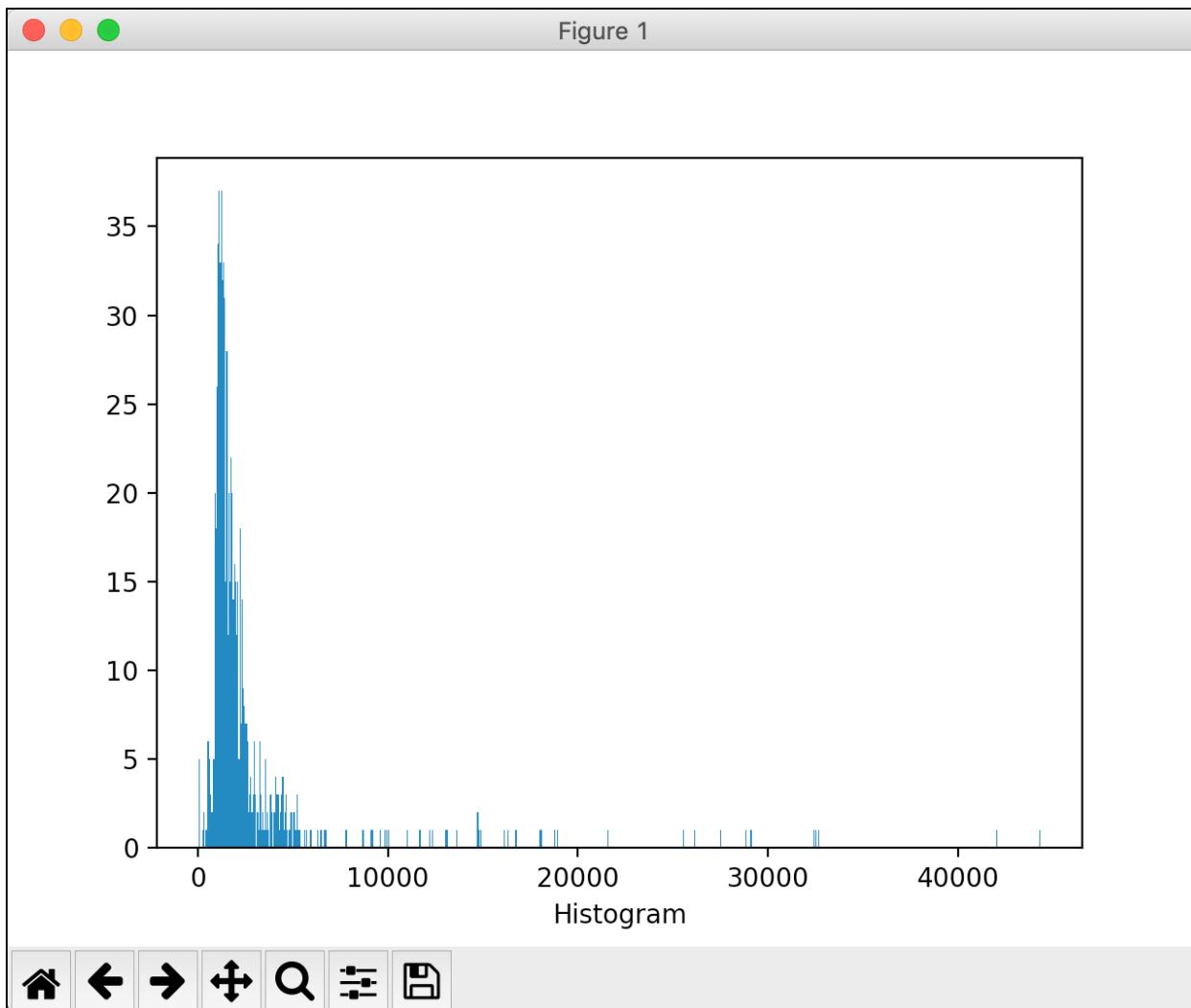


Fig. 41-3: The histogram of the butterfly

This graph shows you the tonal values in the image that were mentioned earlier. You can try passing in some of the other images included on Github to see different graphs or swap in some of your own images to see their histograms.

Now let's discover how you can use Pillow to crop images!

Cropping Images

When you are taking photographs, all too often the subject of the photo will move or you didn't zoom in far enough. This results in a photo where the focus of the image isn't really front-and-center. To fix this issue, you can crop the image to that part of the image that you want to highlight.

Pillow has this functionality built-in. To see how it works, create a file named `cropping.py` and add the following code to it:

```
1 # cropping.py
2
3 from PIL import Image
4
5 def crop_image(path, cropped_path):
6     image = Image.open(path)
7     cropped = image.crop((40, 590, 979, 1500))
8     cropped.save(cropped_path)
9
10 if __name__ == '__main__':
11     crop_image('ducks.jpg', 'ducks_cropped.jpg')
```

The `crop_image()` function takes in the path of the file that you wish to crop as well as the path to the new cropped file. You then `open()` the file as before and call `crop()`. This method takes the beginning and ending x/y coordinates that you are using to crop with. You are creating a box that is used for cropping.

Let's take this fun photo of ducks and try cropping it with the code above:



Fig. 41-4: Baby ducks

Now when you run the code against this, you will end up with the following cropped image:



Fig. 41-5: Cropped baby ducks

The coordinates you use to crop with will vary with the photo. In fact, you should probably change this code so that it accepts the crop coordinates as arguments. You can do that yourself as a little homework. It takes some trial and error to figure out the crop bounding box to use. You can use a tool like Gimp to help you by drawing a bounding box with Gimp and noting the coordinates it gives you to try with Pillow.

Now let's move on and learn about applying filters to your images!

Using Filters

The Pillow package has several filters that you can apply to your images. These are the current filters that are supported:

- BLUR
- CONTOUR
- DETAIL
- EDGE_ENHANCE
- EDGE_ENHANCE_MORE
- EMBOSS
- FIND_EDGES
- SHARPEN
- SMOOTH
- SMOOTH_MORE

Let's use the butterfly image from earlier to test out a couple of these filters. Here is the image you will be using:

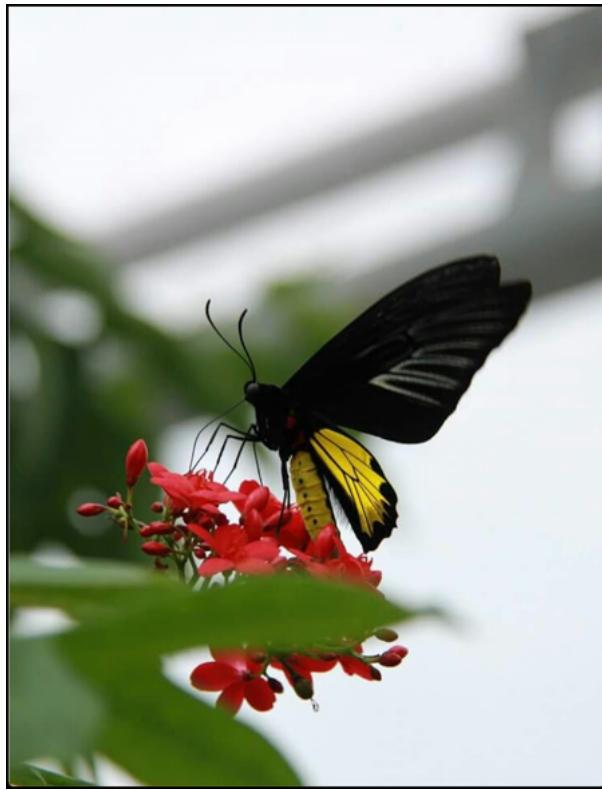


Fig. 41-6: A fun butterfly

Now that you have an image to use, go ahead and create a new file named `blur.py` and add this code to it to try out Pillow's `BLUR` filter:

```
1 # blur.py
2
3 from PIL import Image
4 from PIL import ImageFilter
5
6 def blur(path, modified_photo):
7     image = Image.open(path)
8     blurred_image = image.filter(ImageFilter.BLUR)
9     blurred_image.save(modified_photo)
10
11 if __name__ == '__main__':
12     blur('butterfly.jpg', 'butterfly_blurred.jpg')
```

To actually use a filter in Pillow, you need to import `ImageFilter`. Then you pass in the specific filter that you want to use to the `filter()` method. When you call `filter()`, it will return a new image

object. You then save that file to disk.

This is the image that you will get when you run the code:



Fig. 41-7: A blurry butterfly

That looks kind of blurry, so you can count this as a success! If you want it to be even blurrier, you could run the blurry photo back through your script a second time.

Of course, sometimes you take photos that are slightly blurry and you want to sharpen them up a bit. Pillow includes that as a filter you can apply as well. Create a new file named `sharpen.py` and add this code:

```
1 # sharpen.py
2
3 from PIL import Image
4 from PIL import ImageFilter
5
6 def sharpen(path, modified_photo):
7     image = Image.open(path)
8     sharpened_image = image.filter(ImageFilter.SHARPEN)
9     sharpened_image.save(modified_photo)
10
11 if __name__ == '__main__':
```

```
12     sharpen('butterfly.jpg', 'butterfly_sharper.jpg')
```

Here you take the original butterfly photo and apply the SHARPEN filter to it before saving it off. When you run this code, your result will look like this:



Fig. 41-8: A sharper butterfly

Depending on your eyesight and your monitor's quality, you may or may not see much difference here. However, you can rest assured that it is slightly sharper.

Now let's find out how you can add borders to your images!

Adding Borders

One way to make your photos look more professional is to add borders to them. Pillow makes this pretty easy to do via their `ImageOps` module. But before you can do any borders, you need an image. Here is the one you'll be using:



Fig. 41-9: Another butterfly

Now that you have a nice image to play around with, go ahead and create a file named `border.py` and put this code into it:

```
1 # border.py
2
3 from PIL import Image, ImageOps
4
5
6 def add_border(input_image, output_image, border):
7     img = Image.open(input_image)
8
9     if isinstance(border, int) or isinstance(border, tuple):
10         bimg = ImageOps.expand(img, border=border)
11     else:
12         raise RuntimeError('Border is not an integer or tuple!')
13
14     bimg.save(output_image)
15
16 if __name__ == '__main__':
17     in_img = 'butterfly_grey.jpg'
```

```
18  
19     add_border(in_img, output_image='butterfly_border.jpg',  
20                 border=100)
```

The `add_border()` function takes in 3 arguments:

- `input_image` - the image you want to add a border to
- `output_image` - the image with the new border applied
- `border` - the amount of border to apply in pixels

In this code, you tell Pillow that you want to add a 100 pixel border to the photo that you pass in. When you pass in an integer, that integer is used for the border on all four sides. The default color of the border is black. The key method here is `expand()`, which takes in the image object and the border amount.

When you run this code, you will end up with this lovely result:

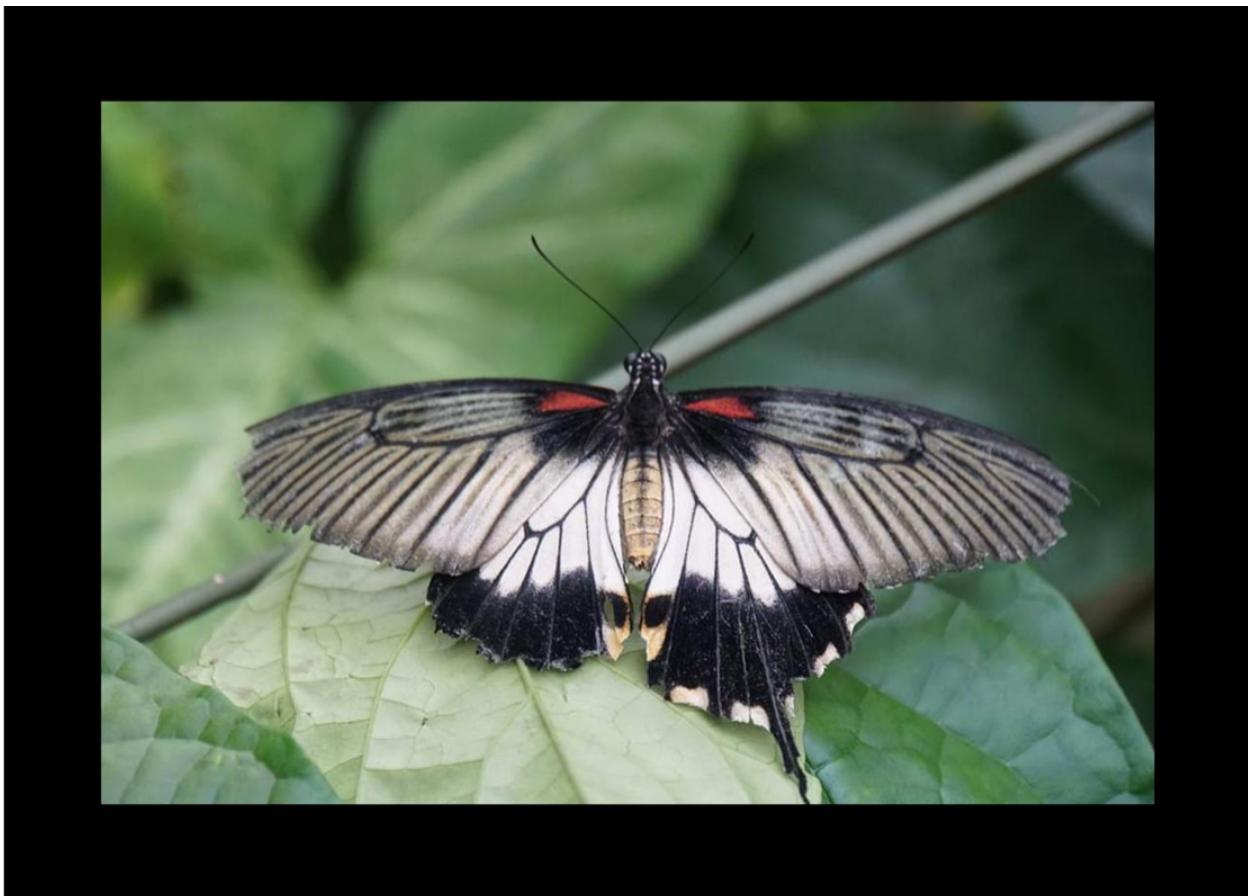


Fig. 41-10: Butterfly with a Border

You can pass in a tuple of values to make the border different widths. For example, if you passed in `(10, 50)`, that would add a 10-pixel border on the left and right sides of the images and a 50-pixel border to the top and bottom. Try doing that with the code above and re-running it. If you do, you'll get the following:

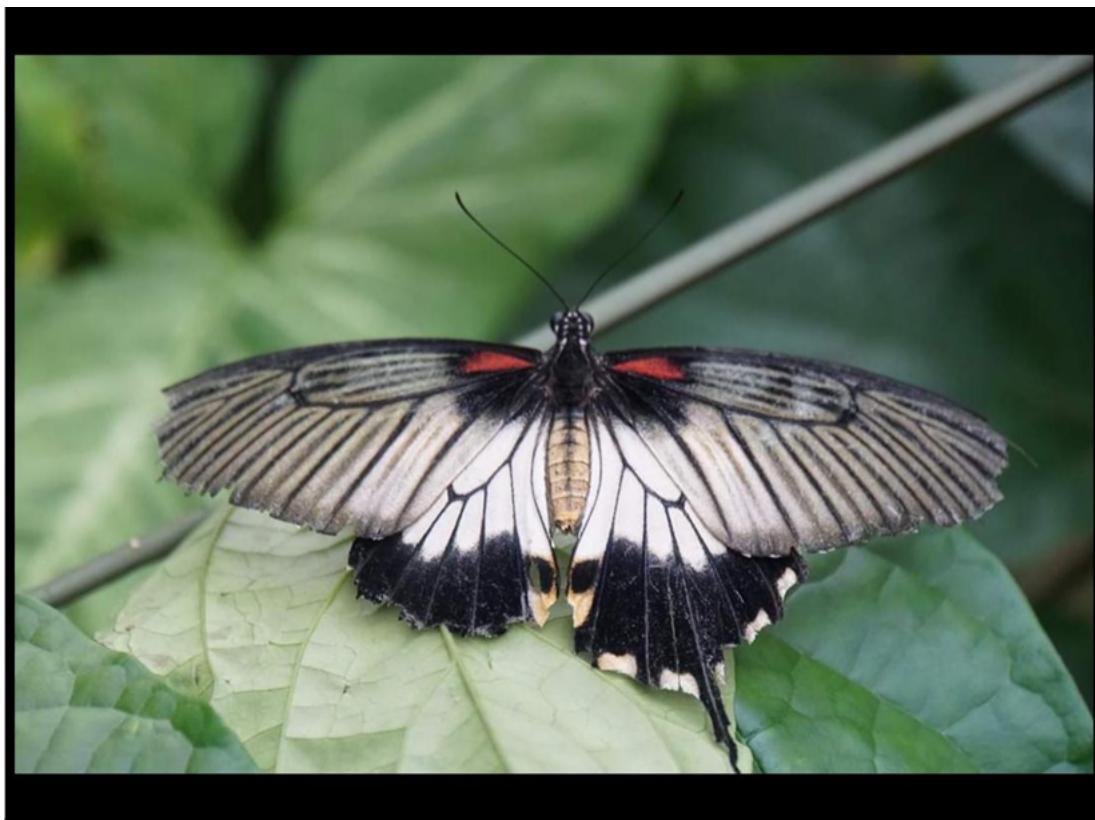


Fig. 41-11: Butterfly with Border of Varying Widths

Isn't that nice? If you want to get really fancy, you can pass in different values for all four sides of the image. But there probably aren't very many use-cases where that makes sense.

Having a black border is nice and all, but sometimes you'll want to add a little pizazz to your picture. You can change that border color by passing in the `fill` argument to `expand()`. This argument takes in a named color or an RGB color.

Create a new file named `colored_border.py` and add this code to it:

```
1 # colored_border.py
2
3 from PIL import Image, ImageOps
4
5 def add_border(input_image, output_image, border, color=0):
6     img = Image.open(input_image)
7
8     if isinstance(border, int) or isinstance(
9         border, tuple):
10        bimg = ImageOps.expand(img,
11                               border=border,
12                               fill=color)
13    else:
14        msg = 'Border is not an integer or tuple!'
15        raise RuntimeError(msg)
16
17    bimg.save(output_image)
18
19 if __name__ == '__main__':
20     in_img = 'butterfly_grey.jpg'
21
22     add_border(in_img,
23                 output_image='butterfly_border_red.jpg',
24                 border=100,
25                 color='indianred')
```

Now your `add_border()` function takes in a `color` argument, which you pass on to the `expand()` method. When you run this code, you'll see this for your result:

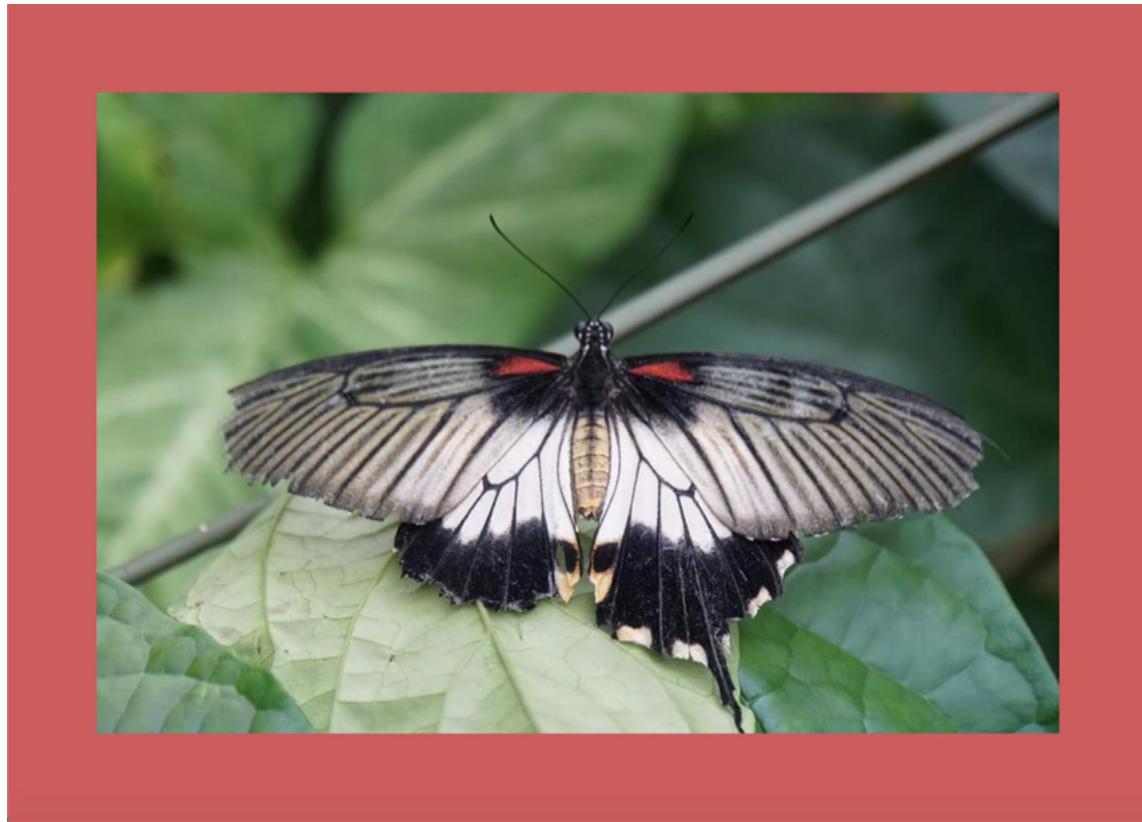


Fig. 41-12: Butterfly with Colored Border

That looks pretty nice. You can experiment around with different colors or apply your own favorite color as the border.

The next item on your Pillow tour is to learn how to resize images!

Resizing Images

Resizing images with Pillow is fairly simple. You will be using the `resize()` method which takes in a tuple of integers that are used to resize the image. To see how this works, you'll be using this lovely shot of a lizard:



Fig. 41-13: Lizard on a Wall

Now that you have a photo, go ahead and create a new file named `resize_image.py` and put this code in it:

```
1 # resize_image.py
2
3 from PIL import Image
4
5 def resize_image(input_image_path, output_image_path, size):
6     original_image = Image.open(input_image_path)
7     width, height = original_image.size
8     print(f'The original image size is {width} wide x {height} '
9           f'high')
10
11    resized_image = original_image.resize(size)
12    width, height = resized_image.size
13    print(f'The resized image size is {width} wide x {height} '
14          f'high')
15    resized_image.show()
16    resized_image.save(output_image_path)
17
18 if __name__ == '__main__':
19     resize_image(
20         input_image_path='lizard.jpg',
21         output_image_path='lizard_small.jpg',
22         size=(800, 400),
23     )
```

Here you pass in the lizard photo and tell Pillow to resize it to 600 x 400. When you run this code, the output will tell you that the original photo was 1191 x 1141 pixels before it resizes it for you.

The result of running this code looks like this:

Fig. 41-14: Lizard Resized

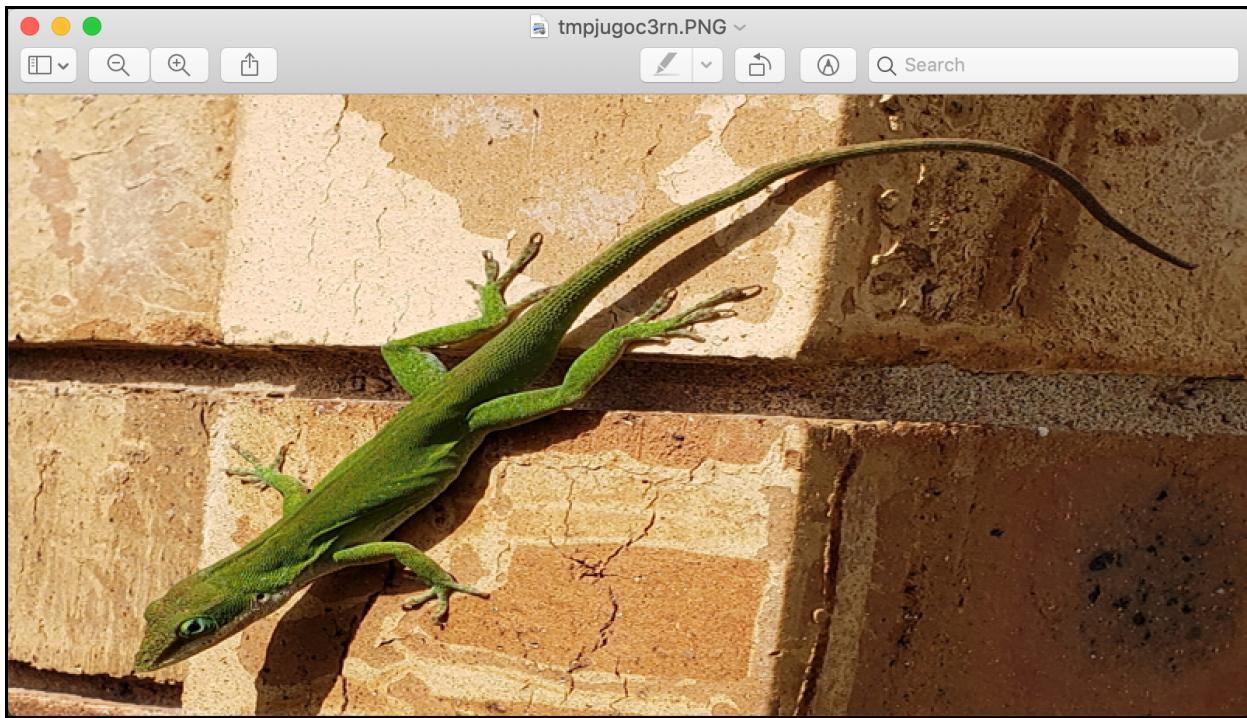


Fig. 41-14: Lizard Resized

Well, that looks a bit odd! Pillow doesn't actually do any scaling when it resizes the image. Instead, Pillow will stretch or contort your image to fit the values you tell it to use.

