CRACKING CODES WITH PYTHON

AN INTRODUCTION TO BUILDING AND BREAKING CIPHERS

AL SWEIGART

no starch press

CRACKING CODES WITH PYTHON

CRACKING CODES WITH PYTHON

An Introduction to Building and Breaking Ciphers

by Al Sweigart

no starch
press

San Francisco

ISBN-10: 1-59327-822-5
ISBN-13: 978-1-59327-822-9

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover Illustration: Josh Ellingson
Interior Design: Octopod Studios
Developmental Editors: Jan Cash and Annie Choi
Technical Reviewers: Ari Lacenski and Jean-Philippe Aumasson
Copyeditor: Anne Marie Walker
Compositors: Riley Hoffman and Meg Sneeringer
Proofreader: Paula L. Fleming

For information on distribution, translations, or bulk sales,
please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Names: Sweigart, Al, author.
Title: Cracking codes with Python : an introduction to building and breaking
   ciphers / Al Sweigart.
Description: San Francisco : No Starch Press,Inc., [2018]
Identifiers: LCCN 2017035704 (print) | LCCN 2017047589 (ebook) | ISBN
   9781593278694 (epub) | ISBN 1593278691 (epub) | ISBN 9781593278229 (pbk.)
   | ISBN 1593278225 (pbk.)
Subjects: LCSH: Data encryption (Computer science) | Python (Computer program
   language) | Computer security. | Hacking.
Classification: LCC QA76.9.A25 (ebook) | LCC QA76.9.A25 S9317 2018 (print) |
   DDC 005.8/7--dc23
LC record available at https://lccn.loc.gov/2017035704
```

Dedicated to Aaron Swartz, 1986–2013

"Aaron was part of an army of citizens that believes democracy only works when the citizenry are informed, when we know about our rights—and our obligations. An army that believes we must make justice and knowledge available to all—not just the well born or those that have grabbed the reins of power—so that we may govern ourselves more wisely. When I see our army, I see Aaron Swartz and my heart is broken. We have truly lost one of our better angels."

—Carl Malamud

About the Author

Al Sweigart is a software developer and tech book author living in San Francisco. Python is his favorite programming language, and he is the developer of several open source modules for it. His other books are freely available under a Creative Commons license on his website *https://inventwithpython.com/*. His cat weighs 12 pounds.

About the Technical Reviewers

Ari Lacenski creates mobile apps and Python software. She lives in Seattle.

Jean-Philippe Aumasson (Chapters 22–24) is Principal Research Engineer at Kudelski Security, Switzerland. He speaks regularly at information security conferences such as Black Hat, DEF CON, Troopers, and Infiltrate. He is the author of *Serious Cryptography* (No Starch Press, 2017).

BRIEF CONTENTS

Acknowledgments xix
Introduction xxi
Chapter 1: Making Paper Cryptography Tools. 1
Chapter 2: Programming in the Interactive Shell. 11
Chapter 3: Strings and Writing Programs 21
Chapter 4: The Reverse Cipher 39
Chapter 5: The Caesar Cipher. 53
Chapter 6: Hacking the Caesar Cipher with Brute-Force 69
Chapter 7: Encrypting with the Transposition Cipher 77
Chapter 8: Decrypting with the Transposition Cipher 99
Chapter 9: Programming a Program to Test Your Program 113
Chapter 10: Encrypting and Decrypting Files. 127
Chapter 11: Detecting English Programmatically 141
Chapter 12: Hacking the Transposition Cipher 161
Chapter 13: A Modular Arithmetic Module for the Affine Cipher 171
Chapter 14: Programming the Affine Cipher 185
Chapter 15: Hacking the Affine Cipher 197
Chapter 16: Programming the Simple Substitution Cipher 207
Chapter 17: Hacking the Simple Substitution Cipher 221
Chapter 18: Programming the Vigenère Cipher 247
Chapter 19: Frequency Analysis 259
Chapter 20: Hacking the Vigenère Cipher 279

Chapter 21: The One-Time Pad Cipher 315
Chapter 22: Finding and Generating Prime Numbers 321
Chapter 23: Generating Keys for the Public Key Cipher 335
Chapter 24: Programming the Public Key Cipher 349
Appendix: Debugging Python Code 375
Index 381

CONTENTS IN DETAIL

ACKNOWLEDGMENTS **xix**

INTRODUCTION **xxi**

Who Should Read This Book? . . . xxii
What's in This Book? . . . xxiii
How to Use This Book . . . xxiv
Typing Source Code . . . xxiv
Checking for Typos . . . xxv
Coding Conventions in This Book . . . xxv
Online Resources . . . xxv
Downloading and Installing Python . . . xxv
Windows Instructions . . . xxvi
macOS Instructions . . . xxvi
Ubuntu Instructions . . . xxvi
Downloading pyperclip.py . . . xxvi
Starting IDLE . . . xxvii
Summary . . . xxviii

1
MAKING PAPER CRYPTOGRAPHY TOOLS **1**

What Is Cryptography? . . . 2
Codes vs. Ciphers . . . 3
The Caesar Cipher . . . 4
The Cipher Wheel . . . 4
Encrypting with the Cipher Wheel . . . 5
Decrypting with the Cipher Wheel . . . 6
Encrypting and Decrypting with Arithmetic . . . 7
Why Double Encryption Doesn't Work . . . 8
Summary . . . 8
Practice Questions . . . 9

2
PROGRAMMING IN THE INTERACTIVE SHELL **11**

Some Simple Math Expressions . . . 12
Integers and Floating-Point Values . . . 13
Expressions . . . 13
Order of Operations . . . 14
Evaluating Expressions . . . 14
Storing Values with Variables . . . 15
Overwriting Variables . . . 17
Variable Names . . . 18
Summary . . . 18
Practice Questions . . . 19

3
STRINGS AND WRITING PROGRAMS 21

Working with Text Using String Values . . . 22
String Concatenation with the + Operator . . . 23
String Replication with the * Operator . . . 24
Getting Characters from Strings Using Indexes . . . 24
Printing Values with the print() Function . . . 27
Printing Escape Characters . . . 28
Quotes and Double Quotes . . . 29
Writing Programs in IDLE's File Editor . . . 30
Source Code for the "Hello, World!" Program . . . 31
Checking Your Source Code with the Online Diff Tool . . . 31
Using IDLE to Access Your Program Later . . . 32
Saving Your Program . . . 32
Running Your Program . . . 33
Opening the Programs You've Saved . . . 34
How the "Hello, World!" Program Works . . . 34
Comments . . . 34
Printing Directions to the User . . . 34
Taking a User's Input . . . 35
Ending the Program . . . 35
Summary . . . 36
Practice Questions . . . 37

4
THE REVERSE CIPHER 39

Source Code for the Reverse Cipher Program . . . 40
Sample Run of the Reverse Cipher Program . . . 40
Setting Up Comments and Variables . . . 41
Finding the Length of a String . . . 41
Introducing the while Loop . . . 42
The Boolean Data Type . . . 43
Comparison Operators . . . 43
Blocks . . . 45
The while Loop Statement . . . 46
"Growing" a String . . . 47
Improving the Program with an input() Prompt . . . 50
Summary . . . 50
Practice Questions . . . 51

5
THE CAESAR CIPHER 53

Source Code for the Caesar Cipher Program . . . 54
Sample Run of the Caesar Cipher Program . . . 55
Importing Modules and Setting Up Variables . . . 56
Constants and Variables . . . 57
The for Loop Statement . . . 58
An Example for Loop . . . 58
A while Loop Equivalent of a for Loop . . . 59

The if Statement . . . 59
An Example if Statement . . . 60
The else Statement . . . 60
The elif Statement . . . 61
The in and not in Operators . . . 61
The find() String Method . . . 62
Encrypting and Decrypting Symbols . . . 63
Handling Wraparound . . . 64
Handling Symbols Outside of the Symbol Set . . . 65
Displaying and Copying the Translated String . . . 65
Encrypting Other Symbols . . . 66
Summary . . . 66
Practice Questions . . . 67

6
HACKING THE CAESAR CIPHER WITH BRUTE-FORCE 69

Source Code for the Caesar Cipher Hacker Program . . . 70
Sample Run of the Caesar Cipher Hacker Program . . . 71
Setting Up Variables . . . 72
Looping with the range() Function . . . 72
Decrypting the Message . . . 73
Using String Formatting to Display the Key and Decrypted Messages . . . 75
Summary . . . 76
Practice Question . . . 76

7
ENCRYPTING WITH THE TRANSPOSITION CIPHER 77

How the Transposition Cipher Works . . . 78
Encrypting a Message by Hand . . . 79
Creating the Encryption Program . . . 80
Source Code for the Transposition Cipher Encryption Program . . . 81
Sample Run of the Transposition Cipher Encryption Program . . . 82
Creating Your Own Functions with def Statements . . . 82
Defining a Function that Takes Arguments with Parameters . . . 83
Changes to Parameters Exist Only Inside the Function . . . 84
Defining the main() Function . . . 85
Passing the Key and Message As Arguments . . . 86
The List Data Type . . . 86
Reassigning the Items in Lists . . . 87
Lists of Lists . . . 88
Using len() and the in Operator with Lists . . . 89
List Concatenation and Replication with the + and * Operators . . . 89
The Transposition Encryption Algorithm . . . 90
Augmented Assignment Operators . . . 91
Moving currentIndex Through the Message . . . 92
The join() String Method . . . 93
Return Values and return Statements . . . 94
A return Statement Example . . . 94
Returning the Encrypted Ciphertext . . . 95
The __name__ Variable . . . 95
Summary . . . 96
Practice Questions . . . 97

8
DECRYPTING WITH THE TRANSPOSITION CIPHER 99

How to Decrypt with the Transposition Cipher on Paper . . . 100
Source Code for the Transposition Cipher Decryption Program . . . 101
Sample Run of the Transposition Cipher Decryption Program . . . 102
Importing Modules and Setting Up the main() Function . . . 102
Decrypting the Message with the Key . . . 103
The round(), math.ceil(), and math.floor() Functions . . . 103
The decryptMessage() Function . . . 104
Boolean Operators . . . 106
Adjusting the column and row Variables . . . 109
Calling the main() Function . . . 110
Summary . . . 110
Practice Questions . . . 111

9
PROGRAMMING A PROGRAM TO TEST YOUR PROGRAM 113

Source Code for the Transposition Cipher Tester Program . . . 114
Sample Run of the Transposition Cipher Tester Program . . . 115
Importing the Modules . . . 116
Creating Pseudorandom Numbers . . . 116
Creating a Random String . . . 118
Duplicating a String a Random Number of Times . . . 118
List Variables Use References . . . 119
Passing References . . . 121
Using copy.deepcopy() to Duplicate a List . . . 122
The random.shuffle() Function . . . 122
Randomly Scrambling a String . . . 123
Testing Each Message . . . 123
Checking Whether the Cipher Worked and Ending the Program . . . 124
Calling the main() Function . . . 124
Testing the Test Program . . . 125
Summary . . . 125
Practice Questions . . . 126

10
ENCRYPTING AND DECRYPTING FILES 127

Plain Text Files . . . 128
Source Code for the Transposition File Cipher Program . . . 128
Sample Run of the Transposition File Cipher Program . . . 130
Working with Files . . . 130
Opening Files . . . 131
Writing to and Closing Files . . . 131
Reading from a File . . . 132
Setting Up the main() Function . . . 132
Checking Whether a File Exists . . . 133
The os.path.exists() Function . . . 133
Checking Whether the Input File Exists with the os.path.exists() Function . . . 134
Using String Methods to Make User Input More Flexible . . . 134
The upper(), lower(), and title() String Methods . . . 134
The startswith() and endswith() String Methods . . . 135
Using These String Methods in the Program . . . 135

Reading the Input File . . . 136
Measuring the Time It Took to Encrypt or Decrypt . . . 136
The time Module and time.time() Function . . . 136
Using the time.time() Function in the Program . . . 137
Writing the Output File . . . 137
Calling the main() Function . . . 138
Summary . . . 138
Practice Questions . . . 139

11
DETECTING ENGLISH PROGRAMMATICALLY 141

How Can a Computer Understand English? . . . 142
Source Code for the Detect English Module . . . 143
Sample Run of the Detect English Module . . . 145
Instructions and Setting Up Constants . . . 145
The Dictionary Data Type . . . 146
The Difference Between Dictionaries and Lists . . . 147
Adding or Changing Items in a Dictionary . . . 147
Using the len() Function with Dictionaries . . . 148
Using the in Operator with Dictionaries . . . 148
Finding Items Is Faster with Dictionaries than with Lists . . . 149
Using for Loops with Dictionaries . . . 149
Implementing the Dictionary File . . . 150
The split() Method . . . 150
Splitting the Dictionary File into Individual Words . . . 151
Returning the Dictionary Data . . . 151
Counting the Number of English Words in message . . . 152
Divide-by-Zero Errors . . . 152
Counting the English Word Matches . . . 153
The float(), int(), and str() Functions and Integer Division . . . 154
Finding the Ratio of English Words in the Message . . . 154
Removing Non-Letter Characters . . . 155
The append() List Method . . . 155
Creating a String of Letters . . . 156
Detecting English Words . . . 156
Using Default Arguments . . . 157
Calculating Percentages . . . 157
Summary . . . 159
Practice Questions . . . 160

12
HACKING THE TRANSPOSITION CIPHER 161

Source Code of the Transposition Cipher Hacker Program . . . 162
Sample Run of the Transposition Cipher Hacker Program . . . 163
Importing the Modules . . . 164
Multiline Strings with Triple Quotes . . . 164
Displaying the Results of Hacking the Message . . . 165
Getting the Hacked Message . . . 166
The strip() String Method . . . 167
Applying the strip() String Method . . . 168
Failing to Hack the Message . . . 168

Calling the main() Function . . . 169
Summary . . . 169
Practice Questions . . . 169

13
A MODULAR ARITHMETIC MODULE FOR THE AFFINE CIPHER 171

Modular Arithmetic . . . 172
The Modulo Operator . . . 173
Finding Factors to Calculate the Greatest Common Divisor . . . 173
Multiple Assignment . . . 175
Euclid's Algorithm for Finding the GCD . . . 176
Understanding How the Multiplicative and Affine Ciphers Work . . . 177
    Choosing Valid Multiplicative Keys . . . 178
    Encrypting with the Affine Cipher . . . 179
    Decrypting with the Affine Cipher . . . 179
    Finding Modular Inverses . . . 181
    The Integer Division Operator . . . 181
Source Code for the Cryptomath Module . . . 182
Summary . . . 183
Practice Questions . . . 183

14
PROGRAMMING THE AFFINE CIPHER 185

Source Code for the Affine Cipher Program . . . 186
Sample Run of the Affine Cipher Program . . . 188
Setting Up Modules, Constants, and the main() Function . . . 188
Calculating and Validating the Keys . . . 189
    The Tuple Data Type . . . 190
    Checking for Weak Keys . . . 190
    How Many Keys Can the Affine Cipher Have? . . . 191
Writing the Encryption Function . . . 193
Writing the Decryption Function . . . 194
Generating Random Keys . . . 195
Calling the main() Function . . . 196
Summary . . . 196
Practice Questions . . . 196

15
HACKING THE AFFINE CIPHER 197

Source Code for the Affine Cipher Hacker Program . . . 198
Sample Run of the Affine Cipher Hacker Program . . . 199
Setting Up Modules, Constants, and the main() Function . . . 200
The Affine Cipher Hacking Function . . . 201
    The Exponent Operator . . . 201
    Calculating the Total Number of Possible Keys . . . 201
    The continue Statement . . . 202
    Using continue to Skip Code . . . 203
Calling the main() Function . . . 204
Summary . . . 205
Practice Questions . . . 205

16
PROGRAMMING THE SIMPLE SUBSTITUTION CIPHER 207

How the Simple Substitution Cipher Works . . . 208
Source Code for the Simple Substitution Cipher Program . . . 209
Sample Run of the Simple Substitution Cipher Program . . . 210
Setting Up Modules, Constants, and the main() Function . . . 211
The sort() List Method . . . 212
Wrapper Functions . . . 213
The translateMessage() Function . . . 215
    The isupper() and islower() String Methods . . . 216
    Preserving Cases with isupper() . . . 217
Generating a Random Key . . . 218
Calling the main() Function . . . 219
Summary . . . 219
Practice Questions . . . 219

17
HACKING THE SIMPLE SUBSTITUTION CIPHER 221

Using Word Patterns to Decrypt . . . 222
    Finding Word Patterns . . . 222
    Finding Potential Decryption Letters . . . 223
Overview of the Hacking Process . . . 225
The Word Pattern Modules . . . 225
Source Code for the Simple Substitution Hacking Program . . . 226
Sample Run of the Simple Substitution Hacking Program . . . 229
Setting Up Modules and Constants . . . 230
Finding Characters with Regular Expressions . . . 230
Setting Up the main() Function . . . 231
Displaying Hacking Results to the User . . . 232
Creating a Cipherletter Mapping . . . 232
    Creating a Blank Mapping . . . 232
    Adding Letters to a Mapping . . . 233
    Intersecting Two Mappings . . . 234
    How the Letter-Mapping Helper Functions Work . . . 235
    Identifying Solved Letters in Mappings . . . 238
    Testing the removeSolvedLetterFromMapping() Function . . . 240
The hackSimpleSub() Function . . . 241
    The replace() String Method . . . 243
    Decrypting the Message . . . 243
    Decrypting in the Interactive Shell . . . 244
Calling the main() Function . . . 245
Summary . . . 246
Practice Questions . . . 246

18
PROGRAMMING THE VIGENÈRE CIPHER 247

Using Multiple Letter Keys in the Vigenère Cipher . . . 248
    Longer Vigenère Keys Are More Secure . . . 249
    Choosing a Key That Prevents Dictionary Attacks . . . 250
Source Code for the Vigenère Cipher Program . . . 251
Sample Run of the Vigenère Cipher Program . . . 252

Setting Up Modules, Constants, and the main() Function 252
Building Strings with the List-Append-Join Process 253
Encrypting and Decrypting the Message 255
Calling the main() Function 257
Summary 257
Practice Questions 258

19
FREQUENCY ANALYSIS **259**

Analyzing the Frequency of Letters in Text 260
Matching Letter Frequencies 262
    Calculating the Frequency Match Score for the Simple Substitution Cipher . . . 262
    Calculating the Frequency Match Score for the Transposition Cipher 263
    Using Frequency Analysis on the Vigenère Cipher 264
Source Code for Matching Letter Frequencies 265
Storing the Letters in ETAOIN Order 266
Counting the Letters in a Message 267
Getting the First Member of a Tuple 268
Ordering the Letters in the Message by Frequency 268
    Counting the Letters with getLetterCount() 269
    Creating a Dictionary of Frequency Counts and Letter Lists 269
    Sorting the Letter Lists in Reverse ETAOIN Order 270
    Sorting the Dictionary Lists by Frequency 274
