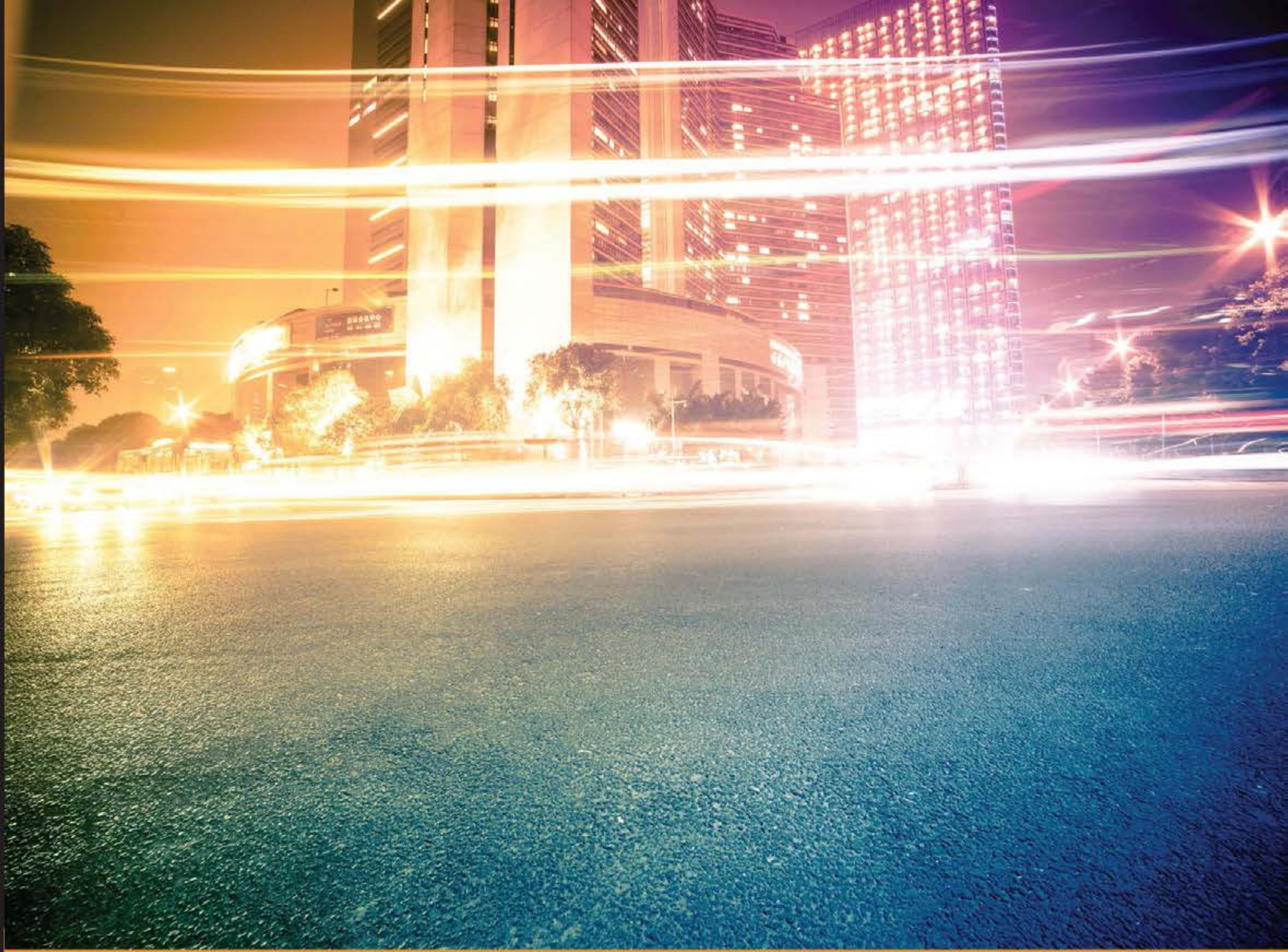


Community Experience Distilled

Learning Python for Forensics

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

Preston Miller   Chapin Bryce

[PACKT] PUBLISHING open source* community experience distilled

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

BIRMINGHAM - MUMBAI

Learning Python for Forensics

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

Reviewer
Sachin Raste

Commissioning Editor
Rebecca Youe

Acquisition Editor
Vinay Argekar

Content Development Editor
Sanjeet Rao

Technical Editor
Pramod Kumavat

Copy Editors
Dipti Mankame
Laxmi Subramanian

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Indexer
Hemangini Bari

Graphics
Kirk D'Penha

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

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.

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 metdata – 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 _list_files() method 412
Developing the Framework _run_plugins() method 414
Exploring the Plugin object 416
Understanding the Plugin __init__() constructor 416
Working with the Plugin run() method 416
Handling output with the Plugin write() method 417
Exploring the Writer object 418
Understanding the Writer __init__() constructor 418
Understanding the Writer run() method 419
Our Final CSV writer – csv_writer.py 419
The writer – xlsx_writer.py 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 are 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 by 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
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

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 (MantaRay[1]), 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

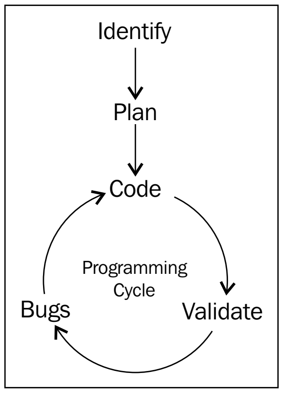
The development life cycle involves at least five steps:

- 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 JetBrain's 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
```

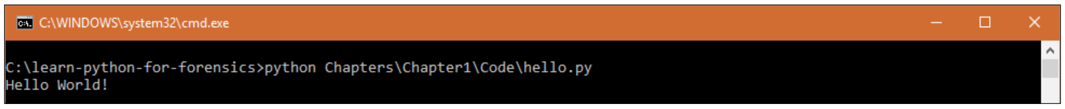
Notice how some methods, such as `__add__()`, have double leading and trailing underscores. These are referred to as **magic methods** and they are the methods the Python interpreter calls and they should not be called by the programmer. These magic methods are instead called indirectly by the user. 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.



```
C:\learn-python-for-forensics>python Chapters\Chapter1\Code\hello.py
Hello World!
```

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

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.

```
# This is a comment
print 5 + 5 # This is an inline comment. Everything to the right of the # symbol does not get executed
"""We can use three quotes to create
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\ncan 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 it's 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('')
'===================='
```

Integers 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'
```