What you want to do is **scale** the image. To make that work, you need to create a new file named `scale_image.py` and add some new code to it. Here's the code you need:

```
1 # scale_image.py
2
3 from PIL import Image
4
5 def scale_image(
6     input_image_path,
7     output_image_path,
8     width=None,
9     height=None
10):
11     original_image = Image.open(input_image_path)
12     w, h = original_image.size
13     print(f'The original image size is {w} wide x {h} '
14           'high')
15
16     if width and height:
17         max_size = (width, height)
```

```
18     elif width:
19         max_size = (width, h)
20     elif height:
21         max_size = (w, height)
22     else:
23         # No width or height specified
24         raise ValueError('Width or height required!')
25
26     original_image.thumbnail(max_size, Image.ANTIALIAS)
27     original_image.save(output_image_path)
28
29     scaled_image = Image.open(output_image_path)
30     width, height = scaled_image.size
31     print(f'The scaled image size is {width} wide x {height} '
32           'high')
33
34
35 if __name__ == '__main__':
36     scale_image(
37         input_image_path='lizard.jpg',
38         output_image_path='lizard_scaled.jpg',
39         width=800,
40     )
```

This time around, you let the user specify both the width and height. If the user specifies a width, a height, or both, then the conditional statement uses that information to create a `max_size`. Once it has the `max_size` value calculated, you pass that to `thumbnail()` and save the result. If the user specifies both values, `thumbnail()` will maintain the aspect ratio correctly when resizing.

When you run this code, you will find that the result is a smaller version of the original image and that it now maintains its aspect ratio.

Wrapping Up

Pillow is very useful for working with images using Python. In this chapter, you learned how to do the following:

- Open Images
- Crop Images
- Use Filters
- Add Borders
- Resize Images

You can do much more with Pillow than what is shown here. For example, you can do various image enhancements, like changing the contrast or brightness of an image. Or you could composite multiple images together. There are many other applications that you can use Pillow for. You should definitely check the package out and read its documentation to learn more.

Review Questions

1. How do you get the width and height of a photo using Pillow?
2. Which method do you use to apply a border to an image?
3. How do you resize an image with Pillow while maintaining its aspect ratio?

Chapter 42 - How to Create a Graphical User Interface

When you first get started as a programmer or software developer, you usually start by writing code that prints to your console or standard out. A lot of students are also starting out by writing front-end programs, which are typically websites written with HTML, JavaScript and CSS. However, most beginners do not learn how to create a graphical user interface until much later in their classwork.

Graphical user interfaces (GUI) are programs that are usually defined as created for the desktop. The desktop refers to Windows, Linux and MacOS. It could be argued that GUIs are also created for mobile and web as well though. For the purposes of this chapter, you will learn about creating desktop GUIs. The concepts you learn in this chapter can be applied to mobile and web development to some degree as well.

A graphical user interface is made up of some kind of window that the user interacts with. The window holds other shapes inside it. These consist of buttons, text, pictures, tables, and more. Collectively, these items are known as “widgets”.

There are many different GUI toolkits for Python. Here is a list of some of the most popular:

- Tkinter
- wxPython
- PyQt
- Kivy

You will be learning about wxPython in this chapter. The reason that wxPython was chosen is that the author has more experience with it than any other and wxPython has a very friendly and helpful community.

In this chapter, you will be learning:

- Learning About Event Loops
- How to Create Widgets
- How to Lay Out Your Application
- How to Add Events
- How to Create an Application

This chapter does not attempt to cover everything there is to know about wxPython. However, you will learn enough to see the power of wxPython as well as discover how much fun it is to create a desktop GUI of your very own.

Note: Some of the examples in this chapter come from my book, **Creating GUI Applications with wxPython**.

Let's get started!

Installing wxPython

Installing wxPython is usually done with pip. If you are installing on Linux, you may need to install some prerequisites before installing wxPython. You can see the most up-to-date set of requirements on the wxPython Github page here:

- <https://github.com/wxWidgets/Phoenix#prerequisites>

On Mac OSX, you may need the XCode compiler to install wxPython.

Here is the command you would use to install wxPython using pip:

```
1 python -m pip install wxpython
```

Assuming everything worked, you should now be able to use wxPython!

Learning About Event Loops

Before you get started, there is one other item that you need to know about. In the introduction, you learned what widgets are. But when it comes to creating GUI programs, you need to understand that they use **events** to tell the GUI what to do. Unlike a command-line application, a GUI is basically an infinite loop, waiting for the user to do something, like click a button or press a key on the keyboard.

When the user does something like that, the GUI receives an event. Button events are usually connected to `wx.EVT_BUTTON`, for example. Some books call this **event-driven programming**. The overarching process is called the **event loop**.

You can think of it like this:

1. The GUI waits for the user to do something
2. The user does something (clicks a button, etc)
3. The GUI responds somehow
4. Go back to step 1

The user can stop the event loop by exiting the program.

Now that you have a basic understanding of event loops, it's time to learn how to write a simple prototype application!

How to Create Widgets

Widgets are the building blocks of your application. You start out with top-level widgets, such as a `wx.Frame` or a `wx.Dialog`. These widgets can contain other widgets, like buttons and labels. When you create a frame or dialog, it includes a title bar and the minimize, maximize, and exit buttons. Note that when using wxPython, most widgets and attributes are pre-fixed with `wx`.

To see how this all works, you should create a little “Hello World” application. Go ahead and create a new file named `hello_wx.py` and add this code to it:

```
1 # hello_wx.py
2
3 import wx
4
5 app = wx.App(False)
6 frame = wx.Frame(parent=None, title='Hello World')
7 frame.Show()
8 app.MainLoop()
```

Here you import `wx`, which is how you access wxPython in your code. Then you create an instance of `wx.App()`, which is your Application object. There can only be one of these in your application. It creates and manages your event loop for you. You pass in `False` to tell it not to redirect standard out. If you set that to `True`, then standard out is redirected to a new window. This can be useful when debugging, but should be disabled in production applications.

Next, you create a `wx.Frame()` where you set its `parent` to `None`. This tells wxPython that this frame is a top-level window. If you create all your frames without a parent, then you will need to close all the frames to end the program. The other parameter that you set is the `title`, which will appear along the top of your application’s window.

The next step is to `Show()` the frame, which makes it visible to the user. Finally, you call `MainLoop()` which starts the event loop and makes your application work. When you run this code, you should see something like this:



Fig. 42-1: Hello World in wxPython

When working with wxPython, you will actually be sub-classing `wx.Frame` and quite a few of the other widgets. Create a new file named `hello_wx_class.py` and put this code into it:

```
1 # hello_wx_class.py
2
3 import wx
4
5 class MyFrame(wx.Frame):
6
7     def __init__(self):
8         super().__init__(None, title='Hello World')
9         self.Show()
10
11 if __name__ == '__main__':
12     app = wx.App(False)
13     frame = MyFrame()
14     frame.Show()
15     app.MainLoop()
```

This code does the same thing as the previous example, but this time you are creating your own version of the `wx.Frame` class.

When you create an application with multiple widgets in it, you will almost always have a `wx.Panel` as the sole child widget of the `wx.Frame`. Then the `wx.Panel` widget is used to contain the other

widgets. The reason for this is that `wx.Panel` provides the ability to tab between the widgets, which is something that does not work if you make all the widget children of `wx.Frame`.

So, for a final “Hello World” example, you can add a `wx.Panel` to the mix. Create a file named `hello_with_panel.py` and add this code:

```
1 # hello_with_panel.py
2
3 import wx
4
5 class MyPanel(wx.Panel):
6
7     def __init__(self, parent):
8         super().__init__(parent)
9         button = wx.Button(self, label='Press Me')
10
11 class MyFrame(wx.Frame):
12
13     def __init__(self):
14         super().__init__(None, title='Hello World')
15         panel = MyPanel(self)
16         self.Show()
17
18 if __name__ == '__main__':
19     app = wx.App(redirect=False)
20     frame = MyFrame()
21     app.MainLoop()
```

In this code, you create two classes. One sub-classes `wx.Panel` and adds a button to it using `wx.Button`. The `MyFrame()` class is almost the same as the previous example except that you now create an instance of `MyPanel()` in it. Note that you are passing `self` to `MyPanel()`, which is telling wxPython that the frame is now the parent of the panel widget.

When you run this code, you will see the following application appear:



Fig. 42-2: Hello World in wxPython with wx.Panel

This example shows that when you add a child widget, like a button, it will automatically appear at the top left of the application. The `wx.Panel` is an exception when it is the only child widget of a `wx.Frame`. In that case, the `wx.Panel` will automatically expand to fill the `wx.Frame`.

What do you think happens if you add multiple widgets to the panel though? Let's find out! Create a new file named `stacked_buttons.py` and add this code:

```
1 # stacked_buttons.py
2
3 import wx
4
5 class MyPanel(wx.Panel):
6
7     def __init__(self, parent):
8         super().__init__(parent)
9         button = wx.Button(self, label='Press Me')
10        button2 = wx.Button(self, label='Press Me too')
11        button3 = wx.Button(self, label='Another button')
12
13 class MyFrame(wx.Frame):
14
15     def __init__(self):
16         super().__init__(None, title='Hello World')
17         panel = MyPanel(self)
```

```
18     self.Show()  
19  
20  
21 if __name__ == '__main__':  
22     app = wx.App(redirect=False)  
23     frame = MyFrame()  
24     app.MainLoop()
```

Now you have three buttons as children of the panel. Try running this code to see what happens:

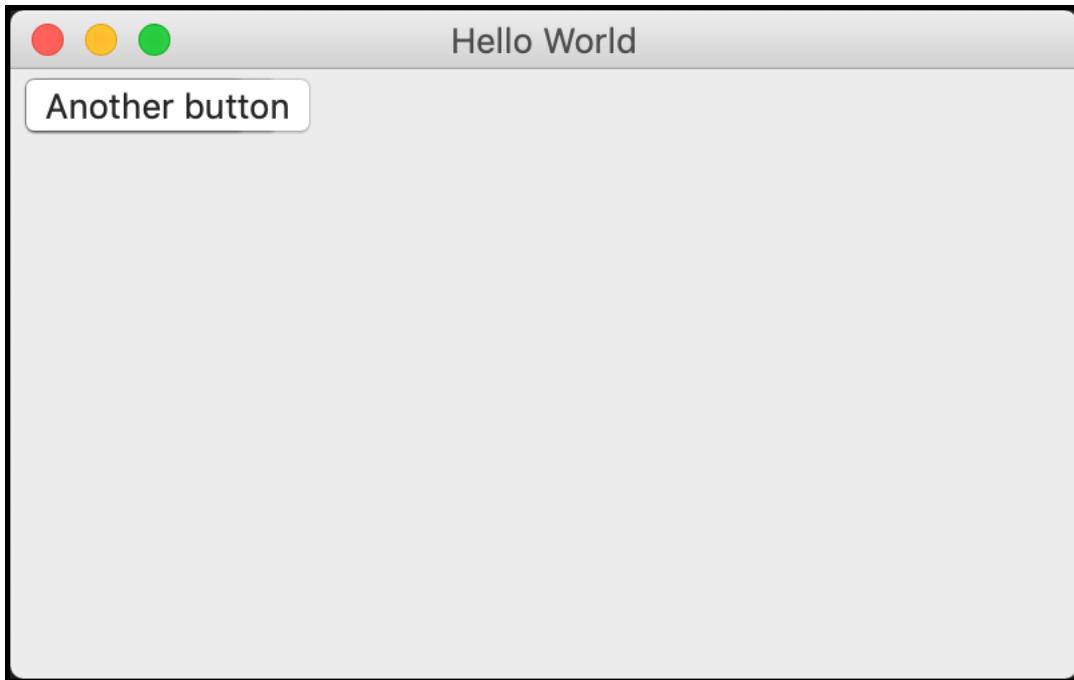


Fig. 42-3: Stacked Buttons

Oops! You only see one button, which happens to be the last one you created. What happened here? You didn't tell the buttons where to go, so they all went to the default location, which is the upper left corner of the widget. In essence, the widgets are now stacked on top of each other.

Let's find out how you can fix that issue in the next section!

How to Lay Out Your Application

You have two options when it comes to laying out your application:

- Absolute positioning
- Sizers

In almost all cases, you will want to use **Sizers**. If you want to use absolute positioning, you can use the widget's pos parameter and give it a tuple that specifies the x and y coordinate in which to place the widget. Absolute positioning can be useful when you need pixel perfect positioning of widgets. However, when you use absolute positioning, your widgets cannot resize or move when the window they are in is resized. They are static at that point.

The solution to those issues is to use Sizers. They can handle how your widgets should resize and adjust when the application size is changed. There are several different sizers that you can use, such as `wx.BoxSizer`, `wx.GridSizer`, and more.

The wxPython documentation explains how they work in detail here:

- https://docs.wxpython.org/sizers_overview.html

Let's take that code from before and reduce it down to two buttons and add a Sizer. Create a new file named `sizer_with_two_widgets.py` and put this code into it:

```
1 # sizer_with_two_widgets.py
2
3 import wx
4
5 class MyPanel(wx.Panel):
6
7     def __init__(self, parent):
8         super().__init__(parent)
9
10        button = wx.Button(self, label='Press Me')
11        button2 = wx.Button(self, label='Second button')
12
13        main_sizer = wx.BoxSizer(wx.HORIZONTAL)
14        main_sizer.Add(button, proportion=1,
15                      flag=wx.ALL | wx.CENTER | wx.EXPAND,
16                      border=5)
17        main_sizer.Add(button2, 0, wx.ALL, 5)
18        self.SetSizer(main_sizer)
19
20 class MyFrame(wx.Frame):
21
22     def __init__(self):
23         super().__init__(None, title='Hello World')
24         panel = MyPanel(self)
25         self.Show()
26
27 if __name__ == '__main__':
```

```
28     app = wx.App(redirect=False)
29     frame = MyFrame()
30     app.MainLoop()
```

In this example, you create a `wx.BoxSizer`. A `wx.BoxSizer` can be set to add widgets horizontally (left-to-right) or vertically (top-to-bottom). For your code, you set the sizer to add widgets horizontally by using the `wx.HORIZONTAL` constant. To add a widget to a sizer, you use the `Add()` method.

The `Add()` method takes up to five arguments:

- `window` - the widget to add
- `proportion` - tells wxPython if the widget can change its size in the same orientation as the sizer
- `flag` - one or more flags that affect the sizer's behavior
- `border` - the border width, in pixels
- `userData` - allows adding an extra object to the sizer item, which is used for subclasses of sizers.

The first button that you add to the sizer is set to a proportion of 1, which will make it expand to fill as much space in the sizer as it can. You also give it three flags:

- `wx.ALL` - add a border on all sides
- `wx.CENTER` - center the widget within the sizer
- `wx.EXPAND` - the item will be expanded as much as possible while also maintaining its aspect ratio

Finally, you add a border of five pixels. These pixels are added to the top, bottom, left, and right of the widget because you set the `wx.ALL` flag.

The second button has a proportion of 0, which means it wont expand at all. Then you tell it to add a five pixel border all around it as well. To apply the sizer to the panel, you need to call the panel's `SetSizer()` method.

When you run this code, you will see the following applications:

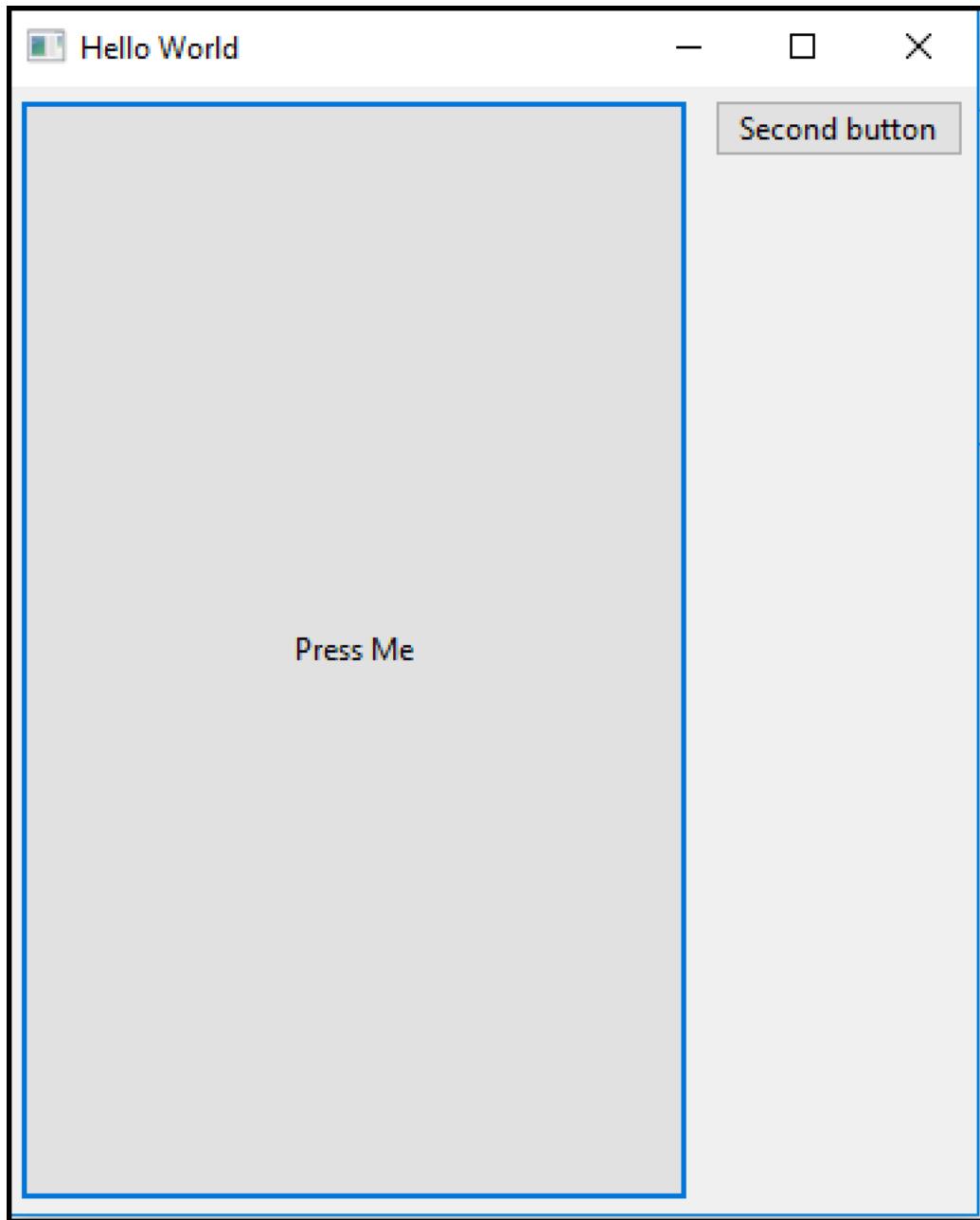


Fig. 42-4: Putting Buttons in a BoxSizer

You can see how the various flags have affected the appearance of the buttons. Note that on MacOS, `wx.Button` cannot be stretched, so if you want to do that on a Mac, you would need to use a generic button from `wx.lib.buttons` instead. Generic buttons are usually made with Python and do not wrap the native widget.

Now let's move on and learn how events work!

How to Add Events

So far you have created a couple of neat little applications with buttons, but the buttons don't do anything when you click on them. Why is that? Well, when you are writing a GUI application, you need to tell it what to do when something happens. That "something" that happens is known as an **event**.

To hook an event to a widget, you will need to use the `Bind()` method. Some widgets have multiple events that can be bound to them while others have only one or two. The `wx.Button` can be bound to `wx.EVT_BUTTON` only.

Let's copy the code from the previous example and paste it into a new file named `button_events.py`. Then update it to add events like this:

```
1 # button_events.py
2
3 import wx
4
5 class MyPanel(wx.Panel):
6
7     def __init__(self, parent):
8         super().__init__(parent)
9
10        button = wx.Button(self, label='Press Me')
11        button.Bind(wx.EVT_BUTTON, self.on_button1)
12        button2 = wx.Button(self, label='Second button')
13        button2.Bind(wx.EVT_BUTTON, self.on_button2)
14
15        main_sizer = wx.BoxSizer(wx.HORIZONTAL)
16        main_sizer.Add(button, proportion=1,
17                        flag=wx.ALL | wx.CENTER | wx.EXPAND,
18                        border=5)
19        main_sizer.Add(button2, 0, wx.ALL, 5)
20        self.SetSizer(main_sizer)
21
22    def on_button1(self, event):
23        print('You clicked the first button')
24
25    def on_button2(self, event):
26        print('You clicked the second button')
27
28 class MyFrame(wx.Frame):
29
```

```
30     def __init__(self):
31         super().__init__(None, title='Hello World')
32         panel = MyPanel(self)
33         self.Show()
34
35 if __name__ == '__main__':
36     app = wx.App(redirect=False)
37     frame = MyFrame()
38     app.MainLoop()
```

Here you call `Bind()` for each of the buttons in turn. You bind the button to `wx.EVT_BUTTON`, which will fire when the user presses a button. The second argument to `Bind()` is the method that should be called when you click the button.

If you run this code, the GUI will still look the same. However, when you press the buttons, you should see different messages printed to stdout (i.e. your terminal or console window). Give it a try and see how it works.

Now let's go ahead and write a simple application!

How to Create an Application

The first step in creating an application is to come up with an idea. You could try to copy something simple like Microsoft Paint or Notepad. You will quickly find that they aren't so easy to emulate as you would think, though! So instead, you will create a simple application that can load and display a photo.

When it comes to creating a GUI application, it is a good idea to think about what it will look like. If you enjoy working with pencil and paper, you could draw a sketch of what your application will look like. There are many software applications you can use to draw with or create simple mock-ups. To simulate a Sizer, you can draw a box.