    Creating a List of the Sorted Letters 276
Calculating the Frequency Match Score of the Message 276
Summary 277
Practice Questions 278

20
HACKING THE VIGENÈRE CIPHER **279**

Using a Dictionary Attack to Brute-Force the Vigenère Cipher 280
Source Code for the Vigenère Dictionary Hacker Program 280
Sample Run of the Vigenère Dictionary Hacker Program 281
About the Vigenère Dictionary Hacker Program 281
Using Kasiski Examination to Find the Key's Length 282
    Finding Repeated Sequences 282
    Getting Factors of Spacings 283
    Getting Every Nth Letters from a String 284
    Using Frequency Analysis to Break Each Subkey 285
    Brute-Forcing Through the Possible Keys 287
Source Code for the Vigenère Hacking Program 287
Sample Run of the Vigenère Hacking Program 293
Importing Modules and Setting Up the main() Function 294
Finding Repeated Sequences 294
Calculating the Factors of the Spacings 297
    Removing Duplicates with the set() Function 298
    Removing Duplicate Factors and Sorting the List 298
    Finding the Most Common Factors 298
Finding the Most Likely Key Lengths 300
    The extend() List Method 301
    Extending the repeatedSeqSpacings Dictionary 301
    Getting the Factors from factorsByCount 302

Getting Letters Encrypted with the Same Subkey . . . 302
Attempting Decryption with a Likely Key Length . . . 303
The end Keyword Argument for print() . . . 306
Running the Program in Silent Mode or Printing Information to the User . . . 306
Finding Possible Combinations of Subkeys . . . 306
Printing the Decrypted Text with the Correct Casing . . . 310
Returning the Hacked Message . . . 311
Breaking Out of the Loop When a Potential Key Is Found . . . 311
Brute-Forcing All Other Key Lengths . . . 312
Calling the main() Function . . . 313
Modifying the Constants of the Hacking Program . . . 313
Summary . . . 314
Practice Questions . . . 314

21
THE ONE-TIME PAD CIPHER 315

The Unbreakable One-Time Pad Cipher . . . 316
Making Key Length Equal Message Length . . . 316
Making the Key Truly Random . . . 318
Avoiding the Two-Time Pad . . . 319
Why the Two-Time Pad Is the Vigenère Cipher . . . 319
Summary . . . 320
Practice Questions . . . 320

22
FINDING AND GENERATING PRIME NUMBERS 321

What Is a Prime Number? . . . 322
Source Code for the Prime Numbers Module . . . 324
Sample Run of the Prime Numbers Module . . . 326
How the Trial Division Algorithm Works . . . 326
Implementing the Trial Division Algorithm Test . . . 328
The Sieve of Eratosthenes . . . 328
Generating Prime Numbers with the Sieve of Eratosthenes . . . 330
The Rabin-Miller Primality Algorithm . . . 331
Finding Large Prime Numbers . . . 332
Generating Large Prime Numbers . . . 333
Summary . . . 334
Practice Questions . . . 334

23
GENERATING KEYS FOR THE PUBLIC KEY CIPHER 335

Public Key Cryptography . . . 336
The Problem with Authentication . . . 337
Digital Signatures . . . 338
Beware the MITM Attack . . . 339
Steps for Generating Public and Private Keys . . . 340
Source Code for the Public Key Generation Program . . . 340
Sample Run of the Public Key Generation Program . . . 342
Creating the main() Function . . . 343

Generating Keys with the generateKey() Function . . . 343
Calculating an e Value . . . 344
Calculating a d Value . . . 344
Returning the Keys . . . 345
Creating Key Files with the makeKeyFiles() Function . . . 345
Calling the main() Function . . . 347
Hybrid Cryptosystems . . . 347
Summary . . . 348
Practice Questions . . . 348

24
PROGRAMMING THE PUBLIC KEY CIPHER 349

How the Public Key Cipher Works . . . 350
Creating Blocks . . . 350
Converting a String into a Block . . . 350
The Mathematics of Public Key Cipher Encryption and Decryption . . . 353
Converting a Block to a String . . . 354
Why We Can't Hack the Public Key Cipher . . . 355
Source Code for the Public Key Cipher Program . . . 357
Sample Run of the Public Key Cipher Program . . . 360
Setting Up the Program . . . 362
How the Program Determines Whether to Encrypt or Decrypt . . . 362
Converting Strings to Blocks with getBlocksFromText() . . . 363
The min() and max() Functions . . . 364
Storing Blocks in blockInt . . . 364
Using getTextFromBlocks() to Decrypt . . . 366
Using the insert() List Method . . . 367
Merging the Message List into One String . . . 367
Writing the encryptMessage() Function . . . 367
Writing the decryptMessage() Function . . . 368
Reading in the Public and Private Keys from Their Key Files . . . 369
Writing the Encryption to a File . . . 369
Decrypting from a File . . . 371
Calling the main() Function . . . 373
Summary . . . 373

APPENDIX
DEBUGGING PYTHON CODE 375

How the Debugger Works . . . 375
Debugging the Reverse Cipher Program . . . 377
Setting Breakpoints . . . 379
Summary . . . 380

INDEX 381

ACKNOWLEDGMENTS

This book would not have been possible without the exceptional work of the No Starch Press team. Thanks to my publisher, Bill Pollock; thanks to my editors, Riley Hoffman, Jan Cash, Annie Choi, Anne Marie Walker, and Laurel Chun, for their incredible help throughout the process; thanks to my technical editor, Ari Lacenski, for her help in this edition and back when it was just a stack of printouts I showed her at Shotwell's; thanks to JP Aumasson for lending his expertise in the public key chapters; and thanks to Josh Ellingson for a great cover.

INTRODUCTION

"I couldn't help but overhear,
probably because I was eavesdropping."
—Anonymous

If you could travel back to the early 1990s with this book, the contents of Chapter 23 that implement part of the RSA cipher would be illegal to export out of the United States. Because messages encrypted with RSA are impossible to hack, the export of encryption software like RSA was deemed a matter of national security and required State Department approval. In fact, strong cryptography was regulated at the same level as tanks, missiles, and flamethrowers.

In 1990, Daniel J. Bernstein, a student at the University of California, Berkeley, wanted to publish an academic paper that featured source code of his Snuffle encryption system. The US government informed him that he would need to become a licensed arms dealer before he could post his source code on the internet. The government also told him that it would deny him an export license if he applied for one because his technology was too secure.

The Electronic Frontier Foundation, a young digital civil liberties organization, represented Bernstein in *Bernstein v. United States*. For the first time ever, the courts ruled that written software code was speech protected by the First Amendment and that the export control laws on encryption violated Bernstein's First Amendment rights.

Now, strong cryptography is at the foundation of a large part of the global economy, safeguarding businesses and e-commerce sites used by millions of internet shoppers every day. The intelligence community's predictions that encryption software would become a grave national security threat were unfounded.

But as recently as the 1990s, spreading this knowledge freely (as this book does) would have landed you in prison for arms trafficking. For a more detailed history of the legal battle for freedom of cryptography, read Steven Levy's book *Crypto: How the Code Rebels Beat the Government, Saving Privacy in the Digital Age* (Penguin, 2001).

Who Should Read This Book?

Many books teach beginners how to write secret messages using ciphers. A couple of books teach beginners how to hack ciphers. But no books teach beginners how to program computers to hack ciphers. This book fills that gap.

This book is for those who are curious about encryption, hacking, or cryptography. The ciphers in this book (except for the public key cipher in Chapters 23 and 24) are all centuries old, but any laptop has the computational power to hack them. No modern organizations or individuals use these ciphers anymore, but by learning them, you'll learn the foundations cryptography was built on and how hackers can break weak encryption.

NOTE *The ciphers you'll learn in this book are fun to play with, but they don't provide true security. Don't use any of the encryption programs in this book to secure your actual files. As a general rule, you shouldn't trust the ciphers that you create. Real-world ciphers are subject to years of analysis by professional cryptographers before being put into use.*

This book is also for people who have never programmed before. It teaches basic programming concepts using the Python programming language, which is one of the best languages for beginners. It has a gentle learning curve that novices of all ages can master, yet it's also a powerful language used by professional software developers. Python runs on Windows, macOS, Linux, and even the Raspberry Pi, and it's free to download and use. (See "Downloading and Installing Python" on page xxv for instructions.)

In this book, I'll use the term *hacker* often. The word has two definitions. A hacker can be a person who studies a system (such as the rules of a cipher or a piece of software) to understand it so well that they're not limited by that system's original rules and can modify it in creative ways.

A hacker can also be a criminal who breaks into computer systems, violates people's privacy, and causes damage. This book uses the term in the first sense. Hackers are cool. Criminals are just people who think they're being clever by breaking stuff.

What's in This Book?

The first few chapters introduce basic Python and cryptography concepts. Thereafter, chapters generally alternate between explaining a program for a cipher and then explaining a program that hacks that cipher. Each chapter also includes practice questions to help you review what you've learned.

- **Chapter 1: Making Paper Cryptography Tools** covers some simple paper tools, showing how encryption was done before computers.
- **Chapter 2: Programming in the Interactive Shell** explains how to use Python's interactive shell to play around with code one line at a time.
- **Chapter 3: Strings and Writing Programs** covers writing full programs and introduces the string data type used in all programs in this book.
- **Chapter 4: The Reverse Cipher** explains how to write a simple program for your first cipher.
- **Chapter 5: The Caesar Cipher** covers a basic cipher first invented thousands of years ago.
- **Chapter 6: Hacking the Caesar Cipher with Brute-Force** explains the brute-force hacking technique and how to use it to decrypt messages without the encryption key.
- **Chapter 7: Encrypting with the Transposition Cipher** introduces the transposition cipher and a program that encrypts messages with it.
- **Chapter 8: Decrypting with the Transposition Cipher** covers the second half of the transposition cipher: being able to decrypt messages with a key.
- **Chapter 9: Programming a Program to Test Your Program** introduces the programming technique of testing programs with other programs.
- **Chapter 10: Encrypting and Decrypting Files** explains how to write programs that read files from and write files to the hard drive.
- **Chapter 11: Detecting English Programmatically** describes how to make the computer detect English sentences.
- **Chapter 12: Hacking the Transposition Cipher** combines the concepts from previous chapters to hack the transposition cipher.
- **Chapter 13: A Modular Arithmetic Module for the Affine Cipher** explains the math concepts behind the affine cipher.
- **Chapter 14: Programming the Affine Cipher** covers writing an affine cipher encryption program.
- **Chapter 15: Hacking the Affine Cipher** explains how to write a program to hack the affine cipher.

- **Chapter 16: Programming the Simple Substitution Cipher** covers writing a simple substitution cipher encryption program.
- **Chapter 17: Hacking the Simple Substitution Cipher** explains how to write a program to hack the simple substitution cipher.
- **Chapter 18: Programming the Vigenère Cipher** explains a program for the Vigenère cipher, a more complex substitution cipher.
- **Chapter 19: Frequency Analysis** explores the structure of English words and how to use it to hack the Vigenère cipher.
- **Chapter 20: Hacking the Vigenère Cipher** covers a program for hacking the Vigenère cipher.
- **Chapter 21: The One-Time Pad Cipher** explains the one-time pad cipher and why it's mathematically impossible to hack.
- **Chapter 22: Finding and Generating Prime Numbers** covers how to write a program that quickly determines whether a number is prime.
- **Chapter 23: Generating Keys for the Public Key Cipher** describes public key cryptography and how to write a program that generates public and private keys.
- **Chapter 24: Programming the Public Key Cipher** explains how to write a program for a public key cipher, which you can't hack using a mere laptop.
- The appendix, **Debugging Python Code**, shows you how to use IDLE's debugger to find and fix bugs in your programs.

How to Use This Book

Cracking Codes with Python is different from other programming books because it focuses on the source code of complete programs. Instead of teaching you programming concepts and leaving it up to you to figure out how to make your own programs, this book shows you complete programs and explains how they work.

In general, you should read the chapters in this book in order. The programming concepts build on those in the previous chapters. However, Python is such a readable language that after the first few chapters, you can probably jump ahead to later chapters and piece together what the code does. If you jump ahead and feel lost, return to earlier chapters.

Typing Source Code

As you read through this book, I encourage you to *manually type the source code from this book into Python*. Doing so will definitely help you understand the code better.

When typing the source code, don't include the line numbers that appear at the beginning of each line. These numbers are not part of the actual programs, and we use them only to refer to specific lines in the code. But aside from the line numbers, be sure to enter the code exactly as it appears, including the uppercase and lowercase letters.

You'll also notice that some of the lines don't begin at the leftmost edge of the page but are indented by four, eight, or more spaces. Be sure to enter the correct number of spaces at the beginning of each line to avoid errors.

But if you would rather not type the code, you can download the source code files from this book's website at *https://www.nostarch.com/crackingcodes/*.

Checking for Typos

Although manually entering the source code for the programs is helpful for learning Python, you might occasionally make typos that cause errors. These typos can be difficult to spot, especially when your source code is very long.

To quickly and easily check for mistakes in your typed source code, you can copy and paste the text into the online diff tool on the book's website at *https://www.nostarch.com/crackingcodes/*. The diff tool shows any differences between the source code in the book and yours.

Coding Conventions in This Book

This book is not designed to be a reference manual; it's a hands-on guide for beginners. For this reason, the coding style sometimes goes against best practices, but that's a conscious decision to make the code easier to learn. This book also skips theoretical computer science concepts.

Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort.

Online Resources

This book's website (*https://www.nostarch.com/crackingcodes/*) includes many useful resources, including downloadable files of the programs and sample solutions to the practice questions. This book covers classical ciphers thoroughly, but because there is always more to learn, I've also included suggestions for further reading on many of the topics introduced in this book.

Downloading and Installing Python

Before you can begin programming, you'll need to install the *Python interpreter*, which is software that executes the instructions you'll write in the Python language. I'll refer to "the Python interpreter" as "Python" from now on.

Download Python for Windows, macOS, and Ubuntu for free from *https://www.python.org/downloads/*. If you download the latest version, all of the programs in this book should work.

NOTE *Be sure to download a version of Python 3 (such as 3.6). The programs in this book are written to run on Python 3 and may not run correctly, if at all, on Python 2.*

Windows Instructions

On Windows, download the Python installer, which should have a filename ending with *.msi,* and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. Select **Install Now** to begin the installation.
2. When the installation is finished, click **Close**.

macOS Instructions

On macOS, download the *.dmg* file for your version of macOS from the website and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter your computer's administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.
3. Select **HD Macintosh** (or the name of your hard drive) and click **Install**.

Ubuntu Instructions

If you're running Ubuntu, install Python from the Ubuntu Software Center by following these steps:

1. Open the Ubuntu Software Center.
2. Type `Python` in the search box in the top-right corner of the window.
3. Select **IDLE (using Python 3.6)**, or whatever is the latest version.
4. Click **Install**.

You may have to enter the administrator password to complete the installation.

Downloading pyperclip.py

Almost every program in this book uses a custom module I wrote called *pyperclip.py*. This module provides functions that let your programs copy and paste text to the clipboard. It doesn't come with Python, so you'll need to download it from *https://www.nostarch.com/crackingcodes/*.

This file must be in the same folder (also called *directory*) as the Python program files you write. Otherwise you'll see the following error message when you try to run your programs:

```
ImportError: No module named pyperclip
```

Now that you've downloaded and installed the Python interpreter and the *pyperclip.py* module, let's look at where you'll be writing your programs.

Starting IDLE

While the Python interpreter is the software that runs your Python programs, the *interactive development environment (IDLE)* software is where you'll write your programs, much like a word processor. IDLE is installed when you install Python. To start IDLE, follow these steps:

- On Windows 7 or newer, click the Start icon in the lower-left corner of your screen, enter **IDLE** in the search box, and select **IDLE (Python 3.6 64-bit)**.
- On macOS, open Finder, click **Applications**, click **Python 3.6**, and then click the **IDLE** icon.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter `idle3`. (You may also be able to click **Applications** at the top of the screen, select **Programming**, and then click **IDLE 3**.)

No matter which operating system you're running, the IDLE window should look something like Figure 1. The header text may be slightly different depending on your specific version of Python.

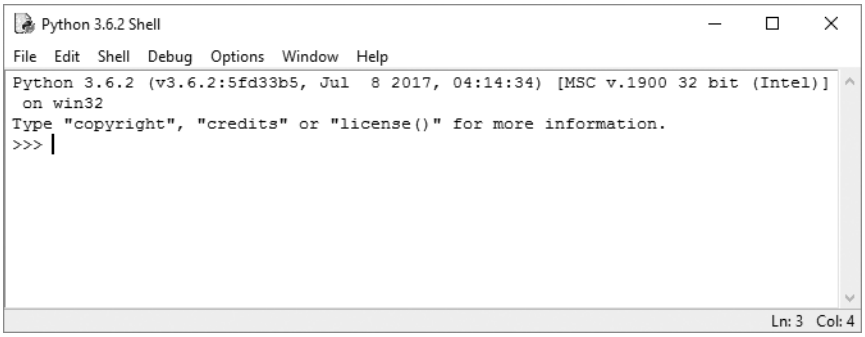


Figure 1: The IDLE window

This window is called the *interactive shell*. A shell is a program that lets you type instructions into the computer, much like the Terminal on macOS or the Windows Command Prompt. Sometimes you'll want to run short snippets of code instead of writing a full program. Python's interactive shell lets you enter instructions for the Python interpreter software, which the computer reads and runs immediately.

For example, type the following into the interactive shell next to the >>> prompt:

```
>>> print('Hello, world!')
```

Press ENTER, and the interactive shell should display this in response:

```
Hello, world!
```

Summary

Before the introduction of computers ushered in modern cryptography, breaking many codes was impossible using just pencil and paper. Although computing made many of the old, classical ciphers vulnerable to attack, they're still fun to learn about. Writing cryptanalysis programs that crack these ciphers is a great way to learn how to program.

In Chapter 1, we'll start with some basic cryptography tools to encrypt and decrypt messages without the aid of computers.

Let's get hacking.

1

MAKING PAPER CRYPTOGRAPHY TOOLS

"The encryption genie is out of the bottle."
—Jan Koum, WhatsApp founder

Before we start writing cipher programs, let's look at the process of encrypting and decrypting with just pencil and paper. This will help you understand how ciphers work and the math that goes into producing their secret messages. In this chapter, you'll learn what we mean by cryptography and how codes are different from ciphers. Then you'll use a simple cipher called the Caesar cipher to encrypt and decrypt messages using paper and pencil.

TOPICS COVERED IN THIS CHAPTER

- What is cryptography?
- Codes and ciphers
- The Caesar cipher
- Cipher wheels
- Doing cryptography with arithmetic
- Double encryption

What Is Cryptography?

Historically, anyone who has needed to share secrets with others, such as spies, soldiers, hackers, pirates, merchants, tyrants, and political activists, has relied on cryptography to make sure their secrets stay secret. *Cryptography* is the science of using secret codes. To understand what cryptography looks like, look at the following two pieces of text:

nyr N.vNwz5uNz5Ns6620Nz0N3z2v
N yvNwz9vNz5N6!9Nyvr9
y0QNnvNwv tyNz
Nw964N6!9N5vzxys690,N.vN2z5u-
3vNz Nr Ny64v,N.vNt644!5ztr vNz
N 6N6 yv90,Nr5uNz Nsvt64v0N
yvN7967v9 BN6wNr33Q N-m63 rz9v

!NN2 Nuwv,N9,vNN!vNrBN3zyN4vN
N6 Qvv0z6nvN.7N0yv4N 4 zzvNN
vyN,NN99z0zz6wz0y3vv26 9
w296vyNNrrNyQst.560N94Nu5y
rN5nz5vv5t6v63zNr5.
N75sz6966NNvw6 zu0 wtNxs6t
49NrN3Ny9Nvzy!

The text on the left is a secret message that has been *encrypted*, or turned into a secret code. It's completely unreadable to anyone who doesn't know how to *decrypt* it, or turn it back into the original English message. The message on the right is random gibberish with no hidden meaning. Encryption keeps a message secret from other people who can't decipher it, even if they get their hands on the encrypted message. *An encrypted message looks exactly like random nonsense.*

A *cryptographer* uses and studies secret codes. Of course, these secret messages don't always remain secret. A *cryptanalyst*, also called a *code breaker* or *hacker*, can hack secret codes and read other people's encrypted messages. This book teaches you how to encrypt and decrypt messages using various techniques. But unfortunately (or fortunately), the type of hacking you'll learn in this book isn't dangerous enough to get you in trouble with the law.

Codes vs. Ciphers

Unlike ciphers, *codes* are made to be understandable and publicly available. Codes substitute messages with symbols that anyone should be able to look up to translate into a message.

In the early 19th century, one well-known code came from the development of the electric telegraph, which allowed for near-instant communication across continents through wires. Sending messages by telegraph was much faster than the previous alternative of sending a horseback rider carrying a bag of letters. However, the telegraph couldn't directly send written letters drawn on paper. Instead, it could send only two types of electric pulses: a short pulse called a "dot" and a long pulse called a "dash."

To convert letters of the alphabet into these dots and dashes, you need an encoding system to translate English to electric pulses. The process of converting English into dots and dashes to send over a telegraph is called *encoding*, and the process of translating electric pulses to English when a message is received is called *decoding*. The code used to encode and decode messages over telegraphs (and later, radio) was called *Morse code*, as shown in Table 1-1. Morse code was developed by Samuel Morse and Alfred Vail.

Table 1-1: International Morse Code Encoding

Letter	Encoding	Letter	Encoding	Number	Encoding
A	•—	N	—•	1	•————
B	—•••	O	———	2	••———
C	—•—•	P	•——•	3	•••——
D	—••	Q	——•—	4	••••—
E	•	R	•—•	5	•••••
F	••—•	S	•••	6	—••••
G	——•	T	—	7	——•••
H	••••	U	••—	8	———••
I	••	V	•••—	9	————•
J	•———	W	•——	0	—————
K	—•—	X	—••—		
L	•—••	Y	—•——		
M	——	Z	——••		

By tapping dots and dashes with a one-button telegraph, a telegraph operator could communicate an English message to someone on the other side of the world almost instantly! (To learn more about Morse code, visit *https://www.nostarch.com/crackingcodes/*.)

In contrast with codes, a *cipher* is a specific type of code meant to keep messages secret. You can use a cipher to turn understandable English text, called *plaintext*, into gibberish that hides a secret message, called the *ciphertext*. A cipher is a set of rules for converting between plaintext and ciphertext. These rules often use a secret key to encrypt or decrypt that only the communicators know. In this book, you'll learn several ciphers and write programs to use these ciphers to encrypt and decrypt text. But first, let's encrypt messages by hand using simple paper tools.

The Caesar Cipher

The first cipher you'll learn is the Caesar cipher, which is named after Julius Caesar who used it 2000 years ago. The good news is that it's simple and easy to learn. The bad news is that because it's so simple, it's also easy for a cryptanalyst to break. However, it's still a useful learning exercise.

The Caesar cipher works by substituting each letter of a message with a new letter after shifting the alphabet over. For example, Julius Caesar substituted letters in his messages by shifting the letters in the alphabet down by three, and then replacing every letter with the letters in his shifted alphabet.

For example, every A in the message would be replaced by a D, every B would be an E, and so on. When Caesar needed to shift letters at the end of the alphabet, such as Y, he would wrap around to the beginning of the alphabet and shift three places to B. In this section, we'll encrypt a message by hand using the Caesar cipher.

The Cipher Wheel

To make converting plaintext to ciphertext using the Caesar cipher easier, we'll use a *cipher wheel*, also called a *cipher disk*. The cipher wheel consists of two rings of letters; each ring is split up into 26 slots (for a 26-letter alphabet). The outer ring represents the plaintext alphabet, and the inner ring represents the corresponding letters in the ciphertext. The inner ring also numbers the letters from 0 to 25. These numbers represent the *encryption key*, which in this case is the number of letters required to shift from A to the corresponding letter on the inner ring. Because the shift is circular, shifting with a key greater than 25 makes the alphabets wrap around, so shifting by 26 would be the same as shifting by 0, shifting by 27 would be the same as shifting by 1, and so on.

You can access a virtual cipher wheel online at *https://www.nostarch.com/crackingcodes/*. Figure 1-1 shows what it looks like. To spin the wheel, click it and then move the mouse cursor around until the configuration you want is in place. Then click the mouse again to stop the wheel from spinning.

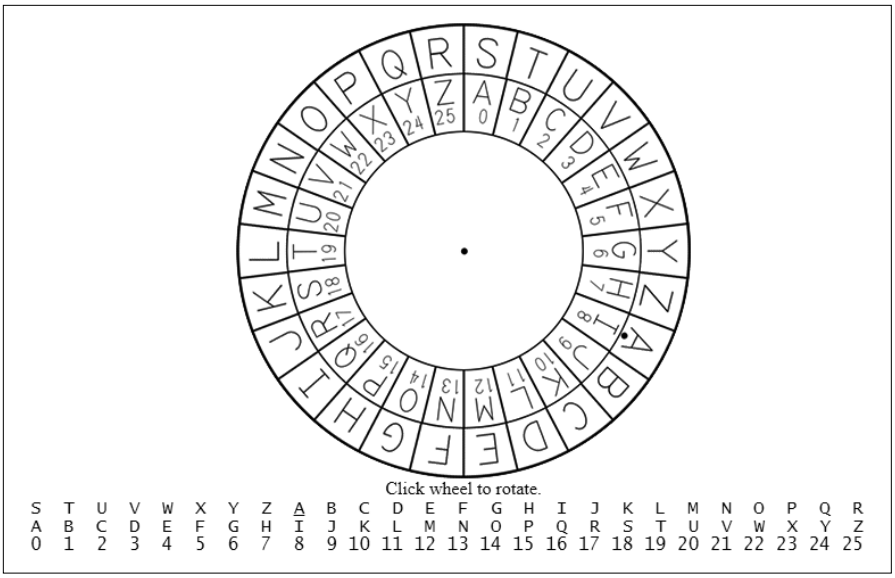


Figure 1-1: The online cipher wheel

A printable paper cipher wheel is also available from the book's website. Cut out the two circles and lay them on top of each other, placing the smaller one in the middle of the larger one. Insert a pin or brad through the center of both circles so you can spin them around in place.

Using either the paper or the virtual wheel, you can encrypt secret messages by hand.

Encrypting with the Cipher Wheel

To begin encrypting, write your message in English on a piece of paper. For this example, we'll encrypt the message THE SECRET PASSWORD IS ROSEBUD. Next, spin the inner wheel of the cipher wheel until its slots match up with slots in the outer wheel. Notice the dot next to the letter A in the outer wheel. Take note of the number in the inner wheel next to this dot. This is the encryption key.

For example, in Figure 1-1, the outer circle's A is over the inner circle's number 8. We'll use this encryption key to encrypt the message in our example, as shown in Figure 1-2.

T	H	E		S	E	C	R	E	T		P	A	S	S	W	O	R	D		I	S		R	O	S	E	B	U	D
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
B	P	M		A	M	K	Z	M	B		X	I	A	A	E	W	Z	L		Q	A		Z	W	A	M	J	C	L

Figure 1-2: Encrypting a message with a Caesar cipher key of 8

For each letter in the message, find it in the outer circle and replace it with the corresponding letter in the inner circle. In this example, the first letter in the message is T (the first T in "THE SECRET…"), so find the

letter T in the outer circle and then find the corresponding letter in the inner circle, which is the letter B. So the secret message always replaces a T with a B. (If you were using a different encryption key, each T in the plaintext would be replaced with a different letter.) The next letter in the message is H, which turns into P. The letter E turns into M. Each letter on the outer wheel always encrypts to the same letter on the inner wheel. To save time, after you look up the first T in "THE SECRET…" and see that it encrypts to B, you can replace every T in the message with B, so you only need to look up a letter once.

After you encrypt the entire message, the original message, THE SECRET PASSWORD IS ROSEBUD, becomes BPM AMKZMB XIAAEWZL QA ZWAMJCL. Notice that non-letter characters, such as the spaces, are not changed.

Now you can send this encrypted message to someone (or keep it for yourself), and nobody will be able to read it unless you tell them the secret encryption key. Be sure to keep the encryption key a secret; the ciphertext can be read by anyone who knows that the message was encrypted with key 8.

Decrypting with the Cipher Wheel

To decrypt a ciphertext, start from the inner circle of the cipher wheel and then move to the outer circle. For example, let's say you receive the ciphertext IWT CTL EPHHLDGS XH HLDGSUXHW. You wouldn't be able to decrypt the message unless you knew the key (or unless you were a clever hacker). Luckily, your friend has already told you that they use the key 15 for their messages. The cipher wheel for this key is shown in Figure 1-3.

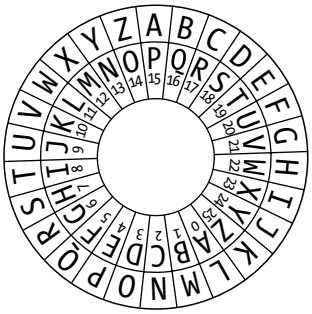


Figure 1-3: A cipher wheel set to key 15

Now you can line up the letter A on the outer circle (the one with the dot below it) over the letter on the inner circle that has the number 15 (which is the letter P). Then, find the first letter in the secret message on the inner circle, which is I, and look at the corresponding letter on the outer circle, which is T. The second letter in the ciphertext, W, decrypts to the letter H. Decrypt the rest of the letters in the ciphertext back to the plaintext, and you'll get the message THE NEW PASSWORD IS SWORDFISH, as shown in Figure 1-4.

I	W	T		C	T	L		E	P	H	H	L	D	G	S		X	H		H	L	D	G	S	U	X	H	W
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
T	H	E		N	E	W		P	A	S	S	W	O	R	D		I	S		S	W	O	R	D	F	I	S	H

Figure 1-4: Decrypting a message with a Caesar cipher key of 15

If you used an incorrect key, like 16, the decrypted message would be SGD MDV OZRRVNQC HR RVNQCEHRG, which is unreadable. Unless the correct key is used, the decrypted message won't be understandable.

Encrypting and Decrypting with Arithmetic

The cipher wheel is a convenient tool for encrypting and decrypting with the Caesar cipher, but you can also encrypt and decrypt using arithmetic. To do so, write the letters of the alphabet from A to Z with the numbers from 0 to 25 under each letter. Begin with 0 under the A, 1 under the B, and so on until 25 is under the Z. Figure 1-5 shows what it should look like.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figure 1-5: Numbering the alphabet from 0 to 25

You can use this letters-to-numbers code to represent letters. This is a powerful concept, because it allows you to do math on letters. For example, if you represent the letters CAT as the numbers 2, 0, and 19, you can add 3 to get the numbers 5, 3, and 22. These new numbers represent the letters FDW, as shown in Figure 1-5. You have just "added" 3 to the word *cat*! Later, we'll be able to program a computer to do this math for us.

To use arithmetic to encrypt with the Caesar cipher, find the number under the letter you want to encrypt and add the key number to it. The resulting sum is the number under the encrypted letter. For example, let's encrypt HELLO. HOW ARE YOU? using the key 13. (You can use any number from 1 to 25 for the key.) First, find the number under H, which is 7. Then add 13 to this number: 7 + 13 = 20. Because the number 20 is under the letter U, the letter H encrypts to U.

Similarly, to encrypt the letter E (4), add 4 + 13 = 17. The number above 17 is R, so E gets encrypted to R, and so on.

This process works fine until the letter O. The number under O is 14. But 14 plus 13 is 27, and the list of numbers only goes up to 25. If the sum of the letter's number and the key is 26 or more, you need to subtract 26 from it. In this case, 27 – 26 = 1. The letter above the number 1 is B, so O encrypts to B using the key 13. When you encrypt each letter in the message, the ciphertext will be URYYB. UBJ NER LBH?

To decrypt the ciphertext, subtract the key instead of adding it. The number of the ciphertext letter B is 1. Subtract 13 from 1 to get –12. Like our "subtract 26" rule for encrypting, when the result is less than 0 when decrypting, we need to add 26. Because –12 + 26 = 14, the ciphertext letter B decrypts to O.

NOTE *If you don't know how to add and subtract with negative numbers, you can read about it at* https://www.nostarch.com/crackingcodes/.

As you can see, you don't need a cipher wheel to use the Caesar cipher. All you need is a pencil, a piece of paper, and some simple arithmetic!

Why Double Encryption Doesn't Work

You might think encrypting a message twice using two different keys would double the strength of the encryption. But this isn't the case with the Caesar cipher (and most other ciphers). In fact, the result of double encryption is the same as what you would get after one normal encryption. Let's try double encrypting a message to see why.

For example, if you encrypt the word KITTEN using the key 3, you're adding 3 to the plaintext letter's number, and the resulting ciphertext would be NLWWHQ. If you then encrypt NLWWHQ, this time using the key 4, the resulting ciphertext would be RPAALU because you're adding 4 to the plaintext letter's number. But this is the same as encrypting the word KITTEN once with a key of 7.

For most ciphers, encrypting more than once doesn't provide additional strength. In fact, if you encrypt some plaintext with two keys that add up to 26, the resulting ciphertext will be the same as the original plaintext!

Summary

The Caesar cipher and other ciphers like it were used to encrypt secret information for several centuries. But if you wanted to encrypt a long message—say, an entire book—it could take days or weeks to encrypt it all by hand. This is where programming can help. A computer can encrypt and decrypt a large amount of text in less than a second!

To use a computer for encryption, you need to learn how to *program*, or instruct, the computer to do the same steps we just did using a language the computer can understand. Fortunately, learning a programming language like Python isn't nearly as difficult as learning a foreign language like Japanese or Spanish. You also don't need to know much math besides addition, subtraction, and multiplication. All you need is a computer and this book!

Let's move on to Chapter 2, where we'll learn how to use Python's interactive shell to explore code one line at a time.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Encrypt the following entries from Ambrose Bierce's *The Devil's Dictionary* with the given keys:
   a. With key 4: "AMBIDEXTROUS: Able to pick with equal skill a right-hand pocket or a left."
   b. With key 17: "GUILLOTINE: A machine which makes a Frenchman shrug his shoulders with good reason."
   c. With key 21: "IMPIETY: Your irreverence toward my deity."
2. Decrypt the following ciphertexts with the given keys:
   a. With key 15: "ZXAI: P RDHIJBT HDBTIXBTH LDGC QN HRDIRWBTC XC PBTGXRP PCS PBTGXRPCH XC HRDIAPCS."
   b. With key 4: "MQTSWXSV: E VMZEP EWTMVERX XS TYFPMG LSRSVW."
3. Encrypt the following sentence with the key 0: "This is a silly example."
4. Here are some words and their encryptions. Which key was used for each word?
   a. ROSEBUD – LIMYVOX
   b. YAMAMOTO – PRDRDFKF
   c. ASTRONOMY – HZAYVUVTF
5. What does this sentence encrypted with key 8 decrypt to with key 9? "UMMSVMAA: Cvkwuuwv xibqmvkm qv xtivvqvo i zmdmvom bpib qa ewzbp epqtm."

2

PROGRAMMING IN THE INTERACTIVE SHELL

"The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform."
—Ada Lovelace, October 1842

Before you can write encryption programs, you need to learn some basic programming concepts. These concepts include values, operators, expressions, and variables.

TOPICS COVERED IN THIS CHAPTER

- Operators
- Values
- Integers and floating-point numbers
- Expressions
- Evaluating expressions
- Storing values in variables
- Overwriting variables

Let's start by exploring how to do some simple math in Python's interactive shell. Be sure to read this book next to your computer so you can enter the short code examples and see what they do. Developing muscle memory from typing programs will help you remember how Python code is constructed.

Some Simple Math Expressions

Start by opening IDLE (see "Starting IDLE" on page xxvii). You'll see the interactive shell and the cursor blinking next to the `>>>` prompt. The interactive shell can work just like a calculator. Type `2 + 2` into the shell and press ENTER on your keyboard. (On some keyboards, this is the RETURN key.) The computer should respond by displaying the number `4`, as shown in Figure 2-1.

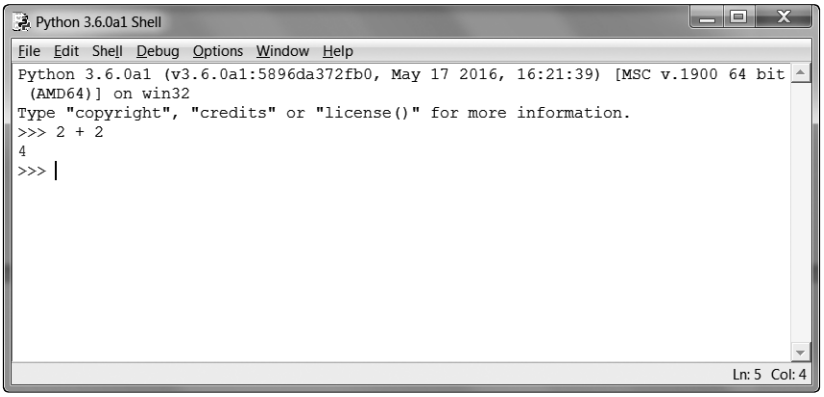


Figure 2-1: Type `2 + 2` into the shell.

In the example in Figure 2-1, the `+` sign tells the computer to add the numbers `2` and `2`, but Python can do other calculations as well, such as subtract numbers using the minus sign (`-`), multiply numbers with an asterisk (`*`), or divide numbers with a forward slash (`/`). When used in this way, `+`, `-`, `*`, and `/` are called *operators* because they tell the computer to perform an operation on the numbers surrounding them. Table 2-1 summarizes the Python math operators. The `2`s (or other numbers) are called *values*.

Table 2-1: Math Operators in Python

Operator	Operation
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division

By itself, `2 + 2` isn't a program; it's just a single instruction. Programs are made of many of these instructions.

Integers and Floating-Point Values

In programming, whole numbers, such as `4`, `0`, and `99`, are called *integers*. Numbers with decimal points (`3.5`, `42.1`, and `5.0`) are called *floating-point numbers*. In Python, the number `5` is an integer, but if you wrote it as `5.0`, it would be a floating-point number.

Integers and floating points are *data types*. The value `42` is a value of the integer, or *int*, data type. The value `7.5` is a value of the floating point, or *float*, data type.

Every value has a data type. You'll learn about a few other data types (such as strings in Chapter 3), but for now just remember that any time we talk about a value, that value is of a certain data type. It's usually easy to identify the data type just by looking at how the value is written. Ints are numbers without decimal points. Floats are numbers with decimal points. So `42` is an int, but `42.0` is a float.

Expressions

You've already seen Python solve one math problem, but Python can do a lot more. Try typing the following math problems into the shell, pressing the ENTER key after each one:

```
❶ >>> 2+2+2+2+2
10
>>> 8*6
48
❷ >>> 10-5+6
11
❸ >>> 2  +       2
4
```

These math problems are called *expressions*. Computers can solve millions of these problems in seconds. Expressions are made up of values (the numbers) connected by operators (the math signs), as shown in Figure 2-2. You can have as many numbers in an expression as you want ❶, as long as they're connected by operators; you can even use multiple types of operators in a single expression ❷. You can also enter any number of spaces between the integers and these operators ❸. But be sure to always start an expression at the beginning of the line, with no spaces in front, because spaces at the beginning of a line change how Python interprets instructions. You'll learn more about spaces at the beginning of a line in "Blocks" on page 45.

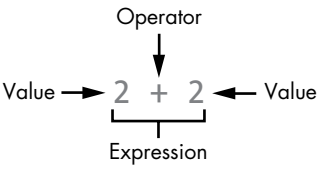


Figure 2-2: An expression is made up of values (like 2) and operators (like +).

Order of Operations

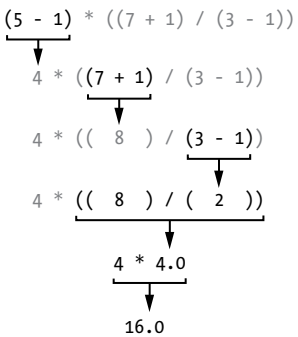
You might remember the phrase "order of operations" from your math class. For example, multiplication is done before addition. The expression `2 + 4 * 3` evaluates to `14` because multiplication is done first to evaluate `4 * 3`, and then `2` is added. Parentheses can make different operators go first. In the expression `(2 + 4) * 3`, the addition is done first to evaluate `(2 + 4)`, and then that sum is multiplied by `3`. The parentheses make the expression evaluate to `18` instead of `14`. The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. Operations inside parentheses are evaluated first; next the `*` and `/` operators are evaluated from left to right; and then the `+` and `-` operators are evaluated from left to right.

Evaluating Expressions

When a computer solves the expression `10 + 5` and gets the value `15`, we say it has *evaluated* the expression. Evaluating an expression reduces the expression to a single value, just like solving a math problem reduces the problem to a single number: the answer.

The expressions `10 + 5` and `10 + 3 + 2` have the same value, because they both evaluate to `15`. Even single values are considered expressions: the expression `15` evaluates to the value `15`.

Python continues to evaluate an expression until it becomes a single value, as in the following:



Python evaluates an expression starting with the innermost, leftmost parentheses. Even when parentheses are nested in each other, the parts of expressions inside them are evaluated with the same rules as any other expression. So when Python encounters `((7 + 1) / (3 - 1))`, it first solves the expression in the leftmost inner parentheses, `(7 + 1)`, and then solves the expression on the right, `(3 - 1)`. When each expression in the inner parentheses is reduced to a single value, the expressions in the outer parentheses are then evaluated. Notice that division evaluates to a floating-point value. Finally, when there are no more expressions in parentheses, Python performs any remaining calculations in the order of operations.

In an expression, you can have two or more values connected by operators, or you can have just one value, but if you enter one value and an operator into the interactive shell, you'll get an error message:

```
>>> 5 +
SyntaxError: invalid syntax
```

This error happens because `5 +` is not an expression. Expressions with multiple values need operators to connect those values, and in the Python language, the `+` operator expects to connect two values. A *syntax error* means that the computer doesn't understand the instruction you gave it because you typed it incorrectly. This may not seem important, but computer programming isn't just about telling the computer what to do—it's also about knowing the correct way to give the computer instructions that it can follow.

ERRORS ARE OKAY!

It's perfectly fine to make errors! You won't break your computer by entering code that causes errors. Python will simply tell you an error has occurred and then display the `>>>` prompt again. You can continue entering new code into the interactive shell.

Until you gain more programming experience, error messages might not make a lot of sense to you. However, you can always google the error message text to find web pages that explain that specific error. You can also go to *https://www.nostarch.com/crackingcodes/* to see a list of common Python error messages and their meanings.

Storing Values with Variables

Programs often need to save values to use later in the program. You can store values in *variables* by using the `=` sign (called the *assignment operator*). For example, to store the value `15` in a variable named `spam`, enter `spam = 15` into the shell:

```
>>> spam = 15
```

You can think of the variable like a box with the value `15` inside it (as shown in Figure 2-3). The variable name `spam` is the label on the box (so we can tell one variable from another), and the value stored in it is like a note inside the box.

When you press ENTER, you won't see anything except a blank line in response. Unless you see an error message, you can assume that the instruction executed successfully. The next `>>>` prompt appears so you can enter the next instruction.

This instruction with the = assignment operator (called an *assignment statement*) creates the variable `spam` and stores the value `15` in it. Unlike expressions, *statements* are instructions that don't evaluate to any value; instead, they just perform an action. This is why no value is displayed on the next line in the shell.

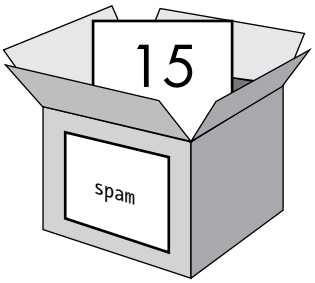


Figure 2-3: Variables are like boxes with names that can hold value.

Figuring out which instructions are expressions and which are statements might be confusing. Just remember that if a Python instruction evaluates to a single value, it's an expression. If it doesn't, it's a statement.

An assignment statement is written as a variable, followed by the = operator, followed by an expression, as shown in Figure 2-4. The value that the expression evaluates to is stored inside the variable.

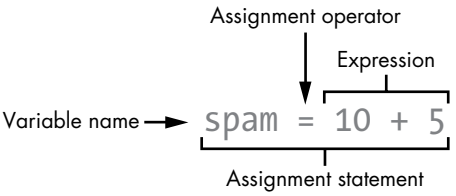


Figure 2-4: The parts of an assignment statement

Keep in mind that variables store single values, not the expressions they are assigned. For example, if you enter the statement `spam = 10 + 5`, the expression `10 + 5` is first evaluated to `15` and then the value `15` is stored in the variable `spam`, as we can see by entering the variable name into the shell:

```
>>> spam = 10 + 5
>>> spam
15
```

A variable by itself is an expression that evaluates to the value stored in the variable. A value by itself is also an expression that evaluates to itself:

```
>>> 15
15
```

And here's an interesting twist. If you now enter `spam + 5` into the shell, you'll get the integer `20`:

```
>>> spam = 15
>>> spam + 5
20
```

As you can see, variables can be used in expressions the same way values can. Because the value of `spam` is `15`, the expression `spam + 5` evaluates to the expression `15 + 5`, which then evaluates to `20`.

Overwriting Variables

You can change the value stored in a variable by entering another assignment statement. For example, enter the following:

```
>>> spam = 15
❶ >>> spam + 5
❷ 20
❸ >>> spam = 3
❹ >>> spam + 5
❺ 8
```

The first time you enter `spam + 5` ❶, the expression evaluates to `20` ❷ because you stored the value `15` inside the variable `spam`. But when you enter `spam = 3` ❸, the value `15` is *overwritten* (that is, replaced) with the value `3`, as shown in Figure 2-5. Now when you enter `spam + 5` ❹, the expression evaluates to `8` ❺ because `spam + 5` evaluates to `3 + 5`. The old value in `spam` is forgotten.

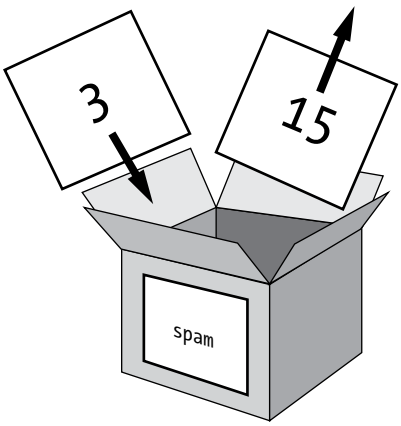


Figure 2-5: The value 15 in spam is overwritten by the value 3.

You can even use the value in the `spam` variable to assign `spam` a new value:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam
20
```