Here is a mockup of what the finished application should look like:

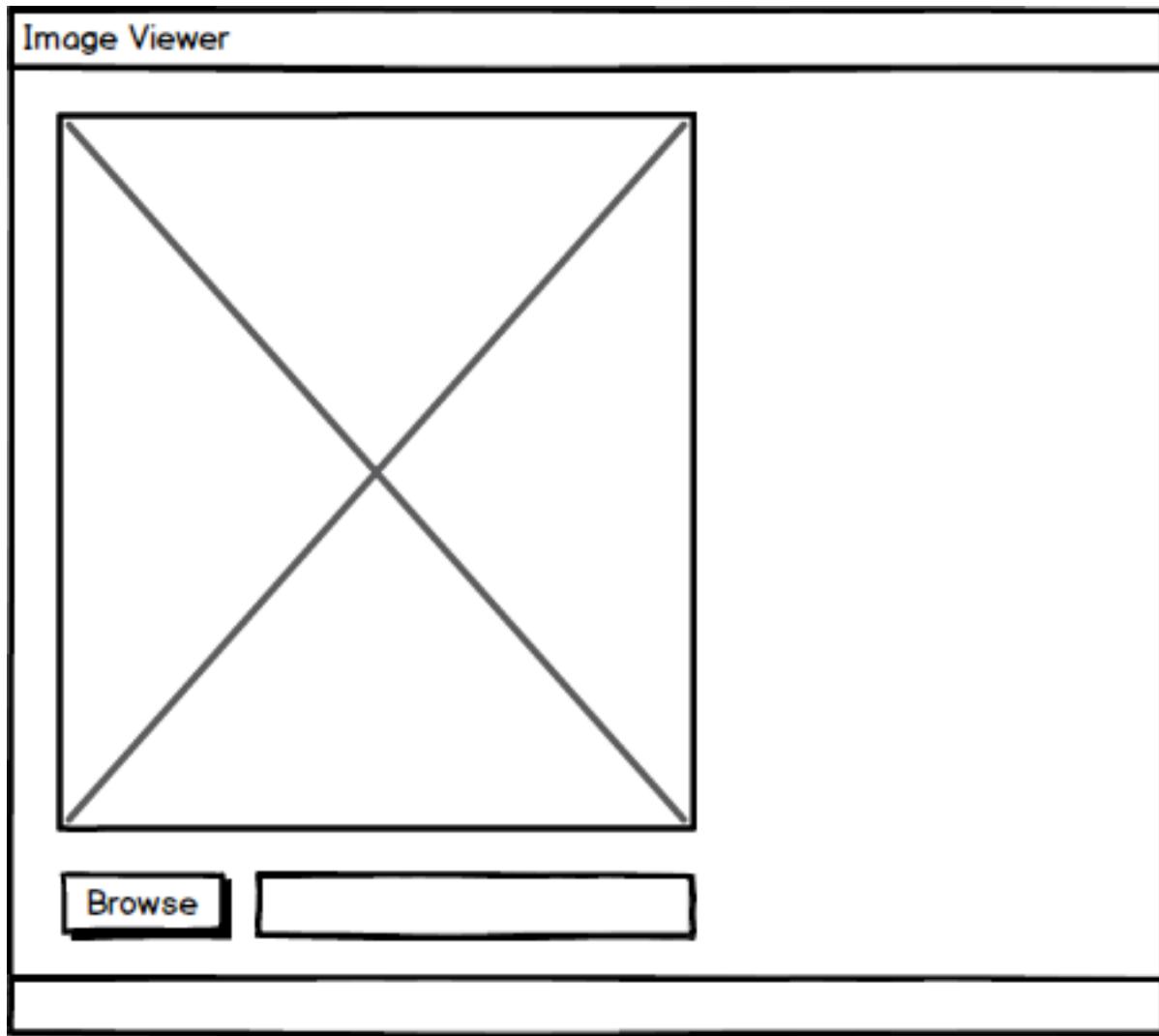


Fig. 42-5: Image Viewer Mockup

Now you have a goal in mind. This allows you to think about how you might lay out the widgets. Go ahead and create a new file named `image_viewer.py` and add the following code to it:

```
1 # image_viewer.py
2
3 import wx
4
5 class ImagePanel(wx.Panel):
6
7     def __init__(self, parent, image_size):
8         super().__init__(parent)
9
10        img = wx.Image(*image_size)
```

```
11     self.image_ctrl = wx.StaticBitmap(self,
12                                     bitmap=wx.Bitmap(img))
13     browse_btn = wx.Button(self, label='Browse')
14
15     main_sizer = wx.BoxSizer(wx.VERTICAL)
16     main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
17     main_sizer.Add(browse_btn)
18     self.SetSizer(main_sizer)
19     main_sizer.Fit(parent)
20     self.Layout()
21
22 class MainFrame(wx.Frame):
23
24     def __init__(self):
25         super().__init__(None, title='Image Viewer')
26         panel = ImagePanel(self, image_size=(240,240))
27         self.Show()
28
29 if __name__ == '__main__':
30     app = wx.App(redirect=False)
31     frame = MainFrame()
32     app.MainLoop()
```

Here you create a new class named `ImagePanel()` that will hold all your widgets. Inside it, you have a `wx.Image`, which you will use to hold the photo in memory in an object that wxPython can work with. To display that photo to the user, you use `wx.StaticBitmap`. The other widget you need is the familiar `wx.Button`, which you will use to browse to the photo to load.

The rest of the code lays out the widgets using a vertically oriented `wx.BoxSizer`. You use the sizer's `Fit()` method to try to make the frame "fit" the widgets. What that means is that you want the application to not have a lot of white space around the widgets.

When you run this code, you will end up with the following user interface:

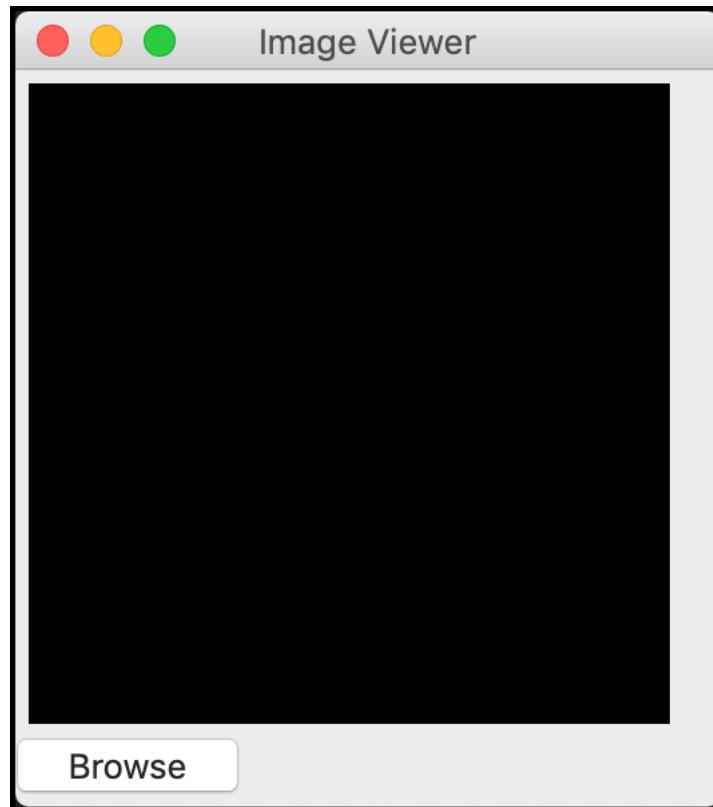


Fig. 42-6: Image Viewer Demo

That looks almost right. It looks like you forgot to add the text entry widget to the right of the browse button, but that's okay. The objective was to try and get a close approximation to what the application would look like in the end, and this looks pretty good. Of course, none of the widgets actually do anything yet.

Your next step is to update the code so it works. Copy the code from the previous example and make a new file named `image_viewer_working.py`. There will be significant updates to the code, which you will learn about soon. But first, here is the full change in its entirety:

```
1 # image_viewer_working.py
2
3 import wx
4
5 class ImagePanel(wx.Panel):
6
7     def __init__(self, parent, image_size):
8         super().__init__(parent)
9         self.max_size = 240
10
11     img = wx.Image(*image_size)
12     self.image_ctrl = wx.StaticBitmap(self,
```

```
13                         bitmap=wx.Bitmap(img))
14
15         browse_btn = wx.Button(self, label='Browse')
16         browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)
17
18         self.photo_txt = wx.TextCtrl(self, size=(200, -1))
19
20         main_sizer = wx.BoxSizer(wx.VERTICAL)
21         hsizer = wx.BoxSizer(wx.HORIZONTAL)
22
23         main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
24         hsizer.Add(browse_btn, 0, wx.ALL, 5)
25         hsizer.Add(self.photo_txt, 0, wx.ALL, 5)
26         main_sizer.Add(hsizer, 0, wx.ALL, 5)
27
28         self.SetSizer(main_sizer)
29         main_sizer.Fit(parent)
30         self.Layout()
31
32     def on_browse(self, event):
33         """
34             Browse for an image file
35             @param event: The event object
36         """
37
38         wildcard = "JPEG files (*.jpg)|*.jpg"
39         with wx.FileDialog(None, "Choose a file",
40                            wildcard=wildcard,
41                            style=wx.ID_OPEN) as dialog:
42             if dialog.ShowModal() == wx.ID_OK:
43                 self.photo_txt.SetValue(dialog.GetPath())
44                 self.load_image()
45
46     def load_image(self):
47         """
48             Load the image and display it to the user
49         """
50
51         filepath = self.photo_txt.GetValue()
52         img = wx.Image(filepath, wx.BITMAP_TYPE_ANY)
53
54         # scale the image, preserving the aspect ratio
55         W = img.GetWidth()
56         H = img.GetHeight()
57         if W > H:
```

```
56         NewW = self.max_size
57         NewH = self.max_size * H / W
58     else:
59         NewH = self.max_size
60         NewW = self.max_size * W / H
61     img = img.Scale(NewW,NewH)
62
63     self.image_ctrl.SetBitmap(wx.Bitmap(img))
64     self.Refresh()
65
66 class MainFrame(wx.Frame):
67
68     def __init__(self):
69         super().__init__(None, title='Image Viewer')
70         panel = ImagePanel(self, image_size=(240,240))
71         self.Show()
72
73 if __name__ == '__main__':
74     app = wx.App(redirect=False)
75     frame = MainFrame()
76     app.MainLoop()
```

This change is pretty long. To make things easier, you will go over each change in its own little chunk. The changes all occurred in the `ImagePanel` class, so you will go over the changes in each of the methods in turn, starting with the constructor below:

```
1 def __init__(self, parent, image_size):
2     super().__init__(parent)
3     self.max_size = 240
4
5     img = wx.Image(*image_size)
6     self.image_ctrl = wx.StaticBitmap(self,
7                                     bitmap=wx.Bitmap(img))
8
9     browse_btn = wx.Button(self, label='Browse')
10    browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)
11
12    self.photo_txt = wx.TextCtrl(self, size=(200, -1))
13
14    main_sizer = wx.BoxSizer(wx.VERTICAL)
15    hsizer = wx.BoxSizer(wx.HORIZONTAL)
16
17    main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
```

```

18     hsizer.Add(browse_btn, 0, wx.ALL, 5)
19     hsizer.Add(self.photo_txt, 0, wx.ALL, 5)
20     main_sizer.Add(hsizer, 0, wx.ALL, 5)
21
22     self.SetSizer(main_sizer)
23     main_sizer.Fit(parent)
24     self.Layout()

```

There are a few minor changes here. The first one is that you added a `max_size` for the image. Then you hooked up an event to the the browse button. This button will now call `on_browse()` when it is clicked.

The next change is that you added a new widget, a `wx.TextCtrl` to be precise. You stored a reference to that widget in `self.photo_txt`, which will allow you to extract the path to the photo later.

The final change is that you now have two sizers. One is horizontal and the other remains vertical. The horizontal sizer is for holding the browse button and your new text control widgets. This allows your to place them next to each other, left-to-right. Then you add the horizontal sizer itself to the vertical `main_sizer`.

Now let's see how `on_browse()` works:

```

1 def on_browse(self, event):
2     """
3     Browse for an image file
4     @param event: The event object
5     """
6     wildcard = "JPEG files (*.jpg)|*.jpg"
7     with wx.FileDialog(None, "Choose a file",
8                         wildcard=wildcard,
9                         style=wx.ID_OPEN) as dialog:
10        if dialog.ShowModal() == wx.ID_OK:
11            self.photo_txt.SetValue(dialog.GetPath())
12            self.load_image()

```

Here you create a `wildcard` which is used by the `wx.FileDialog` to filter out all the other files types except the JPEG format. Next, you create the `wx.FileDialog`. When you do that, you set its parent to `None` and give it a simple title. You also set the `wildcard` and the `style`. `style` is an open file dialog instead of a save file dialog.

Then you show your dialog modally. What that means is that the dialog will appear over your main application and prevent you from interacting with the main application until you have accepted or dismissed the file dialog. If the user presses the OK button, then you will use `GetPath()` to get the path of the selected file and set the text control to that path. This effectively saves off the photo's path so you can use it later.

Lastly, you call `load_image()` which will load the image into wxPython and attempt to show it. You can find out how by reading the following code:

```
1 def load_image(self):
2     """
3         Load the image and display it to the user
4     """
5
6     filepath = self.photo_txt.GetValue()
7     img = wx.Image(filepath, wx.BITMAP_TYPE_ANY)
8
9     # scale the image, preserving the aspect ratio
10    W = img.GetWidth()
11    H = img.GetHeight()
12    if W > H:
13        NewW = self.max_size
14        NewH = self.max_size * H / W
15    else:
16        NewH = self.max_size
17        NewW = self.max_size * W / H
18    img = img.Scale(NewW,NewH)
19
20    self.image_ctrl.SetBitmap(wx.Bitmap(img))
21    self.Refresh()
```

The first step in this method is to extract the `filepath` from the text control widget. Then you pass that path along to a new instance of `wx.Image`. This will load the image into wxPython for you. Next, you get the width and height from the `wx.Image` object and use the `max_size` value to resize the image while maintaining its aspect ratio. You do this for two reasons. The first is because if you don't, the image will get stretched out or warped. The second is that most images at full resolution won't fit on-screen, so they need to be resized.

Once you have the new width and height, you `scale()` the image down appropriately. Then you call your `wx.StaticBitmap` control's `SetBitmap()` method to update it to the new image that you loaded. Finally, you call `Refresh()`, which will force the bitmap widget to redraw with the new image in it.

Here it is with a butterfly photo loaded in it:

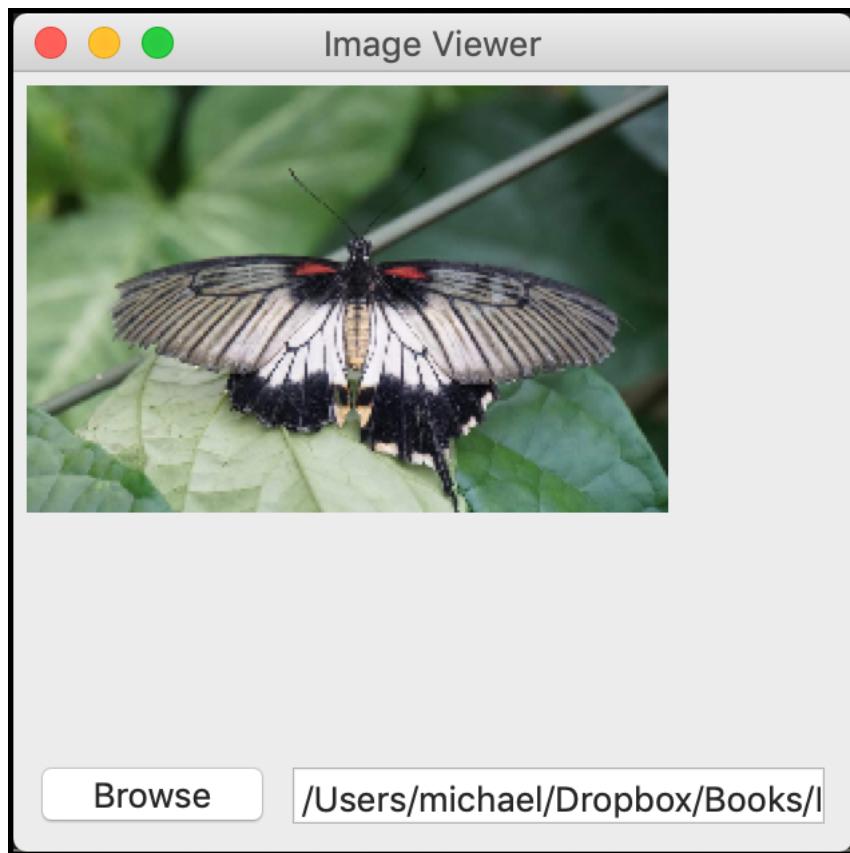


Fig. 42-7: Image Viewer Working

Now you have a fully-functional application that can load JPEG photos. You can update the application to load other image types if you'd like. The wxPython toolkit uses Pillow, so it will load the same types of image file types that Pillow itself can load.

Wrapping Up

The wxPython toolkit is extremely versatile. It comes with many, many widgets built-in and also includes a wonderful demo package. The demo package will help you learn how to use the widgets in your own code. You are only limited by your imagination.

In this chapter, you learned about the following topics:

- Learning About Event Loops
- How to Create Widgets
- How to Lay Out Your Application
- How to Add Events
- How to Create an Application

You can take the code and the concepts in this code and add new features or create brand new applications. If you need ideas, you can check out some of the applications on your own computer or phone. You can also check out my book, **Creating GUI Applications with wxPython**, which has lots of fun little applications you can create and expand upon.

Review Questions

1. What is a GUI?
2. What is an event loop?
3. How do you lay out widgets in your application?

Part IV - Distributing Your Code

If you have reached this part of the book, then you should now have a good understanding of the Python programming language. You may be excited to create an application and share it with your friends. If that is the case, then you are in the right place.

The last section of this book will teach you how to share your code. In it, you will learn how to do the following:

- Chapter 43 - How to Create a Python package
- Chapter 44 - How to Create an Exe for Windows
- Chapter 45 - How to Create an Installer for Windows
- Chapter 46 - How to Create an “exe” for Mac

A Python package is a module or group of modules that you install using pip to enhance your code. Windows and Mac have good solutions for creating a distributable file that your friends and family can use to install your program or run your program without installing Python or any other dependencies.

Let's get started!

Chapter 43 - How to Create a Python Package

When you create a Python file, you are creating a Python module. Any Python file that you create can be imported by another Python script. Thus, by definition, it is also a Python module. If you have two or more related Python files, then you may have a Python package.

Some organizations keep all their code to themselves. This is known as **closed-source**. Python is an **open-source** language and most of the Python modules and packages that you can get from the **Python Package Index (PyPI)** are all free and open-source as well. One of the quickest ways to share your package or module is to upload it to the Python Package Index or Github or both.

In this chapter, you will learn about the following topics:

- Creating a Module
- Creating a Package
- Packaging a Project for PyPI
- Creating Project Files
- Creating `setup.py`
- Uploading to PyPI

The first step in the process is to understand what creating a reusable module looks like. Let's get started!

Creating a Module

Any Python file you create is a module that you can import. You can try it out with some of the examples from this book by adding a new file to any of the chapter's code folders and attempt to import one of the modules in there. For example, if you have a Python file named `a.py` and then create a new file named `b.py`, you can import `a` into `b` through the use of `import a`.

Of course, that's a silly example. Instead, you will create a simple module that has some basic arithmetic functions in it. You can name the file `arithmetic.py` and add this code to it:

```
1 # arithmetic.py
2
3 def add(x, y):
4     return x + y
5
6 def divide(x, y):
7     return x / y
8
9 def multiply(x, y):
10    return x * y
11
12 def subtract(x, y):
13    return x - y
```

This code is very naive. You have no error handling at all, for example. What that means is that you could divide by zero and cause an exception to be thrown. You could also pass incompatible types to these functions, like a string and an integer – that would cause a different kind of exception to be raised.

However, for learning purposes, this code is adequate. You can prove that it is importable by creating a test file using the knowledge you acquired in chapter 30. Create a new file named `test_arithmetic.py` and add this code to it:

```
1 # test_arithmetic.py
2
3 import arithmetic
4 import unittest
5
6 class TestArithmetic(unittest.TestCase):
7
8     def test_addition(self):
9         self.assertEqual(arithmetic.add(1, 2), 3)
10
11    def test_subtraction(self):
12        self.assertEqual(arithmetic.subtract(2, 1), 1)
13
14    def test_multiplication(self):
15        self.assertEqual(arithmetic.multiply(5, 5), 25)
16
17    def test_division(self):
18        self.assertEqual(arithmetic.divide(8, 2), 4)
19
20 if __name__ == '__main__':
21     unittest.main()
```

Now you can run this code using the following command:

```
1 $ python test_arithmetic.py
2 ....
3 -----
4 Ran 4 tests in 0.000s
5
6 OK
```

This demonstrates that you can import `arithmetic.py` as a module. These tests also show the basic functionality of the code works. You can enhance these tests by testing division by zero and mixing strings and integers. Those kinds of tests will currently fail. Once you have a failing test, you can follow the Test Driven Development methodology you learned in **chapter 30** to fix the issues.

Now let's find out how to make a Python package!

Creating a Package

A Python package is one or more files that you plan on sharing with others, usually by uploading it to the Python Package Index (PyPI). Packages are generally made by naming a directory of files rather than a file itself. Then inside of that directory you will have a special `__init__.py` file. When Python sees the `__init__.py` file, it knows that the folder is importable as a package.

There are a couple ways to transform `arithmetic.py` into a package. The simplest is to move the code from `arithmetic.py` into `arithmetic/__init__.py`:

- create the folder `arithmetic`
- move/copy `arithmetic.py` to `arithmetic/__init__.py`
- if you used “copy” in the previous step then delete `arithmetic.py`
- run `test_arithmetic.py`

That last step is extremely important! If your tests still pass then you know your conversion from a module to a package worked. To test out your package, open up a Command Prompt if you’re on Windows or a terminal if you’re on Mac or Linux. Then navigate to the folder that contains the `arithmetic` folder, but not inside of it. You should now be in the same folder as your `test_arithmetic.py` file. At this point you can run `python test_arithmetic.py` and see if your efforts were successful.

It might seem silly to simply put all your code in a single `__init__.py` file, but that actually works fine for files up to a few thousand lines.

The second way to transform `arithmetic.py` into a package is similar to the first, but involves using more files than just `__init__.py`. In real code the functions/classes/etc. in each file would be grouped

somehow – perhaps one file for all your package’s custom exceptions, one file for common utilities, and one file for the main functionality.

For our example, you’ll just split the four functions in `arithmetic.py` into their own files. Go ahead and move each function from `__init__.py` into its own file. Your folder structure should look like this:

```
1 arithmetic/
2     __init__.py
3     add.py
4     subtract.py
5     multiply.py
6     divide.py
```

For the `__init__.py` file, you can add the following code:

```
1 # __init__.py
2 from .add import add
3 from .subtract import subtract
4 from .multiply import multiply
5 from .divide import divide
```

Now that you’ve made these changes what should be your next step? Hopefully you said, “Run my tests!” If your tests still pass then you haven’t broken your API.

Right now your `arithmetic` package is only available to your other Python code if you happen to be in the same folder as your `test_arithmetic.py` file. To make it available in your Python session or in other Python code you can use Python’s `sys` module to add your package to the Python search path. The search path is what Python uses to find modules when you use the `import` keyword. You can see what paths Python searches by printing out `sys.path`.

Let’s pretend that your `arithmetic` folder is in this location: `/Users/michael/packages/arithmetic`. To add that to Python’s search path, you can do this:

```
1 import sys
2
3 sys.path.append("/Users/michael/packages/arithmetic")
4 import arithmetic
5
6 print(arithmetic.add(1, 2))
```

This will add `arithmetic` to Python’s path so you can import it and then use the package in your code. However, that’s really awkward. It would be nice if you could install your package using `pip` so you don’t have to mess around with the path all the time.

Let’s find out how to do that next!

Packaging a Project for PyPI

When it comes to creating a package for the Python Package Index (PyPI), you will need some additional files. There is a good tutorial on the process for creating and uploading a package to PyPI here:

- <https://packaging.python.org/tutorials/packaging-projects/>

The official packaging instructions recommend that you set up a directory structure like this:

```
1 my_package/
2     LICENSE
3     README.md
4     arithmetic/
5         __init__.py
6         add.py
7         subtract.py
8         multiply.py
9         divide.py
10        setup.py
11        tests/
```

The `tests` folder can be empty. This is the folder where you would include tests for your package. Most developers use Python's `unittest` or the `pytest` framework for their tests. For this example, you can leave the folder empty.

Let's move on and learn about the other files you need to create in the next section!

Creating Project Files

The `LICENSE` file is where you mention what license your package has. This tells the users of the package what they can and cannot do with your package. There are a lot of different licenses you can use. The GPL and MIT licenses are just a couple of popular examples.

The `README.md` file is a description of your project, written in Markdown. You will want to write about your project in this file and include any information about dependencies that it might need. You can give instructions for installation as well as example usage of your package. Markdown is quite versatile and even lets you do syntax highlighting!

The other file you need to supply is `setup.py`. That file is more complex, so you'll learn about that in the next section.

Creating setup.py

There is a special file named `setup.py` that is used as a build script for Python distributions. It is used by `setuptools`, which does the actual building for you. If you'd like to know more about `setuptools`, then you should check out the following:

- <https://setuptools.readthedocs.io/en/latest/>

You can use the `setup.py` to create a Python **wheel**. The wheel is a ZIP-format archive with a specially formatted name and a `.whl` extension. It contains everything necessary to install your package. You can think of it as a zipped version of your code that `pip` can unzip and install for you. The wheel follows PEP 376, which you can read about here:

- <https://www.python.org/dev/peps/pep-0376/>

Once you're done reading all that documentation (if you wanted to), you can create your `setup.py` and add this code to it:

```
1 import setuptools
2
3 with open("README.md", "r") as fh:
4     long_description = fh.read()
5
6 setuptools.setup(
7     name="arithmetic-YOUR-USERNAME-HERE", # Replace with your own username
8     version="0.0.1",
9     author="Mike Driscoll",
10    author_email="driscoll@example.com",
11    description="A simple arithmetic package",
12    long_description=long_description,
13    long_description_content_type="text/markdown",
14    url="https://github.com/driscollis/arithmetic",
15    packages=setuptools.find_packages(),
16    classifiers=[
17        "Programming Language :: Python :: 3",
18        "License :: OSI Approved :: MIT License",
19        "Operating System :: OS Independent",
20    ],
21    python_requires='>=3.6',
22 )
```

The first step in this code is to import `setuptools`. Then you read in your `README.md` file into a variable that you will use soon. The last bit is the bulk of the code. Here you call `setuptools.setup()`, which can take in quite a few different arguments. The example above is only a sampling of what you can pass to this function. To see the full listing, you'll need to go here:

- <https://packaging.python.org/guides/distributing-packages-using-setuptools/>

Most of the arguments are self-explanatory. Let's focus on the more obtuse ones. The `packages` argument is a list of packages needed for your package. In this case, you use `find_packages()` to find the necessary packages for you automatically. The `classifiers` argument is used for passing additional metadata to `pip`. For example, this code tells `pip` that the package is Python 3 compatible.