The assignment statement `spam = spam + 5` tells the computer that "the new value of the `spam` variable is the current value of `spam` plus five." The variable on the left side of the `=` sign is assigned the value of the expression on the right side. You can keep increasing the value in `spam` by `5` several times:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
```

The value in `spam` is changed each time `spam = spam + 5` is executed. The value stored in `spam` ends up being `30`.

Variable Names

Although the computer doesn't care what you name your variables, you should. Giving variables names that reflect what type of data they contain makes it easier to understand what a program does. You could give your variables names like `abrahamLincoln` or `monkey` even if your program had nothing to do with Abraham Lincoln or monkeys—the computer would still run the program (as long as you consistently used `abrahamLincoln` or `monkey`). But when you return to a program after not seeing it for a long time, you might not remember what each variable does.

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes *Stuff*. You'd never find anything! The variable names `spam`, `eggs`, `bacon`, and so on (inspired by the *Monty Python* "Spam" sketch) are used as generic names for the examples in this book and in much of Python's documentation, but in your programs, a descriptive name helps make your code more readable.

Variable names (as well as everything else in Python) are case sensitive. *Case sensitive* means the same variable name in a different case is considered an entirely different variable. For example, `spam`, `SPAM`, `Spam`, and `sPAM` are considered four different variables in Python. They each can contain their own separate values and can't be used interchangeably.

Summary

So when are we going to start making encryption programs? Soon. But before you can hack ciphers, you need to learn just a few more basic programming concepts so there's one more programming chapter you need to read.

In this chapter, you learned the basics of writing Python instructions in the interactive shell. Python needs you to tell it exactly what to do in a way it expects, because computers only understand very simple instructions. You learned that Python can evaluate expressions (that is, reduce the expression to a single value) and that expressions are values (such as `2` or `5`) combined with operators (such as `+` or `-`). You also learned that you can store values inside variables so your program can remember them to use later on.

The interactive shell is a useful tool for learning what Python instructions do because it lets you enter them one at a time and see the results. In Chapter 3, you'll create programs that contain many instructions that are executed in sequence rather than one at a time. We'll discuss some more basic concepts, and you'll write your first program!

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Which is the operator for division, `/` or `\` ?
2. Which of the following is an integer value, and which is a floating-point value?

```
42
3.141592
```

3. Which of the following lines are *not* expressions?

```
4 x 10 + 2
3 * 7 + 1
2 +
42
2 + 2
spam = 42
```

4. If you enter the following lines of code into the interactive shell, what do lines ❶ and ❷ print?

```
spam = 20
❶ spam + 20
SPAM = 30
❷ spam
```

3

STRINGS AND WRITING PROGRAMS

"The only way to learn a new programming language is by writing programs in it."
—Brian Kernighan and Dennis Ritchie,
The C Programming Language

Chapter 2 gave you enough integers and math for now. Python is more than just a calculator. Because cryptography is all about dealing with text values by turning plaintext into ciphertext and back again, you'll learn how to store, combine, and display text on the screen in this chapter. You'll also make your first program, which greets the user with the text "Hello, world!" and lets the user input their name.

TOPICS COVERED IN THIS CHAPTER

- Strings
- String concatenation and replication
- Indexes and slices
- The `print()` function
- Writing source code with IDLE
- Saving and running programs in IDLE
- Comments
- The `input()` function

Working with Text Using String Values

In Python, we work with little chunks of text called string values (or simply *strings*). All of our cipher and hacking programs deal with string values to turn plaintext like `'One if by land, two if by space'` into ciphertext like `'b1rJvsJo!Jyn1q,J7O2JvsJo!J63nprM'`. The plaintext and ciphertext are represented in our program as string values, and there are many ways in which Python code can manipulate these values.

You can store string values inside variables just as you can with integer and floating-point values. When you type a string, put it between two single quotes (`'`) to show where the string starts and ends. Enter the following into the interactive shell:

```
>>> spam = 'hello'
```

The single quotes are not part of the string value. Python knows that `'hello'` is a string and `spam` is a variable because strings are surrounded by quotes and variable names are not.

If you enter `spam` into the shell, you will see the contents of the `spam` variable (the `'hello'` string):

```
>>> spam = 'hello'
>>> spam
'hello'
```

This is because Python evaluates a variable to the value stored inside it: in this case, the string `'hello'`. Strings can have almost any keyboard character in them. These are all examples of strings:

```
>>> 'hello'
'hello'
```

```
>>> 'KITTENS'
'KITTENS'
>>> ''
''
>>> '7 apples, 14 oranges, 3 lemons'
'7 apples, 14 oranges, 3 lemons'
>>> 'Anything not pertaining to elephants is irrelephant.'
'Anything not pertaining to elephants is irrelephant.'
>>> 'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
'O*&#wY%*&OcfsdYO*&gfC%YO*&%3yc8r2'
```

Notice that the `''` string has zero characters in it; there is nothing between the single quotes. This is known as a *blank string* or *empty string*.

String Concatenation with the + Operator

You can add two string values to create one new string by using the `+` operator. Doing so is called *string concatenation*. Enter `'Hello,' + 'world!'` into the shell:

```
>>> 'Hello,' + 'world!'
'Hello,world!'
```

Python concatenates *exactly* the strings you tell it to concatenate, so it won't put a space between strings when you concatenate them. If you want a space in the resulting string, there must be a space in one of the two original strings. To put a space between `'Hello,'` and `'world!'`, you can put a space at the end of the `'Hello,'` string and before the second single quote, like this:

```
>>> 'Hello, ' + 'world!'
'Hello, world!'
```

The `+` operator can concatenate two string values into a new string value (`'Hello, ' + 'world!'` to `'Hello, world!'`), just like it can add two integer values to result in a new integer value (`2 + 2` to `4`). Python knows what the `+` operator should do because of the data types of the values. As you learned in Chapter 2, the data type of a value tells us (and the computer) what kind of data the value is.

You can use the `+` operator in an expression with two or more strings or integers as long as the data types match. If you try to use the operator with one string and one integer, you'll get an error. Enter this code into the interactive shell:

```
>>> 'Hello' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> 'Hello' + '42'
'Hello42'
```

The first line of code causes an error because `'Hello'` is a string and `42` is an integer. But in the second line of code, `'42'` is a string, so Python concatenates it.

String Replication with the * Operator

You can also use the * operator on a string and an integer to do *string replication*. This replicates (that is, repeats) a string by however many times the integer value is. Enter the following into the interactive shell:

```
❶ >>> 'Hello' * 3
'HelloHelloHello'
>>> spam = 'Abcdef'
❷ >>> spam = spam * 3
>>> spam
'AbcdefAbcdefAbcdef'
```

To replicate a string, type the string, then the * operator, and then the number of times you want the string to repeat ❶. You can also store a string, like we've done with the `spam` variable, and then replicate the variable instead ❷. You can even store a replicated string back into the same variable or a new variable.

As you saw in Chapter 2, the * operator can work with two integer values to multiply them. But it can't work with two string values, which would cause an error, like this:

```
>>> 'Hello' * 'world!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

String concatenation and string replication show that operators in Python can do different tasks based on the data types of the values they operate on. The + operator can do addition or string concatenation. The * operator can do multiplication or string replication.

Getting Characters from Strings Using Indexes

Your encryption programs often need to get a single character from a string, which you can accomplish through indexing. With *indexing*, you add square brackets [and] to the end of a string value (or a variable containing a string) with a number between them to access one character. This number is called the *index*, and it tells Python which position in the string has the character you want. Python indexes start at `0`, so the index of the first character in a string is `0`. The index `1` is for the second character, the index `2` is for the third character, and so on.

Enter the following into the interactive shell:

```
>>> spam = 'Hello'
>>> spam[0]
'H'
```

```
>>> spam[1]
'e'
>>> spam[2]
'l'
```

Notice that the expression `spam[0]` evaluates to the string value `'H'`, because `H` is the first character in the string `'Hello'` and indexes start at `0`, not `1` (see Figure 3-1).

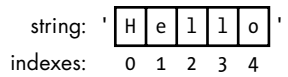


Figure 3-1: The string `'Hello'` and its indexes

You can use indexing with a variable containing a string value, as we did with the previous example, or a string value by itself, like this:

```
>>> 'Zophie'[2]
'p'
```

The expression `'Zophie'[2]` evaluates to the third string value, which is a `'p'`. This `'p'` string is just like any other string value and can be stored in a variable. Enter the following into the interactive shell:

```
>>> eggs = 'Zophie'[2]
>>> eggs
'p'
```

If you enter an index that is too large for the string, Python displays an `"index out of range"` error message, as you can see in the following code:

```
>>> 'Hello'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

There are five characters in the string `'Hello'`, so if you try to use the index `10`, Python displays an error.

Negative Indexes

Negative indexes start at the end of a string and go backward. The negative index `-1` is the index of the *last* character in a string. The index `-2` is the index of the second to last character, and so on, as shown in Figure 3-2.

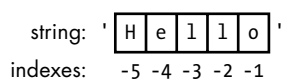


Figure 3-2: The string `'Hello'` and its negative indexes

Enter the following into the interactive shell:

```
>>> 'Hello'[-1]
'o'
>>> 'Hello'[-2]
'l'
>>> 'Hello'[-3]
'l'
```

```
>>> 'Hello'[-4]
'e'
>>> 'Hello'[-5]
'H'
>>> 'Hello'[0]
'H'
```

Notice that `-5` and `0` are the indexes for the same character. Most of the time, your code will use positive indexes, but sometimes it's easier to use negative ones.

Getting Multiple Characters from Strings Using Slices

If you want to get more than one character from a string, you can use slicing instead of indexing. A *slice* also uses the [and] square brackets but has two integer indexes instead of one. The two indexes are separated by a colon (`:`) and tell Python the index of the first and last characters in the slice. Enter the following into the interactive shell:

```
>>> 'Howdy'[0:3]
'How'
```

The string that the slice evaluates to begins at the first index value and goes up to, but does not include, the second index value. Index `0` of the string value `'Howdy'` is `H` and index `3` is `d`. Because a slice goes up to but does not include the second index, the slice `'Howdy'[0:3]` evaluates to the string value `'How'`.

Enter the following into the interactive shell:

```
>>> 'Hello, world!'[0:5]
'Hello'
>>> 'Hello, world!'[7:13]
'world!'
>>> 'Hello, world!'[-6:-1]
'world'
>>> 'Hello, world!'[7:13][2]
'r'
```

Notice that the expression `'Hello, world!'[7:13][2]` first evaluates the list slice to `'world!'[2]` and then further evaluates to `'r'`.

Unlike indexes, slicing never gives you an error if you give too large an index for the string. It'll just return the widest matching slice it can:

```
>>> 'Hello'[0:999]
'Hello'
>>> 'Hello'[2:999]
'llo'
>>> 'Hello'[1000:2000]
''
```

The expression `'Hello'[1000:2000]` returns a blank string because the index `1000` is after the end of the string, so there are no possible characters this slice could include. Although our examples don't show this, you can also slice strings stored in variables.

Blank Slice Indexes

If you omit the first index of a slice, Python will automatically use index `0` for the first index. The expressions `'Howdy'[0:3]` and `'Howdy'[:3]` evaluate to the same string:

```
>>> 'Howdy'[:3]
'How'
>>> 'Howdy'[0:3]
'How'
```

If you omit the second index, Python will automatically use the rest of the string starting from the first index:

```
>>> 'Howdy'[2:]
'wdy'
```

You can use blank indexes in many different ways. Enter the following into the shell:

```
>>> myName = 'Zophie the Fat Cat'
>>> myName[-7:]
'Fat Cat'
>>> myName[:10]
'Zophie the'
>>> myName[7:]
'the Fat Cat'
```

As you can see, you can even use negative indexes with a blank index. Because `-7` is the starting index in the first example, Python counts backward seven characters from the end and uses that as its starting index. Then it returns everything from that index to the end of the string because of the second blank index.

Printing Values with the print() Function

Let's try another type of Python instruction: a `print()` function call. Enter the following into the interactive shell:

```
>>> print('Hello!')
Hello!
>>> print(42)
42
```

A *function* (like `print()` in this example) has code inside it that performs a task, such as printing values onscreen. Many different functions come with Python and can perform useful tasks for you. To *call* a function means to execute the code inside the function.

The instructions in this example pass a value to `print()` between the parentheses, and the `print()` function prints the value to the screen. The values that are passed when a function is called are *arguments*. When you write programs, you'll use `print()` to make text appear on the screen.

You can pass an expression to `print()` instead of a single value. This is because the value that is actually passed to `print()` is the evaluated value of that expression. Enter this string concatenation expression into the interactive shell:

```
>>> spam = 'Al'
>>> print('Hello, ' + spam)
Hello, Al
```

The `'Hello, ' + spam` expression evaluates to `'Hello, ' + 'Al'`, which then evaluates to the string value `'Hello, Al'`. This string value is what is passed to the `print()` call.

Printing Escape Characters

You might want to use a character in a string value that would confuse Python. For example, you might want to use a single quote character as part of a string. But you'd get an error message because Python thinks that single quote is the quote ending the string value and the text after it is bad Python code, instead of the rest of the string. Enter the following into the interactive shell to see the error in action:

```
>>> print('Al's cat is named Zophie.')
SyntaxError: invalid syntax
```

To use a single quote in a string, you need to use an *escape character*. An escape character is a backslash character followed by another character—for example, `\t`, `\n`, or `\'`. The slash tells Python that the character after the slash has a special meaning. Enter the following into the interactive shell.

```
>>> print('Al\'s cat is named Zophie.')
Al's cat is named Zophie.
```

Now Python will know the apostrophe is a character in the string value, not Python code marking the end of the string.

Table 3-1 shows some escape characters you can use in Python.

Table 3-1: Escape Characters

Escape character	Printed result
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\n	Newline
\t	Tab

The backslash always precedes an escape character. Even if you just want a backslash in your string, you can't add a backslash alone because Python will interpret the next character as an escape character. For example, this line of code wouldn't work correctly:

```
>>> print('It is a green\teal color.')
It is a green    eal color.
```

The `'t'` in `'teal'` is identified as an escape character because it comes after a backslash. The escape character `\t` simulates pushing the TAB key on your keyboard.

Instead, enter this code:

```
>>> print('It is a green\\teal color.')
It is a green\teal color.
```

This time the string will print as you intended, because putting a second backslash in the string makes the backslash the escape character.

Quotes and Double Quotes

Strings don't always have to be between two single quotes in Python. You can use double quotes instead. These two lines print the same thing:

```
>>> print('Hello, world!')
Hello, world!
>>> print("Hello, world!")
Hello, world!
```

But you can't mix single and double quotes. This line gives you an error:

```
>>> print('Hello, world!")
SyntaxError: EOL while scanning string literal
```

I prefer to use single quotes because they're a bit easier to type than double quotes and Python doesn't care either way.

But just like you have to use the escape character \' to have a single quote in a string surrounded by single quotes, you need the escape character \" to have a double quote in a string surrounded by double quotes. For example, look at these two lines:

```
>>> print('Al\'s cat is Zophie. She says, "Meow."')
Al's cat is Zophie. She says, "Meow."
>>> print("Zophie said, \"I can say things other than 'Meow' you know.\"")
Zophie said, "I can say things other than 'Meow' you know."
```

You don't need to escape double quotes in single-quote strings, and you don't need to escape single quotes in double-quote strings. The Python interpreter is smart enough to know that if a string starts with one kind of quote, the other kind of quote doesn't mean the string is ending.

Writing Programs in IDLE's File Editor

Until now, you've been entering instructions one at a time into the interactive shell. But when you write programs, you'll enter several instructions and have them run without waiting on you for the next one. It's time to write your first program!

The name of the software program that provides the interactive shell is called IDLE (**I**ntegrated **D**eve**L**opment **E**nvironment). In addition to the interactive shell, IDLE also has a *file editor*, which we'll open now.

At the top of the Python shell window, select **File ▸ New Window**. A new blank window, the file editor, will appear for you to enter a program, as shown in Figure 3-3. The bottom-right corner of the file editor window shows you what line and column the cursor currently is on.

Figure 3-3: The file editor window with the cursor at line 1, column 0

You can tell the difference between the file editor window and the interactive shell window by looking for the >>> prompt. The interactive shell always displays the prompt, and the file editor doesn't.

Source Code for the "Hello, World!" Program

Traditionally, programmers who are learning a new language make their first program display the text `"Hello, world!"` on the screen. We'll create our own "Hello, world!" program next by entering text into the new file editor window. We call this text the program's *source code* because it contains the instructions that Python will follow to determine exactly how the program should behave.

You can download the "Hello, world!" source code from *https://www.nostarch.com/crackingcodes/*. If you get errors after entering this code, compare it to the book's code using the online diff tool (see "Checking Your Source Code with the Online Diff Tool" next). Remember that you don't type the line numbers; they only appear in this book to aid explanation.

hello.py

```
1. # This program says hello and asks for my name.
2. print('Hello, world!')
3. print('What is your name?')
4. myName = input()
5. print('It is good to meet you, ' + myName)
```

The IDLE program will display different types of instructions in different colors. When you're done entering this code, the window should look like Figure 3-4.

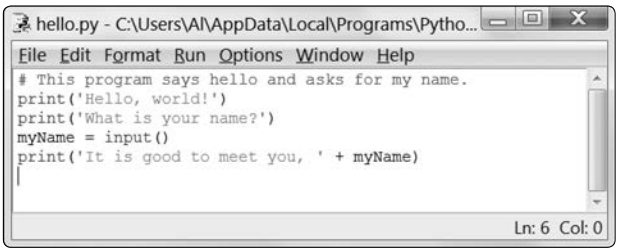


Figure 3-4: The file editor window will look like this after you enter the code.

Checking Your Source Code with the Online Diff Tool

Even though you could copy and paste or download the *hello.py* code from this book's website, you should still type this program manually. Doing so will give you more familiarity with the code in the program. However, you might make some mistakes while typing it into the file editor.

To compare the code you typed to the code in this book, use the online diff tool shown in Figure 3-5. Copy the text of your code and then navigate to the diff tool on the book's website at *https://www.nostarch.com/crackingcodes/*. Select the *hello.py* program from the drop-down menu. Paste your code into the text field on this web page and click the **Compare** button. The diff tool shows any differences between your code and the code in this book. This is an easy way to find any typos causing errors in your program.

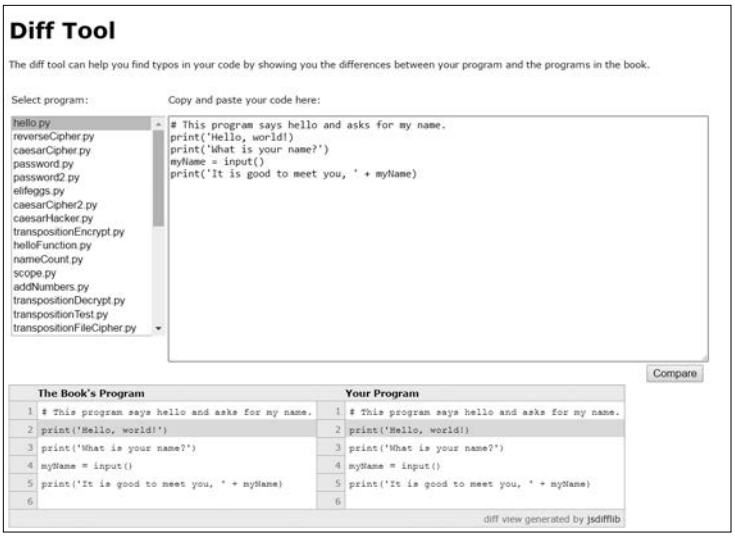


Figure 3-5: The online diff tool

Using IDLE to Access Your Program Later

When you write programs, you might want to save them and come back to them later, especially after you've typed a very long program. IDLE has features for saving and opening programs just like a word processer has features to save and reopen your documents.

Saving Your Program

After you've entered your source code, save it so you won't have to retype it each time you want to run it. Choose **File ▸ Save As** from the menu at the top of the file editor window. The Save As dialog should open, as shown in Figure 3-6. Enter **`hello.py`** in the **File Name** field and click **Save**.

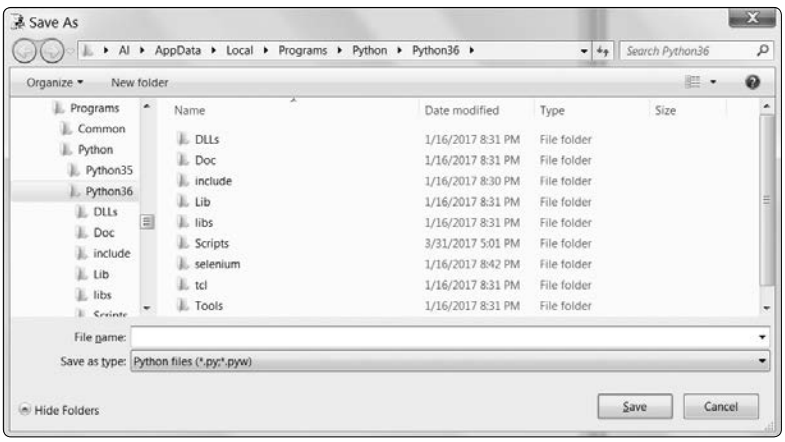


Figure 3-6: Saving the program

You should save your programs often as you type them so you won't lose your work if the computer crashes or if you accidentally exit from IDLE. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on macOS to save your file.

Running Your Program

Now it's time to run your program. Select **Run ▸ Run Module** or just press the F5 key on your keyboard. Your program should run in the shell window that appeared when you first started IDLE. Remember that you must press F5 from the file editor's window, not the interactive shell's window.

When the program asks for your name, enter it, as shown in Figure 3-7.

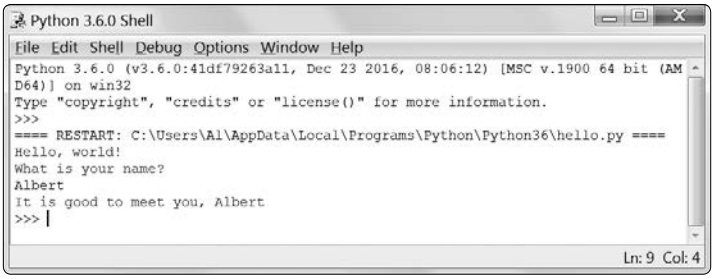


Figure 3-7: The interactive shell looks like this when running the "Hello, world!" program.

Now when you press ENTER, the program should greet you (the *user*, that is, the one using the program) by name. Congratulations! You've written your first program. You are now a beginning computer programmer. (If you like, you can run this program again by pressing F5 again.)

If instead you get an error that looks like this, it means you are running the program with Python 2 instead of Python 3:

```
Hello, world!
What is your name?
Albert
Traceback (most recent call last):
  File "C:/Python27/hello.py", line 4, in <module>
    myName = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

The error is caused by the `input()` function call, which behaves differently in Python 2 and 3. Before continuing, install Python 3 by following the instructions in "Downloading and Installing Python" on page xxv.

Opening the Programs You've Saved

Close the file editor by clicking the X in the top corner. To reload a saved program, choose **File ▸ Open** from the menu. Do that now, and in the window that appears, choose *hello.py*. Then click the **Open** button. Your saved *hello.py* program should open in the file editor window.

How the "Hello, World!" Program Works

Each line in the "Hello, world!" program is an instruction that tells Python exactly what to do. A computer program is a lot like a recipe. Do the first step first, then the second, and so on until you reach the end. When the program follows instructions step-by-step, we call it the *program execution*, or just the *execution*.

Each instruction is followed in sequence, beginning from the top of the program and working down the list of instructions. The execution starts at the first line of code and then moves downward. But the execution can also skip around instead of just going from top to bottom; you'll find out how to do this in Chapter 4.

Let's look at the "Hello, world!" program one line at a time to see what it's doing, beginning with line 1.

Comments

Any text following a *hash mark* (`#`) is a comment:

```
1. # This program says hello and asks for my name.
```

Comments are not for the computer but instead are for you, the programmer. The computer ignores them. They're used to remind you what the program does or to tell others who might look at your code what your code does.

Programmers usually put a comment at the top of their code to give the program a title. The IDLE program displays comments in red text to help them stand out. Sometimes, programmers will put a `#` in front of a line of code to temporarily skip it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program doesn't work. You can remove the `#` later when you're ready to put the line back in.

Printing Directions to the User

The next two lines display directions to the user with the `print()` function. A function is like a mini-program inside your program. The great benefit of using functions is that we only need to know what the function does, not how it does it. For instance, you need to know that `print()` displays text onscreen, but you don't need to know the exact code inside the function that does this.

A function call is a piece of code that tells the program to run the code inside a function.

Line 2 of *hello.py* is a call to `print()` (with the string to be printed inside the parentheses). Line 3 is another `print()` call. This time the program displays `'What is your name?'`

```
2. print('Hello, world!')
3. print('What is your name?')
```

We add parentheses to the end of function names to make it clear that we're referring to a function named `print()`, not a variable named `print`. The parentheses at the end of the function tell Python we're using a function, much as the quotes around the number `'42'` tell Python that we're using the string `'42'`, not the integer `42`.

Taking a User's Input

Line 4 has an assignment statement with a variable (`myName`) and the new function call `input()`:

```
4. myName = input()
```

When `input()` is called, the program waits for the user to type in some text and press ENTER. The text string that the user enters (their name) becomes the string value that is stored in `myName`.

Like expressions, function calls evaluate to a single value. The value that the call evaluates to is called the *return value*. (In fact, we can also use the word "returns" to mean the same thing as "evaluates" for function calls.) In this case, the return value of `input()` is the string that the user entered, which should be their name. If the user entered `Albert`, the `input()` call evaluates to (that is, returns) the string `'Albert'`.

Unlike `print()`, the `input()` function doesn't need any arguments, which is why there is nothing between the parentheses.

The last line of the code in *hello.py* is another `print()` call:

```
5. print('It is good to meet you, ' + myName)
```

For line 5's `print()` call, we use the plus operator (`+`) to concatenate the string `'It is good to meet you, '` and the string stored in the `myName` variable, which is the name that the user input into the program. This is how we get the program to greet the user by name.

Ending the Program

When the program executes the last line, it stops. At this point it has *terminated* or *exited*, and all the variables are forgotten by the computer, including the string stored in `myName`. If you try running the program again and entering a different name, it will print that name.

```
Hello, world!
What is your name?
Zophie
It is good to meet you, Zophie
```

Remember that the computer only does exactly what you program it to do. In this program, it asks you for your name, lets you enter a string, and then says hello and displays the string you entered.