Now that you have a `setup.py`, you are ready to create a Python wheel!

Generating a Python Wheel

The `setup.py` is used to create Python wheels. It's always a good idea to make sure you have the latest version of `setuptools` and `wheel` installed, so before you create your own wheel, you should run the following command:

```
1 python3 -m pip install --user --upgrade setuptools wheel
```

This will update the packages if there is a newer version than the one you currently have installed. Now you are ready to create a wheel yourself. Open up a Command Prompt or terminal application and navigate to the folder that contains your `setup.py` file. Then run the following command:

```
1 python3 setup.py sdist bdist_wheel
```

This command will output a lot of text, but once it has finished you will find a new folder named `dist` that contains the following two files:

- `arithmetic_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl`
- `arithmetic-YOUR-USERNAME-HERE-0.0.1.tar.gz`

The `.tar.gz` is a source archive, which means it has the Python source code for your package inside of it. Your users can use the source archive to build the package on their own machines, if they need to. The `.whl` format is an archive that is used by `pip` to install your package on your user's machine.

You can install the wheel using `pip` directly, if you want to:

```
1 python -m pip install arithmetic_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl
```

But the normal method would be to upload your package to the Python Package Index (PyPI) and then install it. Let's discover how to get your amazing package on PyPI next!

Uploading to PyPI

The first step to upload a package to PyPI is to create an account on *Test PyPI*. This allows you to test that your package can be uploaded on a test server and installed from that test server. To create an account, go to the following URL and follow the steps on that page:

- <https://test.pypi.org/account/register/>

Now you need to create a PyPI API token. This will allow you to upload the package securely. To get the API token, you'll need to go here:

- <https://test.pypi.org/manage/account/#api-tokens>

You can limit a token's scope. However, you don't need to do that for this token as you are creating it for a new project. Make sure you copy the token and save it off somewhere BEFORE you close the page. Once the page is closed, you cannot retrieve the token again. You will be required to create a new token instead.

Now that you are registered and have an API token, you will need to get the `twine` package. You will use `twine` to upload your package to PyPI. To install `twine`, you can use `pip` like this:

```
1 python3 -m pip install --user --upgrade twine
```

Once installed, you can upload your package to Test PyPI using the following command:

```
1 python3 -m twine upload --repository testpypi dist/*
```

Note that you will need to run this command from within the folder that contains the `setup.py` file as it is copying all the files in the `dist` folder to Test PyPI. When you run this command, it will prompt you for a username and password. For the username, you need to use `_token_`. The password is the token value that is prefixed with `pypi-`.

When this command runs, you should see output similar to the following:

```
1 Uploading distributions to https://test.pypi.org/legacy/
2 Enter your username: [your username]
3 Enter your password:
4 Uploading arithmetic_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl
5 100%|████████████████████████████████████████| 4.65k/4.65k [00:01<00:00, 2.88kB/s]
6 Uploading arithmetic_YOUR_USERNAME_HERE-0.0.1.tar.gz
7 100%|████████████████████████████████████████| 4.25k/4.25k [00:01<00:00, 3.05kB/s]
```

At this point, you should now be able to view your package on Test PyPI at the following URL:

- https://test.pypi.org/project/arithmetic_YOUR_USERNAME_HERE

Now you can test installing your package from Test PyPI by using the following command:

```
1 python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps arithmetic\
2 c-YOUR-USERNAME-HERE
```

If everything worked correctly, you should now have the `arithmetic` package installed on your system. Of course, this tutorial showed you how to package things up for Test PyPI. Once you have verified that it works, then you will need to do the following to install to the real PyPI:

- Choose a memorable and unique name for the package
- Register an account at <https://pypi.org>
- Use `twine upload dist/*` to upload your package and enter your credentials for the account you registered on the real PyPI. You won't need to use the `--repository` flag when uploading to the real PyPI as that server is the default
- Install your package from the real PyPI by using `pip install your_unique_package_name`

Now you know how to distribute a package of your own creation on the Python Package Index!

Wrapping Up

Python modules and packages are what you import in your programs. They are, in many ways, the building blocks of your programs. In this chapter, you learned about the following:

- Creating a Module
- Creating a Package
- Packaging a Project for PyPI
- Creating Project Files
- Creating `setup.py`
- Uploading to PyPI

At this point, you not only know what modules and packages are, but also how to distribute them via the Python Package Index. Now you and any other Python developer can download and install your packages. Congratulations! You are now a package maintainer!

Review Questions

1. What is a module?
2. How is a package different from a module?
3. What are at least two tools that are required for packaging a project?
4. What are some of the files you need to include in a package that are not Python files?

Chapter 44 - How to Create an Exe for Windows

You have just created an awesome new application. Maybe it's a game or maybe it's an image viewer. Whatever your application is, you want to share it with your friend or a family member. However, you know they won't know how to install Python or any of the dependencies. What do you do? You need something that will transform your code into an executable!

Python has many different tools you can use to convert your Python code into a Windows executable. Here are a few different tools you can use:

- PyInstaller
- py2exe
- cx_freeze
- Nuitka
- Briefcase

These various tools can all be used to create executables for Windows. They work slightly differently, but the end result is that you will have an executable and perhaps some other files that you need to distribute too.

PyInstaller and Briefcase can be used to create Windows and MacOS executables. You will learn more about creating a Mac executable in [chapter 46](#). Nuitka is a little different in that it turns your Python code into C code before converting it into an executable. What this means is that the result ends up much smaller than PyInstaller's executable.

However, for this chapter, you will focus on **PyInstaller**. It is one of the most popular packages for this purpose and has a lot of support. PyInstaller also has good documentation and there are many tutorials available for it.

In this chapter, you will learn about:

- Installing PyInstaller
- Creating an Executable for a Command-Line Application
- Creating an Executable for a GUI

Let's transform some code into a Windows executable!

Installing PyInstaller

To get started, you will need to install PyInstaller. Fortunately, PyInstaller is a Python package that can be easily installed using pip:

```
1 python -m pip install pyinstaller
```

This command will install PyInstaller and any dependencies that it needs on your machine. You should now be ready to create an executable with PyInstaller!

Creating an Executable for a Command-Line Application

The next step is to pick some code that you want to turn into an executable. You can grab the `PySearch` utility from chapter 32 and turn it into a binary. Here is the code again for your convenience:

```
1 # pysearch.py
2
3 import argparse
4 import pathlib
5
6 def search_folder(path, extension, file_size=None):
7     """
8         Search folder for files
9     """
10    folder = pathlib.Path(path)
11    files = list(folder.rglob(f'*.{extension}'))
12
13    if not files:
14        print(f'No files found with {extension}')
15        return
16
17    if file_size is not None:
18        files = [f for f in files
19                  if f.stat().st_size > file_size]
20
21    print(f'{len(files)} *.{extension} files found:')
22    for file_path in files:
23        print(file_path)
24
25
26 def main():
27     parser = argparse.ArgumentParser(
28         'PySearch',
29         description='PySearch - The Python Powered File Searcher')
```

```

30     parser.add_argument('-p', '--path',
31                         help='The path to search for files',
32                         required=True,
33                         dest='path')
34     parser.add_argument('-e', '--ext',
35                         help='The extension to search for',
36                         required=True,
37                         dest='extension')
38     parser.add_argument('-s', '--size',
39                         help='The file size to filter on in bytes',
40                         type=int,
41                         dest='size',
42                         default=None)
43
44     args = parser.parse_args()
45     search_folder(args.path, args.extension, args.size)
46
47 if __name__ == '__main__':
48     main()

```

You can go back to chapter 32 if you'd like all the details on how this code works. This chapter isn't going to focus on that. Instead, your objective is to turn this code into an executable.

Open up a Command Prompt (cmd.exe) in Windows and navigate to the folder that has your pysearch.py file in it. To turn the Python code into a binary executable, you need to run the following command:

```
1 pyinstaller pysearch.py
```

If Python isn't on your Windows path, you may need to type out the full path to pyinstaller to get it to run. It will be located in a **Scripts** folder wherever your Python is installed on your system.

When you run that command, you will see some output that will look similar to the following:

```

1 6531 INFO: PyInstaller: 3.6
2 6576 INFO: Python: 3.8.2
3 6707 INFO: Platform: Windows-10-10.0.10586-SP0
4 6828 INFO: wrote C:\Users\mike\AppData\Local\Programs\Python\Python38-32\pysearch.sp\
5 ec
6 6880 INFO: UPX is not available.
7 7110 INFO: Extending PYTHONPATH with paths
8 ['C:\\\\Users\\\\mike\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python38-32',
9  'C:\\\\Users\\\\mike\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python38-32']
10 7120 INFO: checking Analysis

```

```
11 7124 INFO: Building Analysis because Analysis-00.toc is non existent
12 7128 INFO: Initializing module dependency graph...
13 7153 INFO: Caching module graph hooks...
14 7172 INFO: Analyzing base_library.zip ...
```

PyInstaller is very verbose and will print out a LOT of output. When it is finished, you will have a `dist` folder with a `pysearch` folder inside of it. Within the `pysearch` folder are many other files, including one called `pysearch.exe`. You can try navigating to the `pysearch` folder in your Command Prompt and then run `pysearch.exe`:

```
1 C:\Users\mike\AppData\Local\Programs\Python\Python38-32\dist\pysearch>pysearch.exe
2 usage: PySearch [-h] -p PATH -e EXTENSION [-s SIZE]
3 PySearch: error: the following arguments are required: -p/--path, -e/--ext
```

That looks like a pretty successful build! However, if you want to give the executable to your friends, you will have to give them the entire `pysearch` folder as all those other files in there are also required.

You can fix that issue by passing the `--onefile` flag, like this:

```
1 pyinstaller pysearch.py --onefile
```

The output from that command is similar to the first command. This time when you go into the `dist` folder though, you will find a single file in there called `pysearch.exe` instead of a folder full of files.

Creating an Executable for a GUI

Creating an executable for a GUI is slightly different than it is for a command-line application. The reason is that the GUI is the main interface and PyInstaller's default is that the user will be using a Command Prompt or console window. If you run either of the PyInstaller commands that you learned about in the previous section, it will successfully create your executable. However, when you go to use your executable, you will see a Command Prompt appear in addition to your GUI.

You usually don't want that. To suppress the Command Prompt, you need to use the `--noconsole` flag.

To test out how this would work, grab the code for the image viewer you created with wxPython in [chapter 42](#). Here is the code again for your convenience:

```
1 # image_viewer.py
2
3 import wx
4
5 class ImagePanel(wx.Panel):
6
7     def __init__(self, parent, image_size):
8         super().__init__(parent)
9         self.max_size = 240
10
11         img = wx.Image(*image_size)
12         self.image_ctrl = wx.StaticBitmap(self,
13                                         bitmap=wx.Bitmap(img))
14
15         browse_btn = wx.Button(self, label='Browse')
16         browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)
17
18         self.photo_txt = wx.TextCtrl(self, size=(200, -1))
19
20         main_sizer = wx.BoxSizer(wx.VERTICAL)
21         hsizer = wx.BoxSizer(wx.HORIZONTAL)
22
23         main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
24         hsizer.Add(browse_btn, 0, wx.ALL, 5)
25         hsizer.Add(self.photo_txt, 0, wx.ALL, 5)
26         main_sizer.Add(hsizer, 0, wx.ALL, 5)
27
28         self.SetSizer(main_sizer)
29         main_sizer.Fit(parent)
30         self.Layout()
31
32     def on_browse(self, event):
33         """
34             Browse for an image file
35             @param event: The event object
36         """
37
38         wildcard = "JPEG files (*.jpg)|*.jpg"
39         with wx.FileDialog(None, "Choose a file",
40                            wildcard=wildcard,
41                            style=wx.ID_OPEN) as dialog:
42             if dialog.ShowModal() == wx.ID_OK:
43                 self.photo_txt.SetValue(dialog.GetPath())
44                 self.load_image()
```

```
44
45     def load_image(self):
46         """
47             Load the image and display it to the user
48         """
49         filepath = self.photo_txt.GetValue()
50         img = wx.Image(filepath, wx.BITMAP_TYPE_ANY)
51
52         # scale the image, preserving the aspect ratio
53         W = img.GetWidth()
54         H = img.GetHeight()
55         if W > H:
56             NewW = self.max_size
57             NewH = self.max_size * H / W
58         else:
59             NewH = self.max_size
60             NewW = self.max_size * W / H
61         img = img.Scale(NewW,NewH)
62
63         self.image_ctrl.SetBitmap(wx.Bitmap(img))
64         self.Refresh()
65
66
67 class MainFrame(wx.Frame):
68
69     def __init__(self):
70         super().__init__(None, title='Image Viewer')
71         panel = ImagePanel(self, image_size=(240,240))
72         self.Show()
73
74
75 if __name__ == '__main__':
76     app = wx.App(redirect=False)
77     frame = MainFrame()
78     app.MainLoop()
```

To turn this into an executable, you would run the following PyInstaller command:

```
1 pyinstaller.exe image_viewer.py --noconsole
```

Note that you are **not** using the `--onefile` flag here. Windows Defender will flag GUIs that are created with the `--onefile` as malware and remove it. You can get around that by not using the

--onefile flag or by digitally signing the executable. Starting in Windows 10, all GUI applications need to be signed or they are considered malware.

Microsoft has a **Sign Tool** you can use, but you will need to purchase a digital certificate or create a self-signed certificate with **Makecert**, a .NET tool or something similar.

Wrapping Up

There are lots of different ways to create an executable with Python. In this chapter, you used PyInstaller. You learned about the following topics:

- Installing PyInstaller
- Creating an Executable for a Command-Line Application
- Creating an Executable for a GUI

PyInstaller has many other flags that you can use to modify its behavior when generating executables. If you run into issues with PyInstaller, there is a mailing list that you can turn to. Or you can search with Google and on StackOverflow. Most of the common issues that crop up are covered either in the PyInstaller documentation or are easily discoverable through searching online.

Review Questions

1. Name 3 different tools you can use to create a Windows executable out of Python code.
2. What command do you use with PyInstaller to create an executable?
3. How do you create a single file executable with PyInstaller?
4. Which flag do you use with PyInstaller to suppress the console window?

Chapter 45 - How to Create an Installer for Windows

Windows users have certain expectations when it comes to installing software. They typically think that the software should be able to be downloaded and then installed using a wizard-type interface. These wizards are created using “installer software”. While there aren’t really any Python packages available for creating installers, there are some nice free applications that you can use.

Two of the most popular are:

- NSIS - https://nsis.sourceforge.io/Main_Page
- Inno Setup - <http://www.jrsoftware.org/isinfo.php>

For this chapter, you will be using Inno Setup. You will also be learning about the following topics:

- Installing Inno Setup
- Creating an Installer
- Testing Your Installer

Let’s get started by learning to install the installer!

Installing Inno Setup

Installing Inno Setup is pretty simple. You will need to go to the following URL:

- <http://www.jrsoftware.org/isinfo.php>

Then go to the **Download Inno Setup** link and choose where you want to download the application from. After downloading Inno Setup, you will need to double-click the installer and install it on your system.

Now that you have Inno Setup installed on your Windows machine, you can create your own installer!

Creating an Installer

The first step in creating an installer is to choose what you want to install. For this example, you will use the image viewer executable that you created in the previous chapter.

Inno Setup makes creating an installer really straight-forward. It uses a wizard that will walk you through it step-by-step. To get started, run Inno Setup and you should see a screen that looks like this:

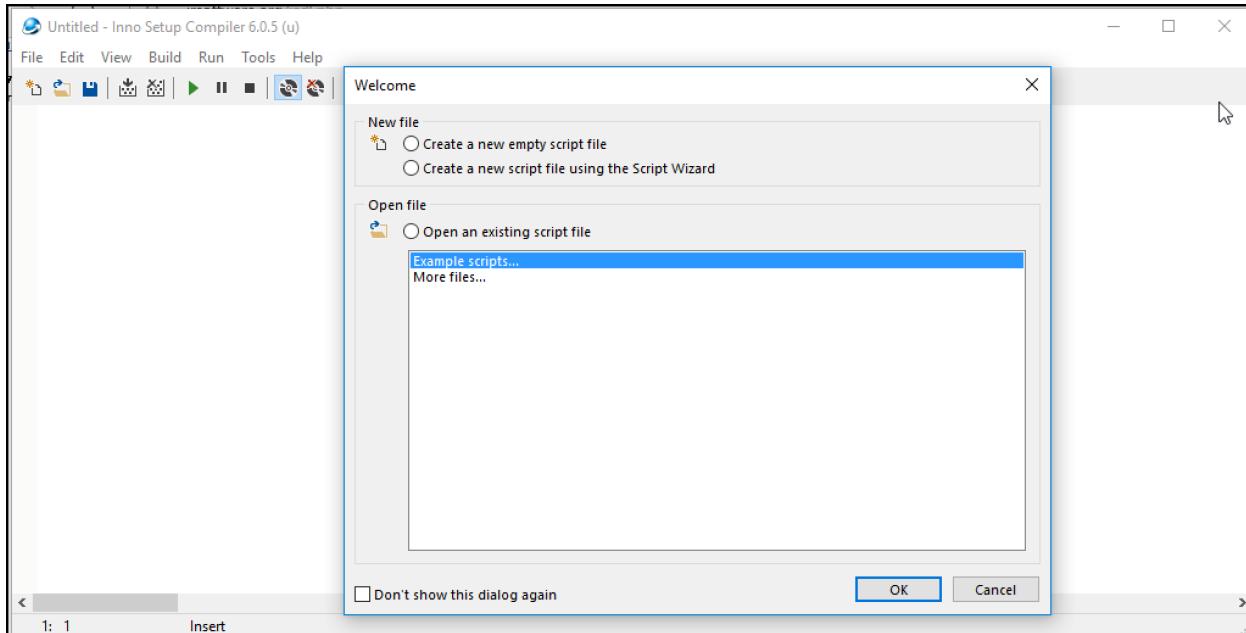


Fig. 45-1: Inno Setup's Startup Page

While Inno Setup defaults to opening an existing file, what you want to do is choose the second option from the top: “Create a new script file using the Script Wizard”. Then press OK.

You should now see the first page of the Inno Setup Script Wizard:

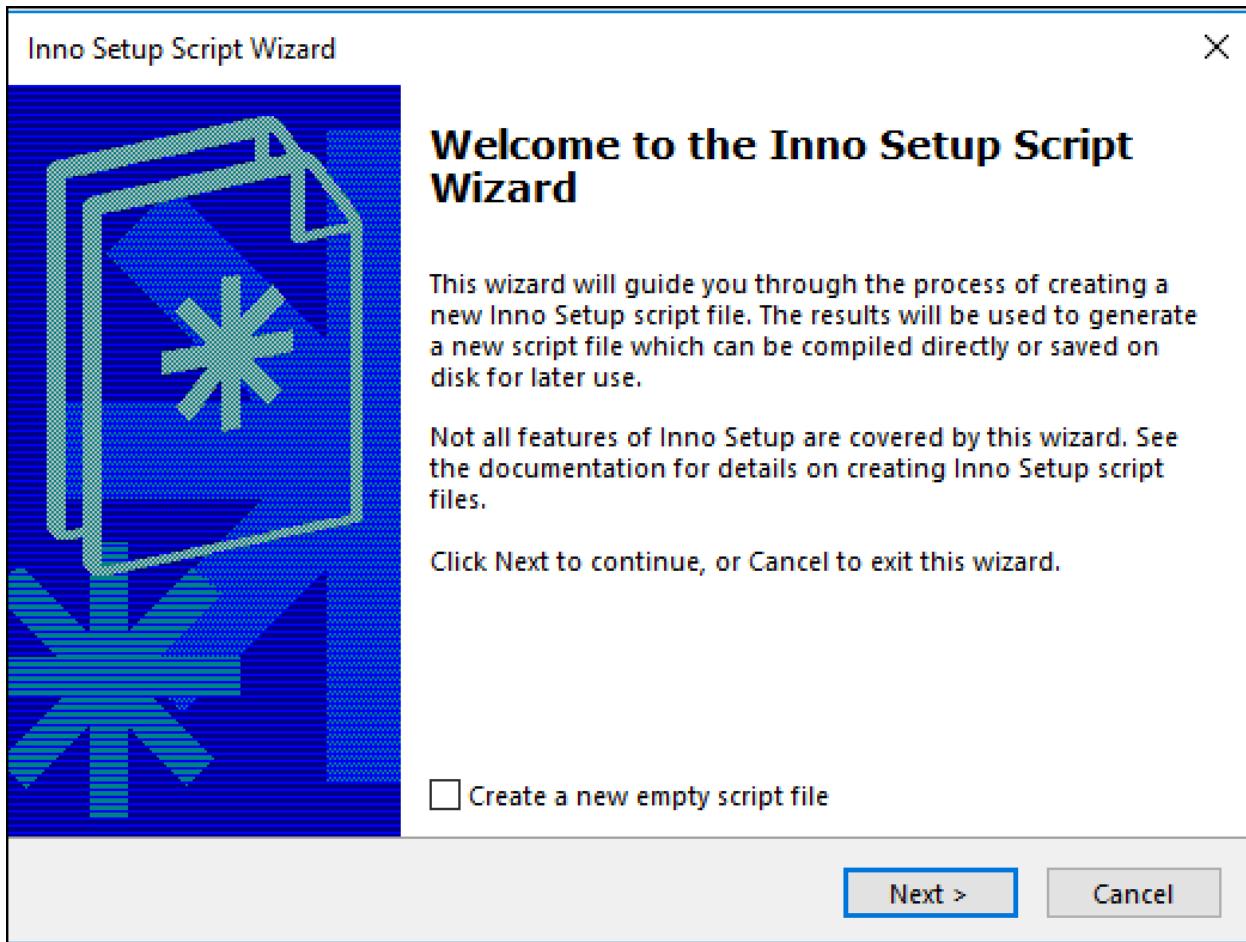


Fig. 45-2: Inno Setup's Welcome Page

Go ahead and hit **Next** here. You will now be on the **Application Information** page:

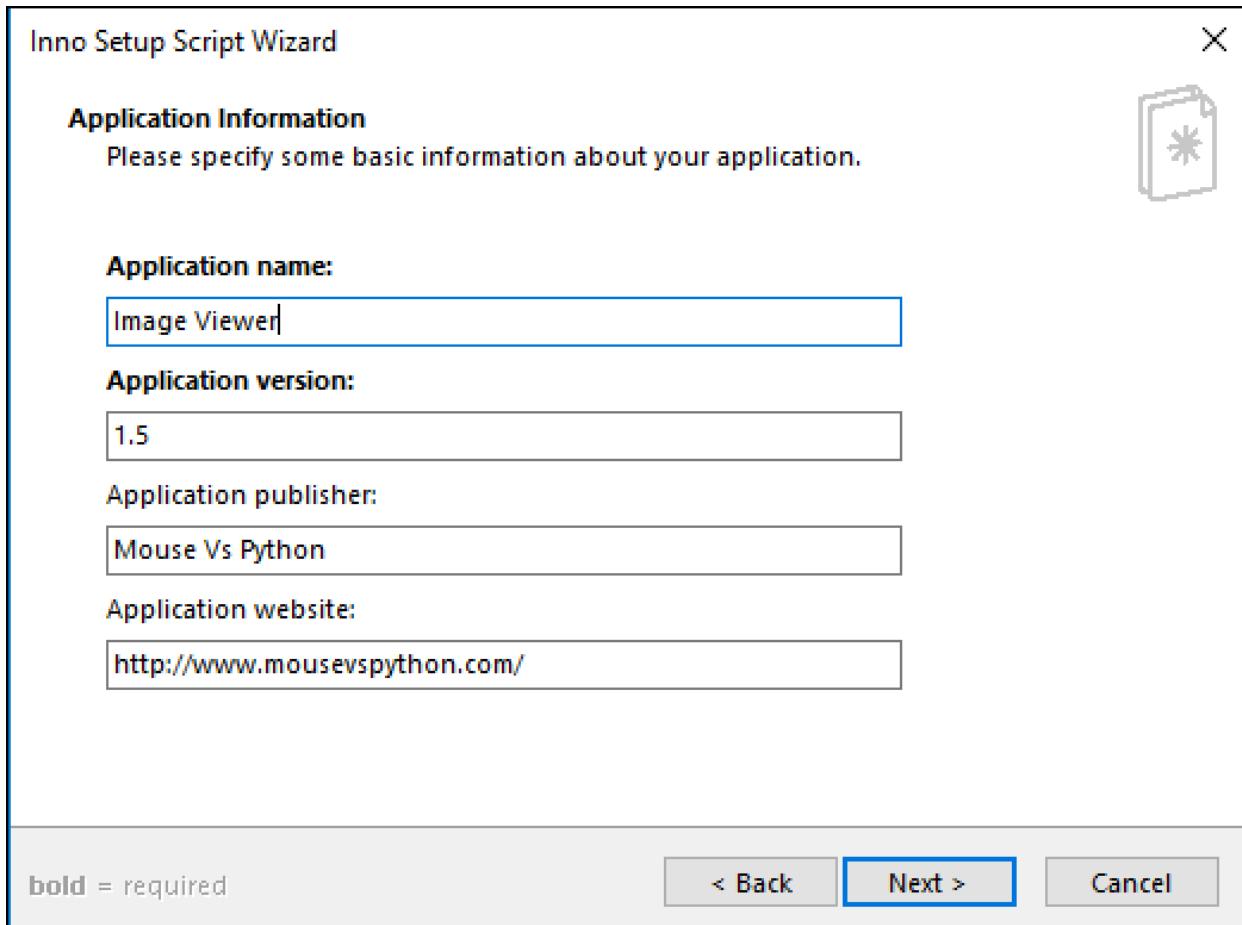


Fig. 45-3: Inno Setup's Application Information Page

This is where you enter your application's name, its version information, the publisher's name and the application's website. You can fill it out to match what's in the screenshot or customize it to your own specifications.

Once you are finished, press **Next** and you should see the following:

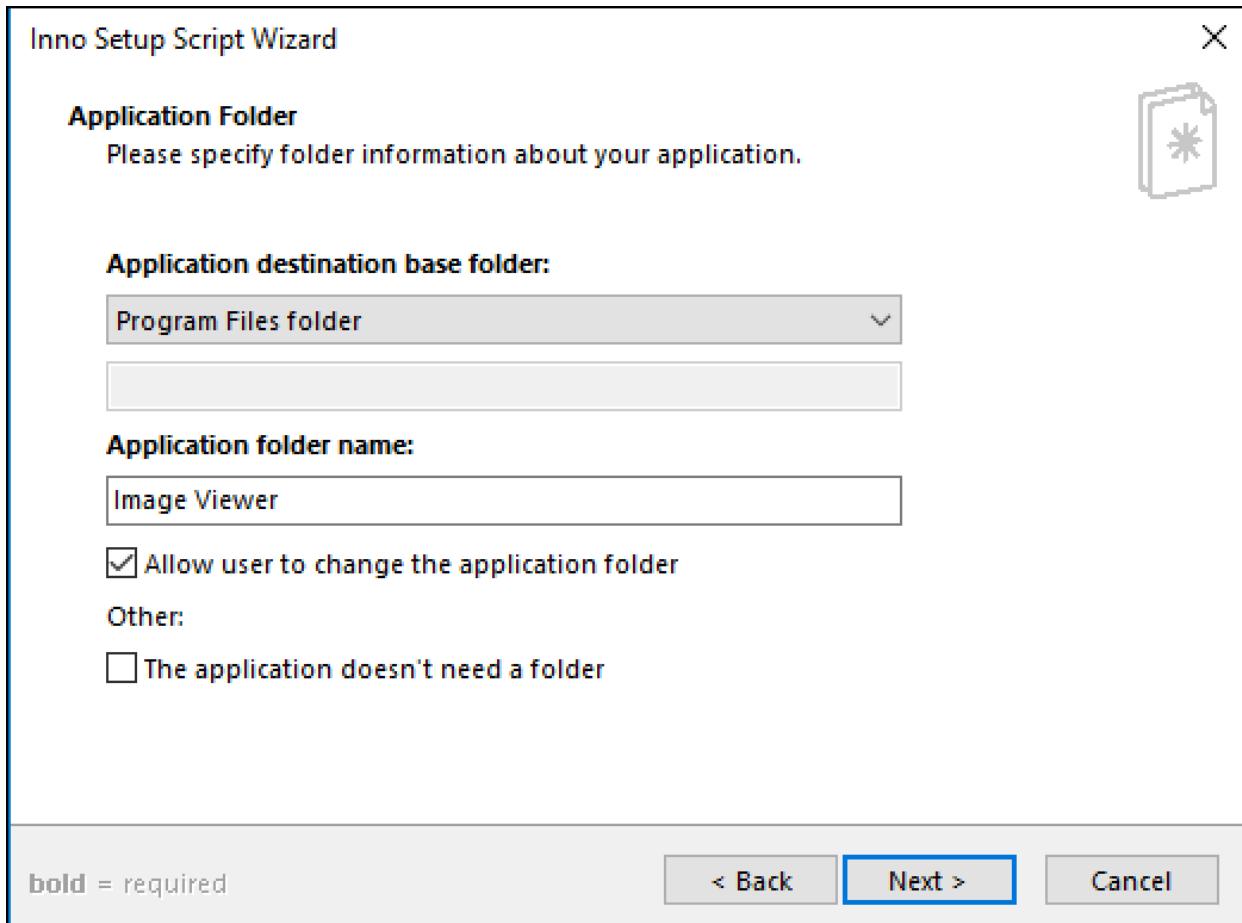


Fig. 45-4: Inno Setup's Application Folder Page

This page of the wizard is where you can set the application's install directory. On Windows, most applications install to **Program Files**, which is also the default here. This is also where you set the folder name for your application. This is the name of the folder that will appear in Program Files. Alternatively, you can check the box at the bottom that indicates that your application doesn't need a folder at all.

You can edit this page if you don't like the defaults. Otherwise, press **Next** to continue:

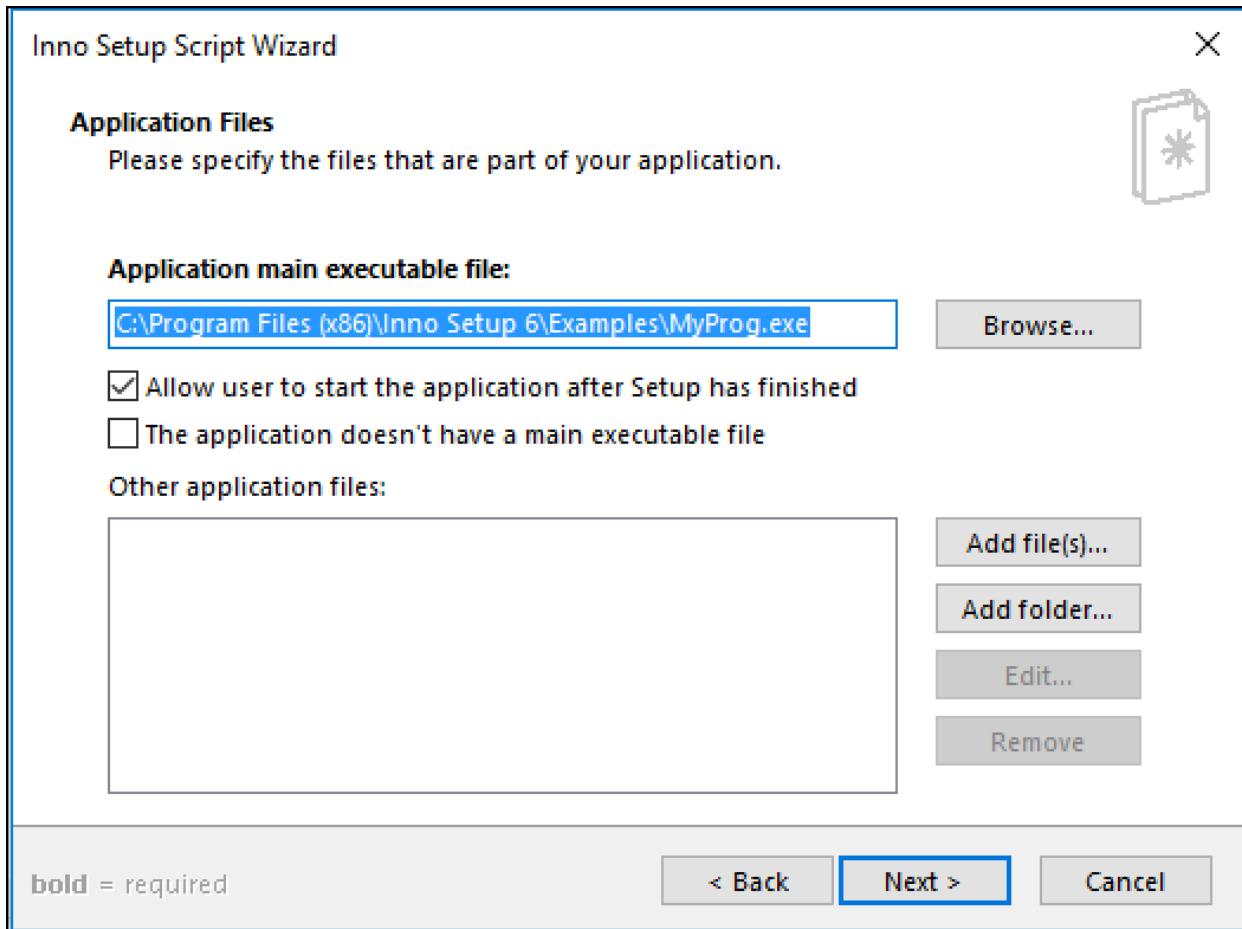


Fig. 45-5: Inno Setup's Application Files Page

Here is where you will choose the main executable file. In this case, you want to choose the executable you created with PyInstaller. If you didn't create the executable using the `--onefile` flag, then you can add the other files using the **Add file(s)...** button. If your application requires any other special files, like a SQLite database file or images, this is also where you would add them.

Note: Remember on Windows 10, if you used `--onefile`, then you also need to sign the file or it will be marked as malware by Windows Defender.

By default, this page will allow the user to run your application when the installer finishes. A lot of installers do this, so it's actually expected by most users.

When you are done, click **Next** to go to the Shortcuts page:

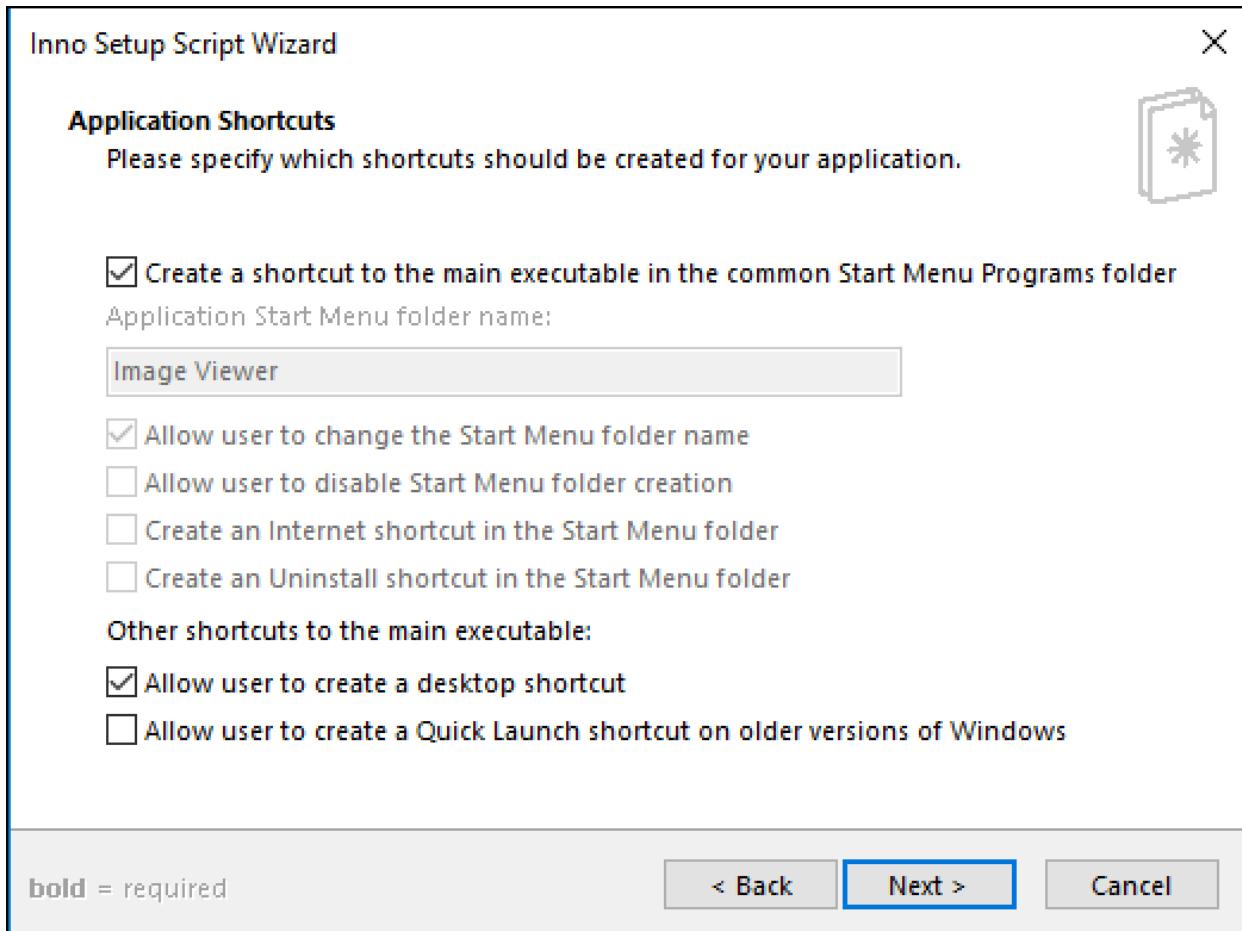


Fig. 45-6: Inno Setup's Application Shortcuts Page

This is the **Application Shortcuts** page and it allows you to manage what shortcuts are created for your application and where they should go. The options are pretty self-explanatory. You can usually use the defaults, but you are welcome to change them however you see fit.

Once you are finished configuring the shortcuts, press **Next** to continue:

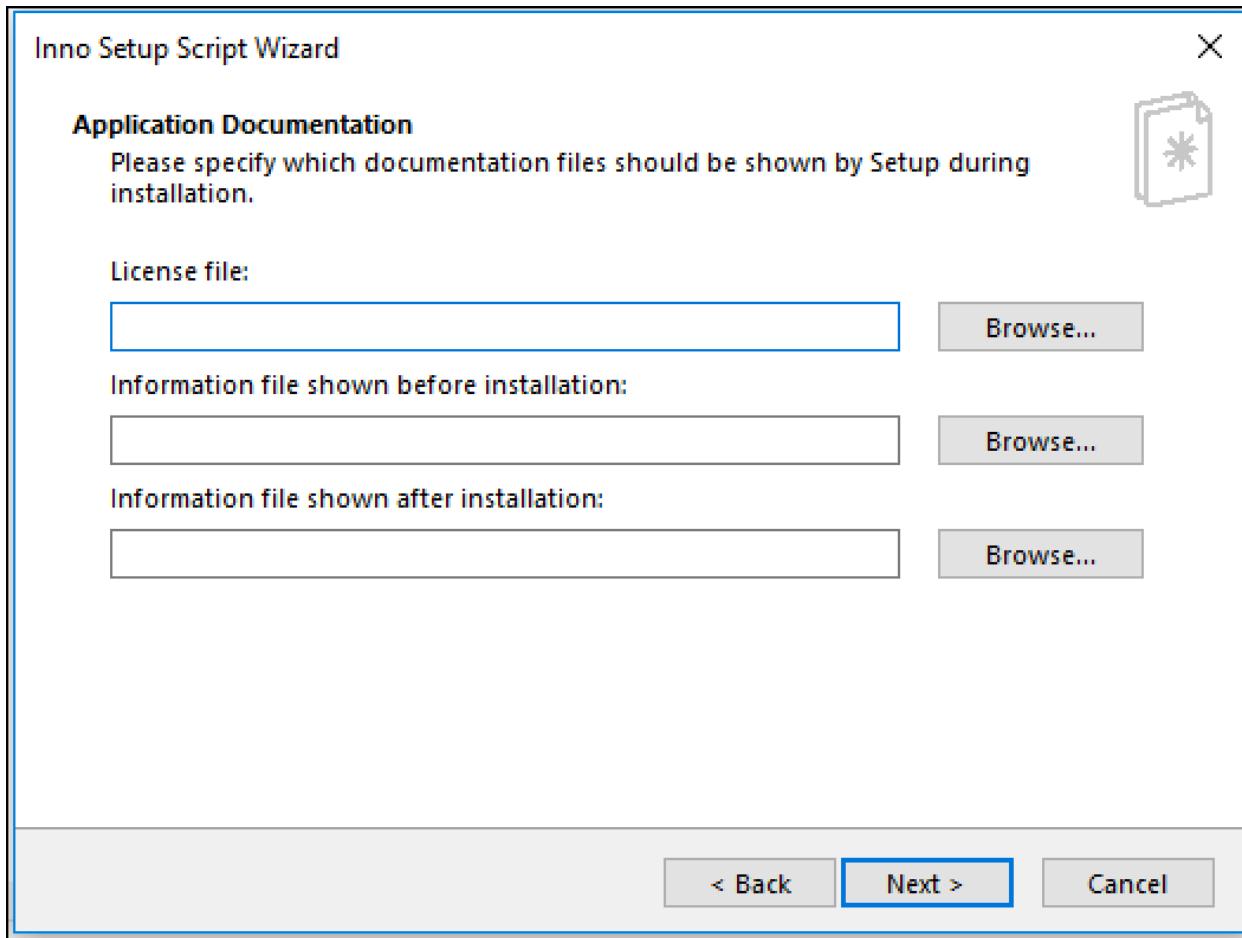


Fig. 45-7: Inno Setup's Application Documentation Page

The **Documentation Page** of the wizard is where you can add your application's license file. For example, if you were putting out an open source application, you can add the GPL or MIT or whatever license file you need there. If this were a commercial application, this is where you would add your **End-User License Agreement (EULA)** file. You can also add some extra information before and after the installation here too!

All these files are optional. If you don't want to apply a license or add before and after instructions, then you can just click **Next**.

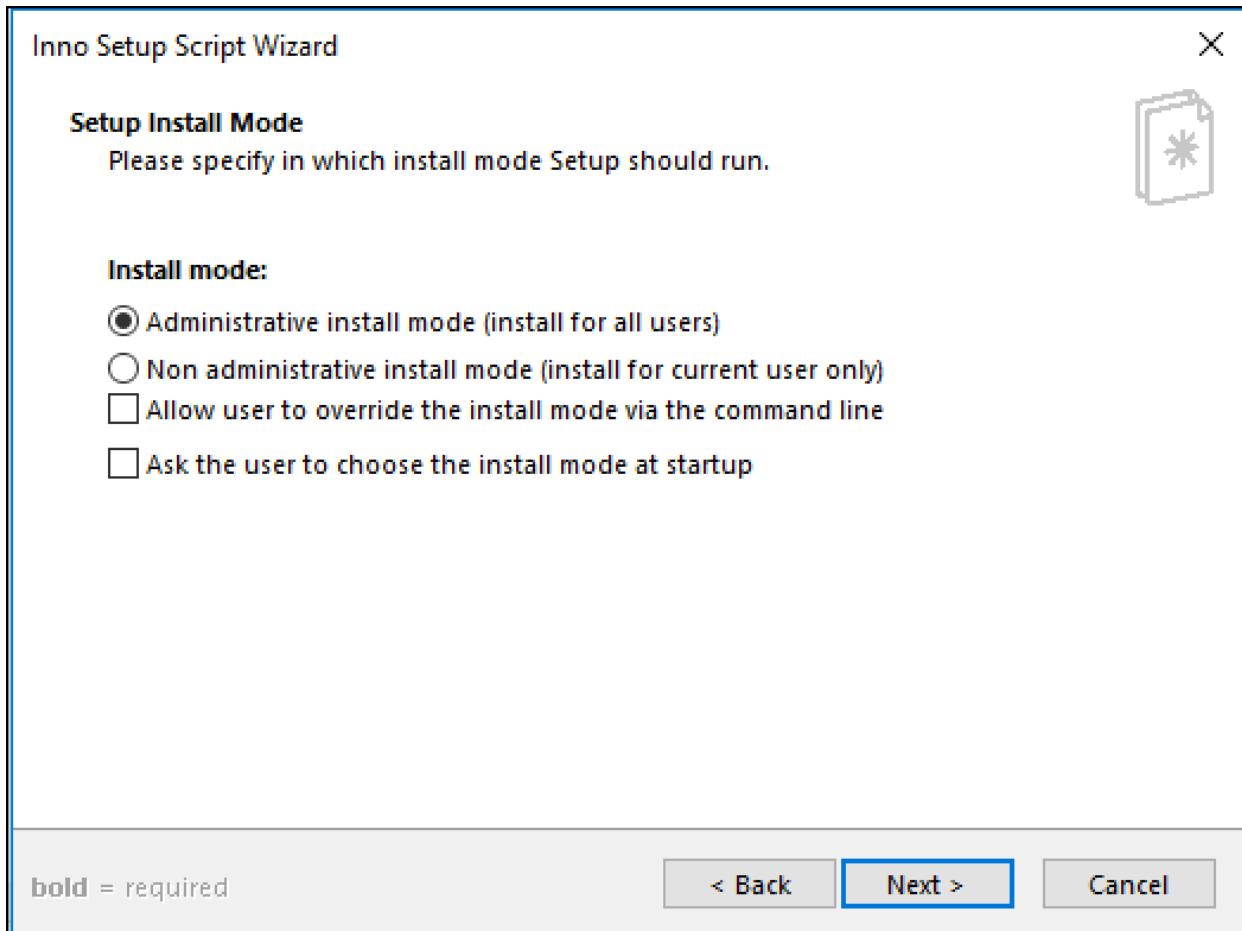


Fig. 45-8: Inno Setup's Setup Install Mode Page

This page specifies which install mode to use when installing your application. The default is to install for all users, which is probably what you want. The other options are pretty self-explanatory. You can play around with them or read more about them in Inno Setup's documentation. Once you're done with this page, go on to the next one.

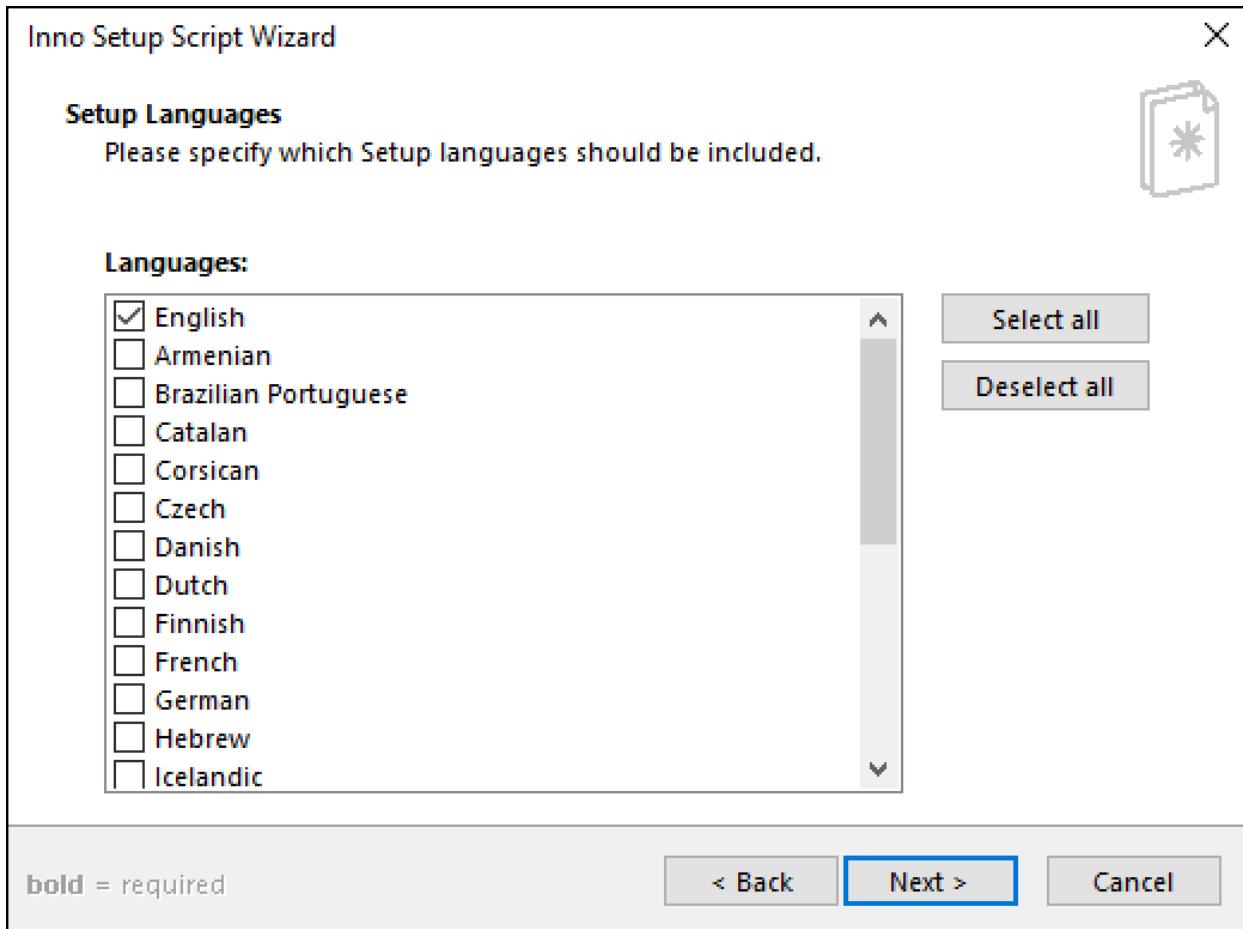


Fig. 45-9: Inno Setup's Setup Languages Page

Here you can set up which languages should be included in your installer. The default is English, but Inno Setup supports many others. Feel free to modify this as needed before continuing on.

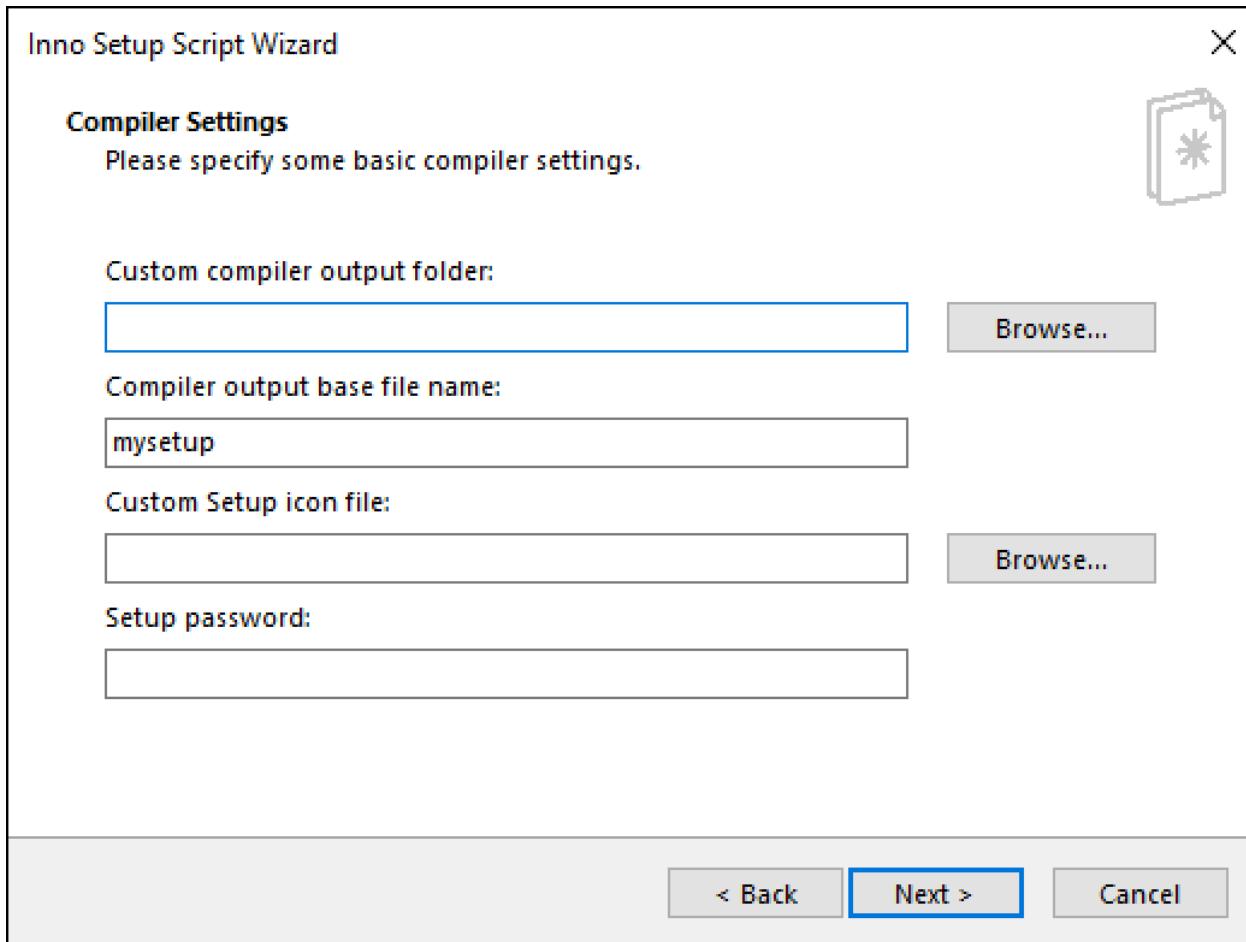


Fig. 45-10: Inno Setup's Compiler Settings Page

The **Compiler Settings** page lets you name the output setup file, which defaults to simply **setup**. You can set the output folder here, add a custom setup file icon and even add password protection to the setup file. I usually just leave the defaults alone, but this is an opportunity to add some branding to the setup if you have a nice icon file handy.

The next page is for the preprocessor:

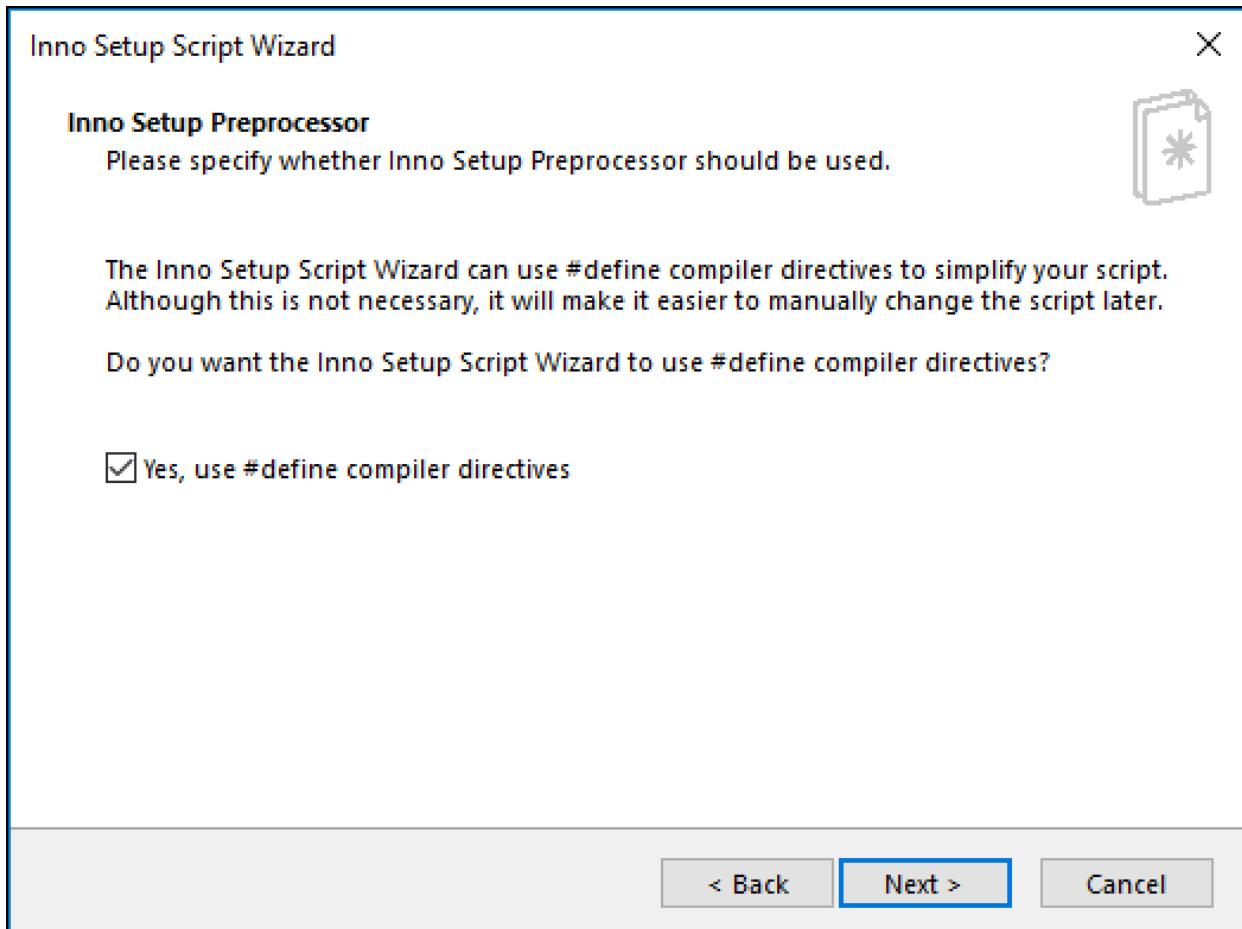


Fig. 45-11: Inno Setup's Preprocessor Page

The preprocessor is primarily for catching typos in the Inno Setup script file. It basically adds some helpful options at compile time to your Inno Setup script.

Check out the following URL for full details:

- <http://www.jrsoftware.org/ispphelp/>

When you press **Next** you will reach the final page of the wizard:

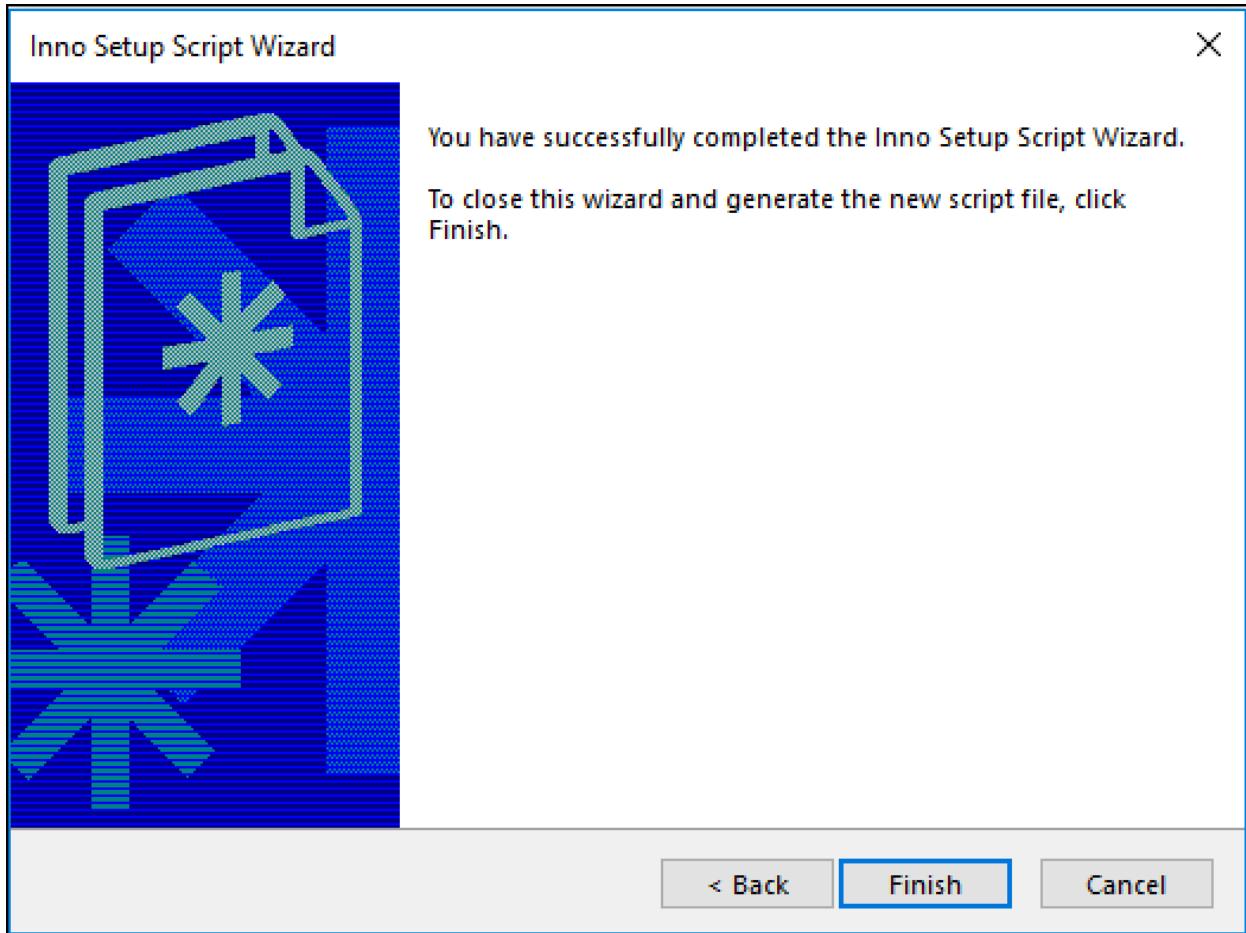


Fig. 45-12: Inno Setup's Last Wizard Page

There's nothing to do here but press **Finish**, **Back** or **Cancel**. Go ahead and complete the wizard by pressing **Finish**:

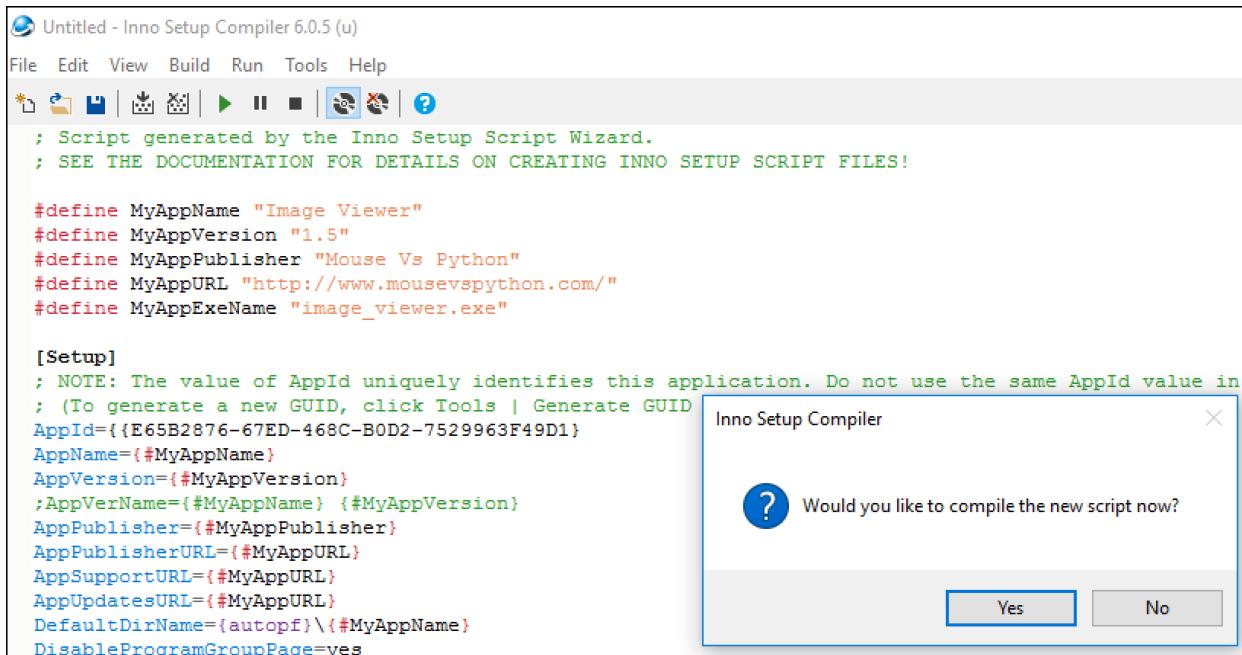


Fig. 45-13: Inno Setup's Run Compiler Script

Inno Setup now asks you if you would like to compile the script. This means that Inno Setup has created a build script for your installer and would like your permission to run that script to actually create your installer executable. Press Yes to continue.

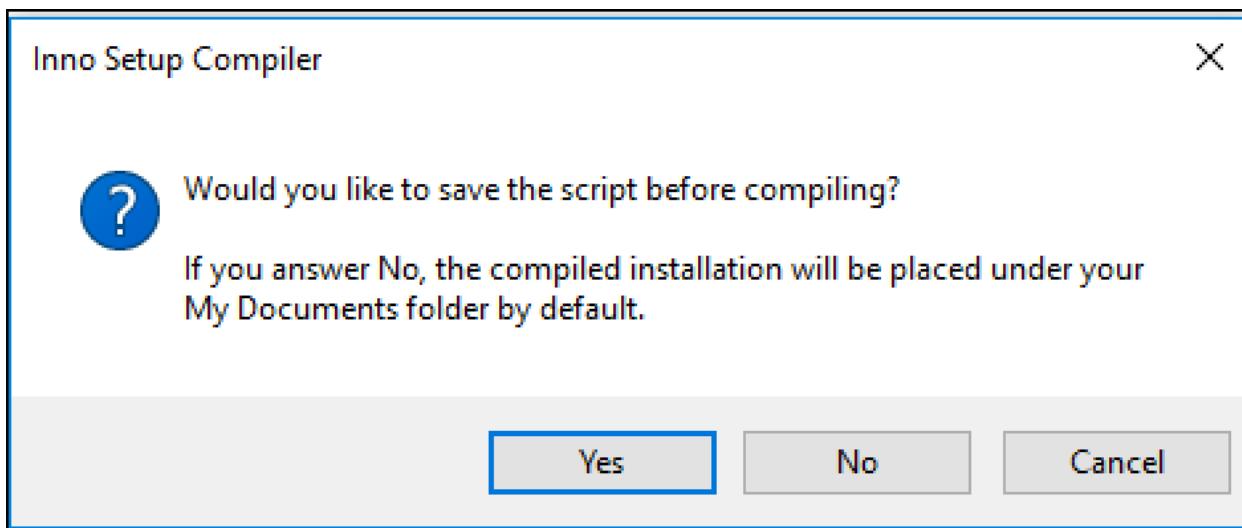


Fig. 45-14: Inno Setup's Save Compiler Script

This dialog asks if you would like to save the Inno Setup compiler script before running it. That is certainly a good idea as you don't want to lose any of your changes. Press Yes to save it and then choose what to name the script and where to save it.

Once the script is saved, it will create your installer. At this point you should have an installer

created! You can edit the script file in any text editor, although the Inno Setup editor is probably best as it has its own syntax highlighting built-in to it.

Testing Your Installer

Now that you have an installer, it's a good idea to test it. When you finish building the installer with Inno Setup, it will automatically run the installer for you. You can choose to install your application now and verify that it launches correctly.

You may see an error when you attempt to run your application. That usually means that you forgot to include some file or other in your installer. Occasionally you will get errors that are related to permission issues. You can solve the first issue by comparing what files were installed to the ones in your `dist` folder that you created with PyInstaller. If there are any missing, you can run Inno Setup and go through the wizard again or you can modify the script file yourself directly.

Permission issues are more difficult and will take some trial and error to figure out. You will likely need to research what exactly the problem is to figure out how to solve it.

Of course, occasionally there's something wrong with the executable itself. Then you'll have to debug it before you can repackage it into an installer.

Wrapping Up

Creating an installer for your Windows users is a good idea. Having an installer gives the users confidence in your program. Installers add that extra bit of polish and they make it easier to make sure all the files you need are installed with your application.

In this chapter, you learned about:

- Installing Inno Setup
- Creating an Installer
- Testing Your Installer

You should spend some time reading the documentation for Inno Setup and then trying it out on some of your own creations. You will quickly find that Inno Setup is intuitive and you'll soon have some installers of your own.

Review Questions

1. Name two programs you can use to create installers for Windows.
2. How do you modify an Inno Setup compiler script?
3. Why should you test your installers?

Chapter 46 - How to Create an “exe” for Mac

Each operating system has its own method of installing or running an application. On Windows, you run an executable most of the time. Executables have an extension of .exe. Apple’s Mac software has the concept of applications that use the extension .app. These are technically a bundle of files including a runnable binary. They are kind of like a runnable zip-file. All you need to do is double-click the .app file to run it.

If you want to distribute a Python application on Mac OSX, you have three options:

- PyInstaller - <https://www.pyinstaller.org/>
- Briefcase - <https://beeware.org/project/projects/tools/briefcase/>
- py2app - <https://py2app.readthedocs.io/en/latest/>

You learned about the PyInstaller package in [chapter 44](#). In this chapter you will learn how to use PyInstaller to create an .app bundle.

In this chapter, you will cover the following two topics:

- Installing PyInstaller
- Creating an Executable with PyInstaller

Being able to create .app bundles gives to a way to distribute your application to Mac users. Let’s find out how to install the Python packages that you need so you can get started!

Installing PyInstaller

Both PyInstaller and Briefcase can be installed using pip. Here is how you would install PyInstaller:

```
1 python -m pip install pyinstaller
```

This is a pretty quick installation because PyInstaller doesn’t have a lot of dependencies. Once PyInstaller is installed, you can move on to the next section and create an executable for Mac OSX.

Creating an Executable with PyInstaller

Creating an executable requires you to have some code. You can use the same wxPython GUI examples that you used in [chapter 44](#). Here is the `image_viewer` program code:

```
1 # image_viewer.py
2
3 import wx
4
5 class ImagePanel(wx.Panel):
6
7     def __init__(self, parent, image_size):
8         super().__init__(parent)
9         self.max_size = 240
10
11     img = wx.Image(*image_size)
12     self.image_ctrl = wx.StaticBitmap(self,
13                                     bitmap=wx.Bitmap(img))
14
15     browse_btn = wx.Button(self, label='Browse')
16     browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)
17
18     self.photo_txt = wx.TextCtrl(self, size=(200, -1))
19
20     main_sizer = wx.BoxSizer(wx.VERTICAL)
21     hsizer = wx.BoxSizer(wx.HORIZONTAL)
22
23     main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
24     hsizer.Add(browse_btn, 0, wx.ALL, 5)
25     hsizer.Add(self.photo_txt, 0, wx.ALL, 5)
26     main_sizer.Add(hsizer, 0, wx.ALL, 5)
27
28     self.SetSizer(main_sizer)
29     main_sizer.Fit(parent)
30     self.Layout()
31
32     def on_browse(self, event):
33         """
34             Browse for an image file
35             @param event: The event object
36         """
37
38         wildcard = "JPEG files (*.jpg)|*.jpg"
39         with wx.FileDialog(None, "Choose a file",
40                           wildcard=wildcard,
41                           style=wx.ID_OPEN) as dialog:
42             if dialog.ShowModal() == wx.ID_OK:
43                 self.photo_txt.SetValue(dialog.GetPath())
44                 self.load_image()
```

```
44
45     def load_image(self):
46         """
47             Load the image and display it to the user
48         """
49         filepath = self.photo_txt.GetValue()
50         img = wx.Image(filepath, wx.BITMAP_TYPE_ANY)
51
52         # scale the image, preserving the aspect ratio
53         W = img.GetWidth()
54         H = img.GetHeight()
55         if W > H:
56             NewW = self.max_size
57             NewH = self.max_size * H / W
58         else:
59             NewH = self.max_size
60             NewW = self.max_size * W / H
61         img = img.Scale(NewW,NewH)
62
63         self.image_ctrl.SetBitmap(wx.Bitmap(img))
64         self.Refresh()
65
66
67 class MainFrame(wx.Frame):
68
69     def __init__(self):
70         super().__init__(None, title='Image Viewer')
71         panel = ImagePanel(self, image_size=(240,240))
72         self.Show()
73
74 if __name__ == '__main__':
75     app = wx.App(redirect=False)
76     frame = MainFrame()
77     app.MainLoop()
```

You can use the same commands for PyInstaller that you use on Windows to create a Mac application bundle. Open up a console window on your Mac and navigate to the folder that contains `image_viewer.py`. Then run the following command:

```
1 pyinstaller image_viewer.py --windowed
```

Running this command will give you this output, which is quite similar to the output you saw on Windows:

```
1 Mikes-MacBook-Pro:chapter46_mac michael$ pyinstaller image_viewer.py
2 102 INFO: PyInstaller: 3.6
3 102 INFO: Python: 3.8.1
4 112 INFO: Platform: macOS-10.14.6-x86_64-i386-64bit
5 112 INFO: wrote /Users/michael/Dropbox/Books/Python101_2nd_ed/python101code/chapter4\
6 _mac/image_viewer.spec
7 121 INFO: UPX is not available.
8 123 INFO: Extending PYTHONPATH with paths
9 ['/Users/michael/Dropbox/Books/Python101_2nd_ed/python101code/chapter46_mac',
10 '/Users/michael/Dropbox/Books/Python101_2nd_ed/python101code/chapter46_mac']
11 123 INFO: checking Analysis
12 123 INFO: Building Analysis because Analysis-00.toc is non existent
13 123 INFO: Initializing module dependency graph...
14 126 INFO: Caching module graph hooks...
15 133 INFO: Analyzing base_library.zip ...
```

Of course, when you run PyInstaller on a Mac, you will see that the platform information reflects that. When this finishes, you will have a `dist` folder that contains your app bundle. There does seem to be a bug in Python 3.7 and 3.8 that occurs when you try to launch the app bundle:

```
1 FileNotFoundError: Tcl data directory
```

To fix this issue, you will need to run the following commands in your console:

```
1 cd dist/image_viewer.app/Contents/MacOS/
2 mkdir tcl tk
```

Now when you go to run the application bundle, it will launch your wxPython GUI correctly. Because of this issue, if you use the `--onefile` option, you won’t be able to add those two folders. So the `--onefile` option won’t actually launch. Instead, you will see it attempt to launch and immediately crash. Hopefully PyInstaller or a newer version of Python will resolve this issue.

Wrapping Up

Creating executables on MacOS is a bit more complicated than creating them on Windows. You also have fewer options for creating application bundles. Windows has many Python packages to choose from when it comes to creating binaries. Mac has PyInstaller and Beeware’s Briefcase. The latter is designed to work with other Beeware products, but can be used with regular Python too.

Go ahead and give them both a try to see what works best for you. PyInstaller is a much more mature option, but sometimes it’s nice to give the newer packaging applications a try.

Review Questions

1. What tools can you use to create executables for Python on MacOS?
2. What is an executable called on MacOS?

Afterword

This book was really fun to write. Python 101 was the very first book I had ever written and trying to decide what to rewrite took a lot of effort. In the end, I decided to rewrite the entire book. Some parts from the original were cut completely out. Other parts were reordered or absorbed into other chapters.

I had a lot of new help with this version of the book. There were many people who read it and gave me feedback about portions of the book. My intention in writing Python 101 has always been to write the kind of beginner Python book that I always wanted when I was learning Python. I always wanted more than just the syntax of the language. My hope is that this book helps you to learn Python and glimpse the many possibilities that it provides for you.

Thanks so much for checking this book out. I hope you will let me know what you thought.

- Mike

Appendix A - Version Control

Version control, also known as **source control** or **revision control**, is a method for managing changes in documents, programs, websites, and the like. Changes to a file or set of files are identified using a revision number or **changeset**. Each revision is timestamped and associated with the person who made the change. A revision can be compared and restored and they can also be merged.

Software developers use version control to version their software. This makes it easier to change your code as well as roll your code back (AKA reverting your change) if need be. Revision control is used by editors of wiki software to track edits, correct mistakes, and protect against vandalism.

In this chapter, you will learn about the following:

- Version Control Systems
- Distributed vs Centralized Versioning
- Common Terminology
- Python IDE Version Control Support

Let's get started!

Version Control Systems

There are many types of version control systems (VCS). Word processors and spreadsheets even include rudimentary version control. Software developers have several different systems to choose from. Here are but a few of the popular version control systems:

- Git
- Mercurial
- Perforce
- Team Foundation Version Control (Microsoft)
- Apache Subversion

The most popular version control system in open source is Git. However, the other version control systems mentioned here are used by many organizations and for many different reasons. For example, if your company uses Microsoft tooling a lot, then you will be using Team Foundation. On the other hand, if your company is into open source or doesn't want to pay licensing fees, then you may choose to use an open source version control system like Git or Mercurial.

Distributed vs Centralized Versioning

There are two types of version control systems: **centralized** or **distributed**. A centralized system follows the client-server model. Two examples of this are Perforce and Subversion. These tools have a centralized server with a canonical copy of the source code. The clients are the developer's machines. They can check code out from the server to their local machine for editing. When finished, they will save or push their changes back to the server. If the server goes down, you cannot work on the code.

A distributed system, like Git or Mercurial, creates an entire copy of the source code on the developer's machine. This means that every developer has, in effect, a backup copy of the code base and its change history. This protects developers from data loss. Another advantage of distributed systems is that when you save code, it saves locally, which can be much faster.

When you want to share your code, you can push it peer-to-peer. The general method of doing this is to create some kind of cloud location to push your changes to, kind of like the centralized model. One of the most popular cloud providers is **Github**. You can push your changes there and then other developers can check out your code, download it, and edit it as they see fit.

Common Terminology

There is a lot of terminology when it comes to version control. It can be helpful to go over some of the most common terms and what they mean.

Branch

A set of files under version control may be *branched* or *forked* at a point in time. This allows developers to work on a new feature independently from another branch.

Changelog / Changeset

A *changelog* describes a change in one or more files that are made in a single *commit*. They are usually identified with a unique ID.

Checkout

Checking out code is creating a local copy from a remote repository. You can specify a specific revision to check out or check out the latest. If you checkout a file from a centralized system, other users may be disallowed from editing that file while it is checked out.

Clone

Cloning a repository means that you are creating a copy of the repository. This is analogous to checking out code from a centralized versioning system.

Commit

Commit refers to saving your code to the version control system or repository. Commit is also known as **checkin**.

Diff

A *diff* is a comparison of two versions of a file. This helps you visually compare the files so that you can see what changes were made.

Fork

See *branch*

Head

Also known as *tip*. This refers to the most recent commit either to the active branch or trunk. The trunk and the branch have their own head, although sometimes “head” is used only to refer to the trunk.

Initialize

Create a new, empty repository in the version control system.

Mainline

Similar to trunk, although there can be a mainline for each branch

Merge

A merge is taking your changes and merging them into a file or files. For example, you might need to sync your changes to the server. The syncing operation is merging your changes. Another common scenario is when you have a branch of files and you need to merge them to mainline or trunk.

Pull / Push

Copy revisions from one repository to another. A *pull* is initiated by the receiving repository while a *push* is initiated at the source.

Pull Request

A developer creates a pull request (PR) which is how you ask other developers to review your code.

Repository

The location where the files' current and historical changes are stored.

Resolve

Resolving a file involves the developer looking at a merge conflict. This happens when two people edit a file at the same time and then the developer has to figure out how to merge both the changes together with the original.

Stream

A container for branched files that also has a relationship to other containers. Popular in Perforce.

Tag

A tag or label is an important snapshot of the code at a specific point. Usually the tag has a user-friendly name. Tags can be used for releases of software.

Trunk

A unique line of development that is not a branch. Also known as *mainline* or *baseline*.

Python IDE Version Control Support

The most popular Python IDEs have version control support built-in. For example, PyCharm supports Git, Mercurial, Perforce and Subversion. WingIDE has similar capabilities.

If the IDE doesn't have built-in support for your version control system, then you will need to use a separate tool or set of tools to check out the code and commit it back to the repository. When you do have full support for your version control system built-in, then you can do all of that from within your editor. This streamlines working on your codebase and makes working on code much nicer.

When choosing a Python IDE, be sure to verify that your IDE works with your version control system. While this is not a deal-breaker, it certainly makes then easier!

Wrapping Up

Version control will help you keep track of your changes and make your life easier. While it does mean adding another tool to your toolbox, this tool will protect you from yourself. When you make a bad edit, you can always go back to a previous revision and fix the problem. You can also use version control to quickly see what change(s) have occurred in your code.

Appendix B - Version Control with Git

Version control allows you to keep previous versions of your code. This is very useful when you are editing code as you now have a safety net. If you happen to write code that breaks a feature, you can easily rollback to a previous version of your code. Version control allows you to work on bugs or new features without the need to worry that you are going to ruin your work.

There are many different kinds of version control software. The most popular version control software is called **Git**. In this appendix, you will learn about the following:

- Installing Git
- Configuring Git
- Creating a Project
- Ignoring Files
- Initializing a Repository
- Checking the Project Status
- Adding Files to a Repository
- Committing Files
- Viewing the Log
- Changing a File
- Reverting a File
- Checking Out Previous Commits
- Pushing to Github

Let's learn how to use version control with Git!

Installing Git

Git is available for all major platforms, which includes Windows, MacOS, and Linux. Of course, each operating system has its own method for installing software.

Installing on Windows

For Windows users, it is recommended to go to the following website:

- <https://git-scm.com/>

Simply download the Git installer for your version of Windows and then install it.

Installing on MacOS

Depending on which version of the operating system you have on your Mac, you may already have Git installed. You can check by running this command:

```
1 git --version
```

Depending on your version of MacOS, one of three things will happen:

- You receive an error
- You are prompted to install Git
- It works!

If you receive an error, then you'll need to install Git manually. You can download a Git installer for MacOS from the same link you used for Windows:

- <https://git-scm.com/>

Installing on Linux

Linux is a little different. Instead of downloading a file, you will usually use a Linux command to install Git. This command will work on Debian and Ubuntu variants:

```
1 apt-get install git
```

If you have another flavor of Linux, then you may want to go to this URL to see how to install Git on your system:

- <https://git-scm.com/download/linux>

Once you have Git installed, you can move on to the next section!

Configuring Git

The whole point of Git is to track file changes, regardless of how many or how few people are on your team. If you are a team of one, you still need to configure Git so it can record who is modifying the files. A username and email address are required. If you are working solo and don't plan on ever uploading your code to something like Github, then you can use whatever username and email you want.

However, if you do use Github or something similar, then you will want to use your Github username and email for this. To set these values up in Git, you would run the following commands in your Command Prompt (Windows) or Terminal (MacOS / Linux):

```
1 git config --global user.name "username"  
2 git config --global user.email "username@example.com"
```

You can then use this command to list out what Git settings have been saved in your configuration:

```
1 git config --list
```

There is a Git config file that is saved that contains all this information and more. This URL describes the various locations that Git saves this file:

- <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

Now let's move on and create a project that you want to put under version control!

Creating a Project

A project is a fancy word for whatever it is that you are working on. You can use whatever code you have or you can take the `arithmetic` module you created in [chapter 43](#). Here is the code again for your convenience:

```
1 # arithmetic.py  
2  
3 def add(x, y):  
4     return x + y  
5  
6 def divide(x, y):  
7     return x / y  
8  
9 def multiply(x, y):  
10    return x * y  
11  
12 def subtract(x, y):  
13    return x - y
```

Now you have a simple program that you can use to learn Git with. The first step to learn is how to ignore files that you don't want to track with Git.

Ignoring Files

Git tracks all your program changes. When you run a Python script, it will usually generate a `*.pyc` file which is stored in a `__pycache__` directory, which you don't want to keep in version control. If you are using a Python IDE like PyCharm or WingIDE, they will sometimes create configuration or project files too. These are also not something that is useful to track as they are tied to your computer. They won't help anyone else on your team when they go to edit your project.

To ignore files like this, you can add a special file called `.gitignore` to your project. This is what your project folder should look like now:

```
1 /my_project  
2     .gitignore  
3     arithmetic.py
```

Inside of the `.gitignore` file, you can tell Git what to ignore. Each line is a new file type or folder to ignore. Open up `.gitignore` and add the following text to it:

```
1 *.pyc  
2 *.zip  
3 *.pdf
```

This will ignore pyc files as well as zip and PDF files.

Now you can go ahead and initialize your package to tell Git that you want to start tracking.

Initializing a Repository

To start using Git to version your code, you need to open up a terminal or Command Prompt and navigate to the location of your code file(s). Then run this command:

```
1 git init
```

This will emit the following message: "Initialized empty Git repository in /path/to/your/files/.git". The `.git` folder is a hidden folder used by Git to track changes. A **repository** is the term used to describe the files that are being tracked by a version control system, like Git. You won't be doing anything at with the `.git` folder. However, you should not delete this folder as it contains all the history of your code edits.

Checking the Project Status

Now you have an empty Git repository all set up. You can check your project's status by running the following command:

```
1 git status
```

When you run this command, you should see the following output:

```
1 On branch master
2
3 No commits yet
4
5 Untracked files:
6   (use "git add <file>..." to include in what will be committed)
7
8       arithmetic.py
9
10 nothing added to commit but untracked files present (use "git add" to track)
```

This tells you a few things about Git. The first is that you are in the **branch master**. A branch is a version of your code. When you check your status, you should be in your master branch unless you have specifically told Git to switch to a different branch. For example, you might make a new branch to add a new feature to your code. By having a new branch, you won't break the currently working code and you can test your changes in this separate branch.

The other bit of information here is that Git has detected that there is an untracked file in this repository called `arithmetic.py`. It tells you that you don't have anything to **commit**. A commit is a checkpoint or snapshot of your current code. You haven't committed anything to your Git repository though.

Let's find out how to add a file next!

Adding Files to a Repository

The output from `git status` actually tells you how to add files to your repository. Go back and look at that the last line of output. It mentions using the `add` command. You can add files by name like this:

```
1 git add arithmetic.py
```

Or if you want to add all the files that Git lists as untracked, you would do this:

```
1 git add .
```

The period above tells Git to add all untracked files. Be careful doing that as you may add some files that you didn't intend to. You should always check your status before running the above command to make sure you don't add files that you do not want to add.

Now if you run the `status` command, you should see this output:

```
1 On branch master
2
3 No commits yet
4
5 Changes to be committed:
6   (use "git rm --cached <file>..." to unstage)
7
8       new file:  arithmetic.py
```

Now the `add` command does not actually take a snapshot of the code. What the output above is telling you is that Git has staged the file and it is ready to save the snapshot. To actually save a snapshot, you need to `commit` it, which is what you'll do next.

Committing Files

Committing the file saves a snapshot of your code. When you save a snapshot, you are required to add a message. This message tells Git what you are committing and makes it easier for you to figure out what changes are in which commits in your history.

You can run this command to commit the file:

```
1 git commit
```

This will open up your default text editor where you can enter the commit message. Alternatively, you can add the commit message on the command line, like this:

```
1 git commit -m "This is my first commit"
```

If you do this, you will see the following output:

```
1 [master (root-commit) 7f76a83] This is my first commit
2 1 file changed, 13 insertions(+)
3 create mode 100644 arithmetic.py
```

Success! Now if you check your status, you will see that there is nothing new to commit.

Viewing the Log

As you know, Git is now tracking your code. You can view the commits you have made by using the `log` command, like this:

```
1 git log
```

If you run that command, you will get output that is similar to the following:

```
1 commit 7f76a8347c32c93cb3f47a7ea2bb292e1da386f1 (HEAD -> master)
2 Author: Mike Driscoll <mike@something.org>
3 Date:   Mon Aug 17 14:38:44 2020 -0500
4
5     This is my first commit
```

Git tracks who made the commit, when it was committed and also adds a unique 40-character ID to the commit. If you'd like the commit log messages to be printed out on one line, you can run this command instead:

```
1 git log --pretty=oneline
```

This command's output is less verbose, but it gives you what you need to know:

```
1 7f76a8347c32c93cb3f47a7ea2bb292e1da386f1 (HEAD -> master) This is my first commit
```

Let's see what happens when you modify a file!

Changing a File

To see how Git really works, you need to do more than one commit. Re-open your `arithmetic.py` file and add this function to the end:

```
1 def cos(x):
2     print('Not implemented')
```

Now save the file and run the `status` command again. You should see this output:

```
1 On branch master
2 Changes not staged for commit:
3   (use "git add <file>..." to update what will be committed)
4   (use "git checkout -- <file>..." to discard changes in working directory)
5
6       modified:   arithmetic.py
7
8 no changes added to commit (use "git add" and/or "git commit -a")
```

You can add and commit this in a single command by using `-am`, like this:

```
1 git commit -am "added new function"
```

Now go ahead and re-run the log message like you did before:

```
1 $ git log --pretty=oneline
2 7ea78e14831f077b6ed5489a222b3387146ce39f (HEAD -> master) added new function
3 7f76a8347c32c93cb3f47a7ea2bb292e1da386f1 This is my first commit
```

As you can see, there are now two commits in your code. But what do you do if you decide that your last change was a bad one? Let's find out in the next section!

Reverting a File

When you decide that you saved a bad change, you can recover by reverting that change. To see how this works, go ahead and modify `arithmetic.py` so that it contains some bad code:

```
1 # arithmetic.py
2
3 def add(x, y):
4     return x + y
5
6 def divide(x, y):
7     return x / y
8
9 def multiply(x, y):
10    return x * y
11
12 def subtract(x, y):
13    return x - y
14
```

```
15 def cos(x):
16     print('Not implemented')
17
18 1 / 0
```

That last line will cause an error to be raised if you attempt to run or import the code. Go ahead and save the code, then run `git status`:

```
1 On branch master
2 Changes not staged for commit:
3   (use "git add <file>..." to update what will be committed)
4   (use "git checkout -- <file>..." to discard changes in working directory)
5
6       modified:   arithmetic.py
7
8 no changes added to commit (use "git add" and/or "git commit -a")
```

Git correctly sees that you changed the file, but you haven't committed it yet. You can undo your bad change in your Python editor or you can run this Git command to revert it:

```
1 git checkout .
```

Now if you run `status` again, the output will be different:

```
1 $ git status
2 On branch master
3 nothing to commit, working tree clean
```

You just reverted a non-committed change! This is most useful when you have many files checked out that you don't want to commit. When it's only one or two files, it may be simpler to use your Python editor. That is really up to you.

What do you do in the case where you have committed some code that is breaking something though? You can't undo that easily with your Python editor. Let's see how Git will help you with that issue next!

Checking Out Previous Commits

Git allows you to check out previous commits. This allows you fix your mistake with the old version of the code. You can check out any previous commit by using the first six characters of the unique ID for the commit.

Here's the current log:

```
1 $ git log --pretty=oneline
2 7ea78e14831f077b6ed5489a222b3387146ce39f (HEAD -> master) added new function
3 7f76a8347c32c93cb3f47a7ea2bb292e1da386f1 This is my first commit
```

If you wanted to checkout the first commit, you would do this:

```
1 git checkout 7f76a8
```

When you run this command, you will get the following message:

```
1 Note: checking out '7f76a8'.
2
3 You are in 'detached HEAD' state. You can look around, make experimental
4 changes and commit them, and you can discard any commits you make in this
5 state without impacting any branches by performing another checkout.
6
7 If you want to create a new branch to retain commits you create, you may
8 do so (now or later) by using -b with the checkout command again. Example:
9
10 git checkout -b <new-branch-name>
11
12 HEAD is now at 7f76a83... This is my first commit
```

This means that you have left your master branch temporarily. Git calls this a *detached HEAD state*. HEAD refers to the current commit state of your project. But you're not committing anything right now. You are "detached" from HEAD instead because you're no longer in *master*.

When you're done looking around, you can return to the **master** branch by using the following command:

```
1 git checkout master
```

If you want to rollback to a previous commit permanently, you can use the **reset** command:

```
1 git reset --hard 7f76a8
```

You will need to use **--hard** to make this work correctly. If you were to run this, you would rollback to the very first commit in your repository. Use **reset** with caution. You don't want to accidentally wipe out your work.

Pushing to Github

Most of the time, software development occurs with more than one developer. You can do your work on your own machine, but then you need to push your work to a central location so that other people on your team can use it. There are many popular websites that specialize in this space. Github is the most popular, but there are others such as GitLab and Bitbucket that are also popular.

To push your code to Github, you will need to go there and create an account:

- <https://github.com/>

Next, you will need to create a new repository. Here is the repository for this book's code:

- <https://github.com/driscollis/python101code>

Let's take this piece-by-piece. After the Github address, you see a username: **driscollis**. That is followed by the repository name **python101code**. So if you were adding the `arithmetic` code to Github, the URL would look something like this:

- <https://github.com/USERNAME/arithmetic>

To tell Git that you want to push your code here, you need to run the following command in your terminal:

```
1 git remote add origin https://github.com/USERNAME/arithmetic.git
```

This command will tell Git that you want to push code to this specific repository on Github. However, it doesn't actually send the code there. You need to do that by running the following:

```
1 git push origin master
```

This command tells Git to push or upload your code to Github. After that, you should be able to use `git push` to upload any other changes you make to your code.

Wrapping Up

Git is a very useful tool. Whether or not you choose to work with Git is up to you. However, you should learn how to use at least one of the most popular version control systems. Git is the most popular. Once you have learned how to use it, you can migrate to most other version control software and get up to speed with them quickly.

Using Git will help you become a better programmer. Remember to commit often. It's a lot easier to roll back a small change than it is a large one.

Review Question Answer Key

Chapter 3 - Documenting Your Code

1) How do you create a comment?

By using the pound or octothorpe key: #. Here are a couple of examples:

```
1 # This is a comment  
2 a = 2 # this is an in-line comment
```

2) What do you use a docstring for?

Docstrings are used for documenting functions, classes or modules. You can learn more about functions and classes in chapters 17 and 18 respectively.

3) What is Python's style guide?

It is a guide to formatting Python code so that it is consistent between authors and easier to read. Python's style guide is defined in PEP8. You can find it here:

- <https://www.python.org/dev/peps/pep-0008/>

4) Why is documenting your code important?

Documenting your code is important because it will help you understand the code better later on. It will also help other developers who need to use or modify your code. This is especially true when you are documenting complex code or algorithms.

Chapter 4 - Working with Strings

1) What are 3 ways to create a string?

Strings are defined by starting and ending a string of characters with a single, double or three single or double quotes:

```
1 >>> string_1 = 'Mike'  
2 >>> string_2 = "likes"  
3 >>> string_3 = '''Python!'''
```

Triple quotes can be used to create multi-line strings.

2) Run `dir("")`. This lists all the string methods you can use. Which of these methods will capitalize each of the words in a sentence?

`.title()` will capitalize every first letter of each word. `capitalize()` will only capitalize the first letter in a string.

3) Change the following example to use f-strings:

```
1 >>> name = 'Mike'  
2 >>> age = 21  
3 >>> print('Hello %s! You are %i years old.' % (name, age))  
4 Hello Mike! You are 21 years old.
```

You can modify the last line of the code above to use f-strings like this:

```
1 >>> print(f'Hello {name}! You are {age} years old.')
```

4) How do you concatenate these two strings together?

```
1 >>> first_string = 'My name is'  
2 >>> second_string = 'Mike'
```

You can use the `+` operator to concatenate strings:

```
1 >>> first_string + second_string  
2 'My name isMike'
```

Note that it will join them together without a space between them, so you may need to account for that.

You can also use the `string.join()` method:

```
1 >>> ' '.join([first_string, second_string])
2 'My name is Mike'
3
4 ### 5) Use string slicing to get the substring, "is a", out of the following string:
5
6 ````python
7 >>> 'this is a string'
```

Here are three ways to do this:

```
1 >>> 'this is a string'[5:9]
2 'is a'
3 >>> 'this is a string'[5:-7]
4 'is a'
5 >>> 'this is a string'[-11:-7]
6 'is a'
```

Chapter 5 - Numeric Types

1) What 3 numeric types does Python support without importing anything?

The 3 built-in numeric types are:

- int
- float
- complex

2) Which module should you use for money or other precise calculations?

You should use the decimal module for precise calculations:

- <https://docs.python.org/3/library/decimal.html>

3) Give an example of how to use augmented assignment.

```
1 >>> x = 10
2 >>> x *= 5
3 >>> x
4 50
```

Chapter 6 - Learning About Lists

1) How do you create a list?

A list is a sequence enclosed in square braces:

```
1 >>> empty_list = []
2 >>> my_list = [1, 2, 3] # list with 3 integers
```

2) Create a list with 3 items and then use append() to add two more.

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list.append(4)
3 >>> my_list
4 [1, 2, 3, 4]
5 >>> my_list.append(7)
6 >>> my_list
7 [1, 2, 3, 4, 7]
```

3) What is wrong with this code?

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list.remove(4)
```

This will cause a `ValueError` to be raised because you are trying to remove an item, 4, that does not exist.

4) How do you remove the 2nd item in this list?

```
1 >>> my_list = [1, 2, 3]
```

The most effective way is to use `pop()`:

```
1 >>> my_list.pop(1)
2 2
3 >>> my_list
4 [1, 3]
```

`pop()` takes the index of the item that you want to remove. Remember that items in a list start at index zero.

5) Create a list that looks like this: [4, 10, 2, 1, 23]. Use string slicing to get only the middle 3 items.

Here are two ways to do it:

```
1 >>> my_list = [4, 10, 2, 1, 23]
2 >>> my_list[1:4]
3 [10, 2, 1]
4 >>> my_list[1:-1]
5 [10, 2, 1]
```

Chapter 7 - Learning About Tuples

1) How do you create a tuple?

Tuples are sequences of objects (strings, integers, etc) separated by commas and optionally enclosed within parentheses:

```
1 >>> my_tuple = (1, 2, 3)
2 >>> your_tuple = 4, 5, 6
```

2) Can you show how to access the 3rd element in this tuple?

```
1 >>> a_tuple = (1, 2, 3, 4)
```

You can access the 3rd element like this:

```
1 >>> a_tuple = (1, 2, 3, 4)
2 >>> a_tuple[2]
3 3
```

Remember, tuples start at index zero.

3) Is it possible to modify a tuple after you create it? Why or why not?

No. Tuples are immutable, which means you cannot change them after they are created.

4) How do you create a tuple with a single item?

```
1 >>> single_tuple = (1, )  
2 >>> single_tuple  
3 (1, )
```

Chapter 8 - Learning About Dictionaries

1) How do you create a dictionary?

Dictionaries are creating using curly braces: {}. They are a series of key: value pairs.

Here is an example:

```
1 >>> my_dict = {1: 'one', 2: 'two'}
```

2) You have the following dictionary. How do you change the last_name field to 'Smith'?

```
1 >>> my_dict = {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
```

You can change a dictionary's key mapping by writing out the dictionary variable, followed by square braces with the name of the key inside of it:

```
1 >>> my_dict['last_name'] = 'Smith'  
2 >>> my_dict  
3 {'email': 'jdoe@gmail.com', 'first_name': 'James', 'last_name': 'Smith'}
```

3) Using the dictionary above, how would you remove the email field from the dictionary?

By using the pop() method:

```
1 >>> my_dict.pop('email')
2 'jdoe@gmail.com'
3 >>> my_dict
4 {'first_name': 'James', 'last_name': 'Smith'}
```

4) How do you get just the values from a dictionary?

You can use the `values()` method:

```
1 >>> my_dict = {'first_name': 'James', 'last_name': 'Doe', 'email': 'jdoe@gmail.com'}
2 >>> my_dict.values()
3 dict_values(['James', 'Doe', 'jdoe@gmail.com'])
```

Chapter 9 - Learning About Sets

1) How do you create a set?

Sets are created using curly braces, {}, but instead of key, value pairs, you will use a sequence of objects:

```
1 >>> my_set = {1, 'two', 3}
```

2) Using the following set, how would you check to see if it contains the string, "b"?

```
1 >>> my_set = {"a", "b", "c", "c"}
```

You can use Python's `in` keyword to test if the set contains something:

```
1 >>> 'b' in my_set
2 True
```

3) How do you add an item to a set?

You can use the set's `add()` method:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.add(1)  
3 >>> my_set  
4 {1, 'a', 'b', 'c'}
```

4) Remove the letter "c" from the following set using a set method:

```
1 >>> my_set = {"a", "b", "c", "c"}
```

You can use `remove()` or `discard()`.

Here is an example using `remove()`:

```
1 >>> my_set = {"a", "b", "c", "c"}  
2 >>> my_set.remove('c')  
3 >>> my_set  
4 {'a', 'b'}
```

5) How do you find the common items between two sets?

You can use the `intersection()` method to find the common items between two sets:

```
1 >>> first_set = {'one', 'two', 'three'}  
2 >>> second_set = {'orange', 'banana', 'peach', 'one'}  
3 >>> first_set.intersection(second_set)  
4 {'one'}
```

Chapter 10 - Boolean Operations and None

1) What number does `True` equal?

The integer value, 1.

2) How do you cast other data types to `True` or `False`?

By using Python's `bool()` function.

3) What is Python's null type?

`None`

Chapter 11 - Conditional Statements

1) Give a couple of examples of comparison operators:

<, >, ==, !=

2) Why does indentation matter in Python?

Indentation matters because it tells Python where a block of code begins and ends. Also, indentation must be consistent (tabs or spaces, not both).

3) How do you create a conditional statement?

You can create a conditional statement in Python using the keyword `if` followed by an expression that evaluates to `True` or `False`:

```
1  >>> x = 15
2  >>> if x > 10:
3  ...     print(f'{x=}')
4  ...
5  x=15
```

4) How do you use logical operators to check more than one thing at once?

To check multiple things at once, you can use `and` or `or`:

```
1  >>> if x > 10 and y < 15:
2      print('The value falls in range')
```

5) What are some examples of special operators?

`is`, `is not`, `in` and `not in`

6) What is the difference between these two?

```
1 x = [4, 5, 6]
2 y = [4, 5, 6]
```

and

```
1 x = [4, 5, 6]
2 y = x
```

The first example creates two separate list objects, which have different ids (or memory addresses). The second example creates a new alias for x – meaning that x and y both point to the same object with the same id.

Chapter 12 - Learning About Loops

1) What two types of loops does Python support?

You can create while loops or for loops in Python.

2) How do you loop over a string?

```
1 >>> my_string = 'the fox jumps'
2 >>> for char in my_string:
3     ...     print(char)
```

3) What keyword do you use to exit a loop?

break

4) How do you “skip” over an item when you are iterating?

By using the continue keyword.

5) What is the else statement for in loops?

The else statement only gets executed if break never occurs. You may use it to raise an error if you are looping over something and never finds a match.

6) What are the flow control statements in Python?

- if
- elif
- else
- for
- while
- continue
- break

Chapter 13 - Python Comprehensions

1) How do you create a list comprehension?

You can create a list comprehension by creating a specially formatted `for` loop inside of square braces with an optional expression.

Here's an example that filters out even integers:

```
1 >>> my_list = [1, 2, 3, 4, 5, 6, 7]
2 >>> list_comp = [x for x in my_list if x % 2]
3 >>> list_comp
4 [1, 3, 5, 7]
```

2) What is a good use case for a list comprehension?

They are a great way to apply a filter to a list.

3) Create a dictionary using a dict comprehension.

```
1 >>> {key: value for key, value in enumerate('abcde')}
2 {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

4) Create a set using a set comprehension.

```
1 >>> my_set = {item for item in range(5)}
2 >>> my_set
3 {0, 1, 2, 3, 4}
4 >>> type(my_set)
5 <class 'set'>
```

Chapter 14 - Exception Handling

1) What are a couple of common exceptions?

ModuleNotFoundError, IndexError, and ValueError are pretty common ones.

2) How do you catch an exception in Python?

By using a try/except statement.

3) What do you need to do to raise a run time error?

You would use the following: raise RuntimeError.

4) What is the finally statement for?

This is where you would put your code for cleaning up after an exception occurs. For example, if you had an exception while working with a database, you would want to close the connection to the database here.

5) How is the else statement used with an exception handler?

The else code executes when no errors occur.

Chapter 15 - Working with Files

1) How do you open a file?

You use Python's open() function and pass in a relative or fully qualified path to a file.

2) What do you need to do to read a file?

```

1 with open('path/to/some/file.txt') as fh:
2     contents = fh.read()

```

3) Write the following sentence to a file named test.txt:

1 The quick red fox jumped over the python

You will need to open a file in write-mode to write something to it:

```

1 with open('test.txt', 'w') as fh:
2     fh.write('The quick red fox jumped over the python')

```

4) How do you append new data to a pre-existing file?

```

1 # open the file in append-mode
2 with open('test.txt', 'a') as fh:
3     fh.write('This is some new text')

```

5) What do you need to do to catch a file exception?

Most of the time, you only need to catch the OSError:

```

1 try:
2     with open('test.txt') as file_handler:
3         for line in file_handler:
4             print(line)
5 except OSError:
6     print('An error has occurred')

```

Chapter 16 - Importing

1) How do you include the `math` library from Python's standard library in your code?

You need to use the `import` keyword followed by the library you wish to import:

```
1 import math
```

Put the code above at the top of your file.

2) How do you include `cos` from the `math` library in your own code?

If all you need from the `math` module is `cos()` function, you can import it like this:

```
1 from math import cos
```

3) How do you import a module/function/etc. with a different name?

By adding the as keyword and the new name to use:

```
1 from math import cos as cosine
```

4) Python has a special syntax you can use to include everything. What is it?

```
1 from math import *
```

You can replace math with whichever library you wish to import everything from. Note that this is NOT recommended!

Chapter 17 - Functions

1) How do you create a function that accepts two positional arguments?

```
1 def my_function(a, b):
2     print(a)
3     print(b)
```

2) Create a function named address_builder that accepts the following and add type hints:

- name (string)
- address (string)
- zip code (integer)

```
1 def address_builder(name: str, address: str, zip_code: int) -> str:
2     return f'{name} - {address} - {zip_code}'
```

3) Using the function from question 2, give the zip code a default of 55555

```
1 def address_builder(name: str, address: str, zip_code: int=55555) -> str:  
2     return f'{name} - {address} - {zip_code}'
```

4) How do you allow an arbitrary number of keyword arguments to be passed to a function?

Use `**kwargs` to allow an arbitrary number of keyword arguments to be passed in:

```
1 def keywords(**kwargs):  
2     print(kwargs)
```

5) What syntax do you use to force a function to use positional-only parameters?

After defining some positional-only parameters, you can add a forward-slash, “/”. This makes the parameters that come before it positional-only:

```
1 def positional(a, b, /, c):  
2     print(a)
```

In the example above, `a` and `b` are positional-only arguments.

Chapter 18 - Classes

1) How do you create a class in Python?

You can create a `class` by using the `class` keyword followed by the name of the `class` and a colon:

```
1 class MyClass:  
2     pass
```

2) What do you name a class initializer?

Initializers in classes have a special name called `__init__()`. An initializer is a method inside of a class and is usually the first one defined.

3) Explain the use of `self` in your own words

The word `self` is used internally by the class to keep track of instances.

4) What does overriding a method do?

When you override a method, you are redefining what it does. For example, if the parent class has a `jump()` method, its child may redefine it in such a way that its `jump()` behaves differently. Perhaps the parent is measured in feet while the child uses the Metric system.

5) What is a subclass?

A subclass will inherit all the methods from its parent or base class. You can use it as a simple way to create a new class that is similar to one of your old ones, but has new features.

```
1 class MyBall(Ball):
2     """This class inherits from the Ball class"""
3
4     def roll(self):
5         print('rolling the ball')
```

Chapter 19 - Introspection

1) What is introspection?

Introspection is the ability for code to investigate itself and other code using tools that are built into your programming language. Python lets you do introspection of code easily using `dir()`, `type()`, `help()`, and others.

2) What is the `type()` function used for?

You can use `type()` to figure out the type of an object. For example, you might need to know if the type of an object is a string versus a numeric type.

3) How is `dir()` helpful?

You can use `dir()` to learn what attributes and functions or methods an object has (if any).

Chapter 20 - Installing Packages with pip

1) How do you install a package with `pip`?

```
1 python -m pip install <package>
```

2) What command do you use to see the version of the packages you installed?

```
1 python -m pip list
```

3) How do you uninstall a package?

```
1 python -m pip uninstall <package>
```

Chapter 21 - Python Virtual Environments

1) How do you create a Python virtual environment?

If you use Python's `venv` module, you can create one like this:

```
1 python -m venv env_name
```

If you use `virtualenv`, then the command is:

```
1 virtualenv env_name
```

2) What do you need to do after creating a virtual environment to use it?

You need to activate it. On Mac/Linux, you can activate the environment by running the following inside the folder you created:

```
1 source bin/activate
```

On Windows, you need to run the `activate.bat` file or the Powershell file equivalent.

3) Why would you use a Python virtual environment?

Python virtual environments are useful for isolating your code. You can test out new packages without installing them to your system Python.

Chapter 22 - Type Checking in Python

1) What is type hinting in your own words?

Type hinting in Python is a special syntax that tells the developer what data type a variable or argument is. Type hints are optional and are not enforced in Python. They can be checked using an IDE or the Mypy tool.

2) Why would you use type hinting?

Type hinting is helpful in large, complex code bases. It can also be helpful if you are planning to create a Python package that you wish to distribute to others on the Python Packaging Index (PyPI). The type hints will make it easier for others to contribute to your code.

3) Demonstrate your understanding of type hinting by adding type annotations to the variables as well as the function. Don't forget the return type!

Here is one way to add type hints:

```
1 from typing import List, Optional
2
3 a: int = 1
4 b: float = 3.14
5
6 def my_function(x: List[int] = [], y: Optional[int] = None) -> List[int]:
7     if y is not None:
8         result = [i * y for i in x]
9     else:
10        result = x
11    return result
12
13
14 my_function([1, 2, 3], 2)
```

Chapter 23 - Creating Multiple Threads

1) What are threads good for?

Threads are best used for applications that will be doing I/O intensive activities, such as writing files, downloading files, working with databases, etc.

2) Which module do you use to create a thread in Python?

You will use the `threading` module. Most of the time, you will be using the `threading.Thread` class.

3) What is the Global Interpreter Lock?

The Global Interpreter Lock is a mutex that protects Python objects. This means that it prevents multiple threads from executing Python bytecode at the same time. So when you use threads, they do not run on all the CPUs on your machine.

Chapter 24 - Creating Multiple Processes

1) What are processes good for?

Processes are good for complex mathematical computations, encryption and search.

2) How do you create a process in Python?

You can create a process using the `multiprocessing` module, specifically the `Process` class.

3) Can you create a process pool in Python? How?

Yes you can. You can use `multiprocessing.Pool` to create one.

4) What effect, if any, does the Global Interpreter Lock have on processes?

Using the `multiprocessing` module avoids the Global Interpreter Lock, so it does not have any effect.

5) What happens if you don't use `process.join()`?

The point of using `join()` is to make the main process (your application) wait for the child processes to finish. If you don't call `join()`, your main process may try to end prematurely.

Chapter 25 - Launching Subprocesses with Python

1) How would you launch Microsoft Notepad or your favorite text editor with Python?

```
1 import subprocess  
2  
3 cmd = ['notepad.exe']  
4 subprocess.run(cmd)
```

2) Which method do you use to get the result from a process?

You can use the `Popen.communicate()` method.

3) How do you get `subprocess` to return strings instead of bytes?

One way to get `subprocess` to return strings instead of bytes is to set the `encoding` argument to `utf-8`.

Chapter 26 - Debugging Your Code

1) What is `pdb`?

`pdb` is Python's built-in debugger. It is also a module that you can import into your code to add debugging directly.

2) How do you use `pdb` to get to a specific location in your code?

When inside of the `Pdb` debugger in the terminal or command line, use the `jump` command followed by the line number that you wish to go to.

3) What is a breakpoint?

A `breakpoint` is a position in a file that is marked in some way that tells the debugger to stop execution there. This allows the developer to tell their debugger or their IDE to stop at a specific point in their code so that they can analyze the variables and state of objects in the hopes of figuring out an issue.

4) What is a callstack?

A call stack is the data structure that Python uses to keep track of function and method calls.

Chapter 27 - Learning About Decorators

1) What is a decorator?

A decorator is a function that accepts another function as its argument. A decorator often adds a wrapper around the function to do pre- or post-processing of the data the function uses.

2) What special syntax do you use to apply a decorator?

To apply a decorator function onto another function, you use the @ sign followed by the name of the decorator.

3) Name at least two of Python's built-in decorators

- `property`
- `classmethod`
- `staticmethod`

4) What is a Python property?

Python properties are a way of adding code to handle attribute lookups and assignments while still using normal attribute access in your code.

Chapter 28 - Assignment Expressions

1) What is an assignment expression?

An assignment expression is a new syntax that allows you to assign to a variable within an expression (such as within a `while` or `if` statement).

2) How do you create an assignment expression?

The general syntax is as follows:

1 NAME := expr

3) Why would you use assignment expressions?

You can use assignment expressions to avoid repeated function calls, attribute lookups, boolean checks, etc.

Chapter 29 - Profiling Your Code

1) What does “profiling” mean in a programming context?

To use a tool to run your code and help you find CPU and memory issues or bottlenecks. The goal is to create a fast and memory efficient program whenever possible.

2) Which Python library do you use to profile your code?

You can use the `cProfile` module to profile your code.

3) How do you extract data from saved profile statistics?

You can use the `pstats` module to extract, filter and format profile statistics.

Chapter 30 - An Introduction to Testing

1) How do you add tests to be used with `doctest`?

You can add the test to a docstring. However, you must use the proper syntax and make it look like a Python session:

```
1 >>> add(1, 2)
2 3
```

2) Are tests for `doctest` required to be in the docstring? If not, how would you execute it?

No, they can also be in a separate file. When you execute them from a separate file, you pass that file to `doctest` instead of the Python file.

3) What is a unit test?

A unit test is a way to test the smallest piece of code that can be isolated from the rest.

4) What is test driven development?

The idea behind test driven development is that you should write the tests before you write the code. At the very least, you should write your tests while you write the code to be sure that you are testing everything.

Chapter 31 - Learning About the Jupyter Notebook

1) What is Jupyter Notebook?

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contains code, equations, visualizations and formatted text.

2) Name two Notebook cell types

- Code
- Markdown
- Raw NBConvert
- Heading

3) What markup language do you use to format text in Jupyter Notebook?

Markdown

4) How do you export a Notebook to another format?

You can use nbconvert or use the *File* menu with the *Download as* option to convert the Notebook to other formats.

Chapter 32 - How to Create a Command Line Application with argparse

1) Which module in the standard library can you use to create a command-line application?

You can use the argparse module.

2) How do you add arguments to the ArgumentParser()?

By using the add_argument() method.

3) How do you create helpful messages for your users?

When adding an argument with the add_argument() method, you can specify what the help message is by setting the help argument.

4) Which method do you use to create a mutually exclusive group of commands?

You will use the `add_mutually_exclusive_group()` method to create a group that you can add mutually exclusive arguments to.

Chapter 33 - How to Parse XML

1) What XML modules are available in Python's standard library?

- `xml.etree.ElementTree`
- `xml.dom`
- `xml.dom.minidom`
- `xml.dom.pulldom`
- `xml.sax`
- `xml.parsers.expat`

2) How do you access an XML tag using ElementTree?

Usually you will need to do this by looping / iterating over the elements and then accessing its `tag` property.

3) How do you get the root element using the ElementTree API?

To get to the root element, you need to load the XML file into the `ElementTree` class and call `getroot()`:

```
1 tree = ElementTree(file=xml_file)
2 root_element = tree.getroot()
```

Chapter 34 - How to Parse JSON

1) What is JSON?

JavaScript Object Notation is a lightweight data interchange format used especially for web applications.

2) How do you decode a JSON string in Python?

You would use `json.loads()` to decode or deserialize a JSON string in Python.

3) How do you save JSON to disk with Python?

You would use `json.dump` to save JSON to disk.

Chapter 35 - How to Scrape a Website

1) What are some popular Python web scraping packages?

BeautifulSoup and Scrapy are the most popular.

2) How do you examine a web page with your browser?

Right-click anywhere on a web page and choose the “Inspect Element” (Mozilla Firefox) or “Inspect” (Google Chrome) menu option.

3) Which Python module from the standard library do you use to download a file?

`urllib.request`

Chapter 36 - How to Work with CSV files

1) How do you read a CSV file with Python's standard library?

You must use the `csv` module and create a reader object using either `reader` or `DictReader`.

2) If your CSV file doesn't use commas as the delimiter, how do you use the `csv` module to read it?

When you create the `reader()`, you can set the `delimiter` argument to something other than comma. Here's an example:

```
1 import csv
2
3 with open(path) as f:
4     reader = csv.reader(f, delimiter=';')
```

3) How do you write a row of CSV data using the `csv` module?

You would need to create a writer object using either `writer()` or `DictWriter()`. Then you would use the `writerow()` function to write data.

Chapter 37 - How to Work with a Database Using `sqlite`

1) How do you create a database with the `sqlite3` library?

You can use the following code:

```
1 import sqlite3  
2 sqlite3.connect("/path/to/database.db")
```

2) Which SQL command is used to add data to a table?

You would use `INSERT`

3) How do you change a field in a database with SQL?

Use SQL's `UPDATE` command

4) What are SQL queries used for?

SQL queries are for getting data out of the database. You can extract everything or use filters to narrow it down to just the output you desire.

5) By default, how many records in a table will `DELETE` affect? How about `UPDATE` and `SELECT`?

All of them.

6) The `delete_author` function above is susceptible to an SQL Injection attack. Why, and how would you fix it?

The `delete_author()` function is using Python's string formatting and passed in data to create the query. To avoid the issue, use the `?` syntax instead:

```
1 def delete_author(author):
2     conn = sqlite3.connect("library.db")
3     cursor = conn.cursor()
4     sql = "DELETE FROM books WHERE author=?"
5     cursor.execute(sql, [author])
6     conn.commit()
```

Chapter 38 - Working with an Excel Document in Python

1) What Python package can you use to work with Microsoft Excel spreadsheets?

You can use OpenPyXL, xlwings or several other 3rd party Python packages to work with Excel.

2) How do you open an Excel spreadsheet with Python?

You would use something like this:

```
1 import openpyxl
2
3 workbook = openpyxl.load_workbook('/path/to/excel/file.xlsx')
```

3) Which class do you use to create an Excel spreadsheet with OpenPyXL?

You would use `openpyxl.Workbook()` to create a new document.

Chapter 39 - How to Generate a PDF

1) What class in ReportLab do you use to draw directly on the PDF at a low-level?

You use the `Canvas()` class from `reportlab.pdfgen`.

2) What does PLATYPUS stand for?

Page Layout and Typography Using Scripts.

3) How do you apply a stylesheet to a Paragraph?

You need to get a stylesheet. One of the most popular ways is to use the sample stylesheet:

```
1 styles = getSampleStyleSheet()
```

Then you apply the style by passing it in to the `Paragraph()` class via the `style` flag:

```
1 para = Paragraph(text, style=styles["Normal"])
```

4) Which method do you use to apply a `TableStyle`?

You use the `Table()` object's `setStyle()` method to apply the `TableStyle()`.

Chapter 40 - How to Create Graphs

1) Which module in Matplotlib do you use to create plots?

You use the `pyplot` sub-module: `import matplotlib.pyplot as plt`.

2) How do you add a label to the x-axis of a plot?

You would use the `label()` function of the plot module.

3) Which functions do you use to create titles and legends for plots?

The `title()` and `legend()` functions, respectively.

Chapter 41 - How to Work with Images in Python

1) How do you get the width and height of a photo using Pillow?

You can get the width and height by accessing the `size` attribute of the image object.

2) Which method do you use to apply a border to an image?

You would use the `expand()` method.

3) How do you resize an image with Pillow while maintaining its aspect ratio?

You would want to use the `thumbnail()` method.

Chapter 42 - How to Create a Graphical User Interface

1) What is a GUI?

A GUI, or Graphical User Interface, is a way for people to interact with your application using buttons or other widgets, their keyboard, and their mouse. This is in contrast to how you would interact with a console-only application where you are working in a terminal using your keyboard almost exclusively.

2) What is an event loop?

The **event loop** is an infinite loop that runs within your GUI application. It waits for the user to “do something”, like click a button, press a key on the keyboard or interact with the GUI in some other way. When the user does these things, it generates one or more events that your application can respond to.

3) How do you layout widgets in your application?

You have two choices:

- Absolute positioning - using x/y pixel coordinates for each widget
- Sizers - A special object in wxPython that contains widgets and lays them out dynamically as you resize the application.

Chapter 43 - How to Create a Python Package

1) What is a module?

A module is a Python file. Any Python file can be imported.

2) How is a package different from a module?

A Python package is a directory with one or more files inside of it. When you import a package, you are basically importing a directory.

3) Name at least two tools that are required for packaging a project?

You need pip, setuptools and twine to package and upload a package.

4) What are some of the files you need to include in a package that are not Python files?

The README.md and LICENSE files.

Chapter 44 - How to Create an Exe for Windows

1) Name 3 different tools you can use to create a Windows executable out of Python code.

PyInstaller, py2exe, Briefcase, Nuitka, cx_freeze.

2) What command do you use with PyInstaller to create an executable?

You would use this one:

```
1 pyinstaller my_python_script.py
```

If PyInstaller is not on your path, then you would need to modify the command to use the full path to PyInstaller:

```
1 /path/to/pyinstaller my_python_script.py
```

3) How do you create a single file executable with PyInstaller?

You would need to use the --onefile flag:

```
1 pyinstaller my_python_script.py --onefile
```

4) Which flag do you use with PyInstaller to suppress the console window?

You would need to use the --noconsole flag:

```
1 pyinstaller my_python_script.py --noconsole
```

Chapter 45 - How to Create an Installer for Windows

1) Name two programs you can use to create installers for Windows.

- Inno Setup
- NSIS

2) How do you modify an Inno Setup compiler script?

You can use any text editor to modify the compiler script as it is a text file. Inno Setup itself can be used to edit the file.

3) Why should you test your installers?

You should always test out your installers to make sure they work before giving them to your users. You will get a bad reputation quite quickly if your installers don't work.

Chapter 46 - How to Create an “exe” for Mac

1) What tools can you use to create executables for Python on MacOS?

- PyInstaller
- Briefcase
- py2app

2) What is an executable called on MacOS?

They are known as “app bundles” on Mac.