But computers are dumb. The program doesn't care if you enter your name, someone else's name, or just something silly. You can type in anything you want, and the computer will treat it the same way:

```
Hello, world!
What is your name?
poop
It is good to meet you, poop
```

Summary

Writing programs is just about knowing how to speak the computer's language. You learned a bit about how to do this in Chapter 2, and now you've put together several Python instructions to make a complete program that asks for the user's name and greets that user.

In this chapter, you learned several new techniques to manipulate strings, like using the + operator to concatenate strings. You can also use indexing and slicing to create a new string from part of a different string.

The rest of the programs in this book will be more complex and sophisticated, but they'll all be explained line by line. You can always enter instructions into the interactive shell to see what they do before you put them into a complete program.

Next, we'll start writing our first encryption program: the reverse cipher.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. If you assign `spam = 'Cats'`, what do the following lines print?

```
spam + spam + spam
spam * 3
```

2. What do the following lines print?

```
print("Dear Alice,\nHow are you?\nSincerely,\nBob")
print('Hello' + 'Hello')
```

3. If you assign `spam = 'Four score and seven years is eighty seven years.'`, what would each of the following lines print?

```
print(spam[5])
print(spam[-3])
print(spam[0:4] + spam[5])
print(spam[-3:-1])
print(spam[:10])
print(spam[-5:])
print(spam[:])
```

4. Which window displays the >>> prompt, the interactive shell or the file editor?
5. What does the following line print?

```
#print('Hello, world!')
```

4

THE REVERSE CIPHER

"Every man is surrounded by a neighborhood of voluntary spies."
—Jane Austen, Northanger Abbey

The reverse cipher encrypts a message by printing it in reverse order. So "Hello, world!" encrypts to "!dlrow ,olleH". To decrypt, or get the original message, you simply reverse the encrypted message. The encryption and decryption steps are the same.

However, this reverse cipher is weak, making it easy to figure out the plaintext. Just by looking at the ciphertext, you can figure out the message is in reverse order.

.syas ti tahw tuo erugif llits ylbaborp nac uoy ,detpyrcne si siht hguoht neve ,elpmaxe roF

But the code for the reverse cipher program is easy to explain, so we'll use it as our first encryption program.

TOPICS COVERED IN THIS CHAPTER

- The `len()` function
- `while` loops
- Boolean data type
- Comparison operators
- Conditions
- Blocks

Source Code for the Reverse Cipher Program

In IDLE, click **File ▸ New Window** to create a new file editor window. Enter the following code, save it as *reverseCipher.py*, and press F5 to run it, but remember not to type the numbers before each line:

reverseCipher.py

```
 1. # Reverse Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. message = 'Three can keep a secret, if two of them are dead.'
 5. translated = ''
 6.
 7. i = len(message) - 1
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Sample Run of the Reverse Cipher Program

When you run the *reverseCipher.py* program, the output looks like this:

```
.daed era meht fo owt fi ,terces a peek nac eerhT
```

To decrypt this message, copy the `.daed era meht fo owt fi ,terces a peek nac eerhT` text to the clipboard by highlighting the message and pressing CTRL-C on Windows and Linux or ⌘-C on macOS. Then paste it (using CTRL-V on Windows and Linux or ⌘-V on macOS) as the string value stored in `message` on line 4. Be sure to retain the single quotes at the beginning and end of the string. The new line 4 looks like this (with the change in bold):

```
4. message = '.daed era meht fo owt fi ,terces a peek nac eerhT'
```

Now when you run the *reverseCipher.py* program, the output decrypts to the original message:

```
Three can keep a secret, if two of them are dead.
```

Setting Up Comments and Variables

The first two lines in *reverseCipher.py* are comments explaining what the program is and the website where you can find it.

```
1. # Reverse Cipher
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
```

The `BSD Licensed` part means this program is free to copy and modify by anyone as long as the program retains the credits to the original author (in this case, the book's website at `https://www.nostarch.com/crackingcodes/` in the second line). I like to have this info in the file so if it gets copied around the internet, a person who downloads it always knows where to look for the original source. They'll also know this program is open source software and free to distribute to others.

Line 3 is just a blank line, and Python skips it. Line 4 stores the string we want to encrypt in a variable named `message`:

```
4. message = 'Three can keep a secret, if two of them are dead.'
```

Whenever we want to encrypt or decrypt a new string, we just type the string directly into the code on line 4.

The `translated` variable on line 5 is where our program will store the reversed string:

```
5. translated = ''
```

At the start of the program, the `translated` variable contains this blank string. (Remember that the blank string is two single quote characters, not one double quote character.)

Finding the Length of a String

Line 7 is an assignment statement storing a value in a variable named `i`:

```
7. i = len(message) - 1
```

The expression evaluated and stored in the variable is `len(message) - 1`. The first part of this expression, `len(message)`, is a function call to the `len()` function, which accepts a string argument, just like `print()`, and returns an

integer value of how many characters are in the string (that is, the *length* of the string). In this case, we pass the `message` variable to `len()`, so `len(message)` returns how many characters are in the string value stored in `message`.

Let's experiment with the `len()` function in the interactive shell. Enter the following into the interactive shell:

```
>>> len('Hello')
5
>>> len('')
0
>>> spam = 'Al'
>>> len(spam)
2
>>> len('Hello,' + ' ' + 'world!')
13
```

From the return value of `len()`, we know the string `'Hello'` has five characters in it and the blank string has zero characters in it. If we store the string `'Al'` in a variable and then pass the variable to `len()`, the function returns `2`. If we pass the expression `'Hello,' + ' ' + 'world!'` to the `len()` function, it returns `13`. The reason is that `'Hello,' + ' ' + 'world!'` evaluates to the string value `'Hello, world!'`, which has 13 characters in it. (The space and the exclamation point count as characters.)

Now that you understand how the `len()` function works, let's return to line 7 of the *reverseCipher.py* program. Line 7 finds the index of the last character in `message` by subtracting 1 from `len(message)`. It has to subtract 1 because the indexes of, for example, a 5-character length string like `'Hello'` are from 0 to 4. This integer is then stored in the `i` variable.

Introducing the while Loop

Line 8 is a type of Python instruction called a `while` loop or `while` statement:

```
8. while i >= 0:
```

A `while` loop is made up of four parts (as shown in Figure 4-1).

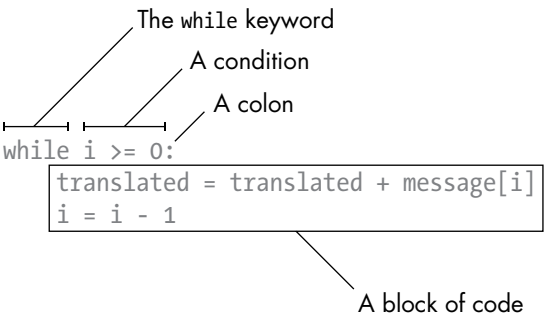


Figure 4-1: The parts of a `while` loop

A *condition* is an expression used in a `while` statement. The block of code in the `while` statement will execute as long as the condition is true.

To understand `while` loops, you first need to learn about Booleans, comparison operators, and blocks.

The Boolean Data Type

The *Boolean* data type has only two values: `True` or `False`. These Boolean values, or *bools*, are case sensitive (you always need to capitalize the *T* and *F*, while leaving the rest in lowercase). They are not string values, so you don't put quotes around `True` or `False`.

Try out some bools by entering the following into the interactive shell:

```
>>> spam = True
>>> spam
True
>>> spam = False
>>> spam
False
```

Like a value of any other data type, bools can be stored in variables.

Comparison Operators

In line 8 of the *reverseCipher.py* program, look at the expression after the `while` keyword:

```
8. while i >= 0:
```

The expression that follows the `while` keyword (the `i >= 0` part) contains two values (the value in the variable `i` and the integer value `0`) connected by the `>=` sign, called the "greater than or equal" operator. The `>=` operator is a *comparison operator*.

We use comparison operators to compare two values and evaluate to a `True` or `False` Boolean value. Table 4-1 lists the comparison operators.

Table 4-1: Comparison Operators

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

Enter the following expressions in the interactive shell to see the Boolean value they evaluate to:

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10.5
False
>>> 10.5 < 11.3
True
>>> 10 < 10
False
```

The expression `0 < 6` returns the Boolean value `True` because the number `0` is less than the number `6`. But because `6` is not less than `0`, the expression `6 < 0` evaluates to `False`. The expression `50 < 10.5` is `False` because `50` isn't less than `10.5`. The expression `10 < 11.3` evaluates to `True` because `10.5` is less than `11.3`.

Look again at `10 < 10`. It's `False` because the number `10` isn't less than the number `10`. They are exactly the same. (If Alice were the same height as Bob, you wouldn't say that Alice was shorter than Bob. That statement would be false.)

Enter some expressions using the `<=` (less than or equal to) and `>=` (greater than or equal to) operators:

```
>>> 10 <= 20
True
>>> 10 <= 10
True
>>> 10 >= 20
False
>>> 20 >= 20
True
```

Notice that `10 <= 10` is `True` because the operator checks if 10 is less than *or equal to* 10. Remember that for the "less than or equal to" and "greater than or equal to" operators, the `<` or `>` sign always comes before the `=` sign.

Now enter some expressions that use the `==` (equal to) and `!=` (not equal to) operators into the shell to see how they work:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
```

These operators work as you would expect for integers. Comparing integers that are equal to each other with the == operator evaluates as True and unequal values as False. When you compare with the != operator, it's the opposite.

String comparisons work similarly:

```
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Goodbye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Goodbye' != 'Hello'
True
```

Capitalization matters to Python, so string values that don't match capitalization exactly are not the same string. For example, the strings 'Hello' and 'HELLO' are not equal to each other, so comparing them with == evaluates to False.

Notice the difference between the assignment operator (=) and the "equal to" comparison operator (==). The single equal sign (=) is used to assign a value to a variable, and the double equal sign (==) is used in expressions to check whether two values are the same. If you're asking Python whether two things are equal, use ==. If you're telling Python to set a variable to a value, use =.

In Python, string and integer values are always considered different values and will never be equal to each other. For example, enter the following into the interactive shell:

```
>>> 42 == 'Hello'
False
>>> 42 == '42'
False
>>> 10 == 10.0
True
```

Even though they look alike, the integer 42 and the string '42' aren't considered equal because a string isn't the same as a number. Integers and floating-point numbers can be equal to each other because they're both numbers.

When you're working with comparison operators, just remember that every expression always evaluates to a True or False value.

Blocks

A *block* is one or more lines of code grouped together with the same minimum amount of *indentation* (that is, the number of spaces in front of the line).

A block begins when a line is indented by four spaces. Any following line that is also indented by at least four spaces is part of the block. When

a line is indented with another four spaces (for a total of eight spaces in front of the line), a new block begins inside the first block. A block ends when there is a line of code with the same indentation as before the block started.

Let's look at some imaginary code (it doesn't matter what the code is, because we're only going to focus on the indentation of each line). The indented spaces are replaced with gray dots here to make them easier to count.

```
1. codecodecode          # 0 spaces of indentation
2. ····codecodecode      # 4 spaces of indentation
3. ····codecodecode      # 4 spaces of indentation
4. ········codecodecode  # 8 spaces of indentation
5. ····codecodecode      # 4 spaces of indentation
6.
7. ····codecodecode      # 4 spaces of indentation
8. codecodecode          # 0 spaces of indentation
```

You can see that line 1 has no indentation; that is, there are zero spaces in front of the line of code. But line 2 has four spaces of indentation. Because this is a larger amount of indentation than the previous line, we know a new block has begun. Line 3 also has four spaces of indentation, so we know the block continues on line 3.

Line 4 has even more indentation (eight spaces), so a new block has begun. This block is inside the other block. In Python, you can have blocks within blocks.

On line 5, the amount of indentation has decreased to four, so we know that the block on the previous line has ended. Line 4 is the only line in that block. Because line 5 has the same amount of indentation as the block in lines 2 and 3, it's still part of the original outer block, even though it's not part of the block on line 4.

Line 6 is a blank line, so we just skip it; it doesn't affect the blocks.

Line 7 has four spaces of indentation, so we know that the block that started on line 2 has continued to line 7.

Line 8 has zero spaces of indentation, which is less indentation than the previous line. This decrease in indentation tells us that the previous block, the block that started on line 2, has ended.

This code shows two blocks. The first block goes from line 2 to line 7. The second block just consists of line 4 (and is inside the other block).

NOTE *Blocks don't always have to be delineated by four spaces. Blocks can use any number of spaces, but the convention is to use four per indentation.*

The while Loop Statement

Let's look at the full `while` statement starting on line 8 of *reverseCipher.py*:

```
8. while i >= 0:
9.     translated = translated + message[i]
```

```
10.     i = i - 1
11.
12. print(translated)
```

A `while` statement tells Python to first check what the condition evaluates to, which on line 8 is `i >= 0`. You can think of the `while` statement `while i >= 0:` as meaning "While the variable `i` is greater than or equal to zero, keep executing the code in the following block." If the condition evaluates to `True`, the program execution enters the block following the `while` statement. By looking at the indentation, you can see that this block is made up of lines 9 and 10. When it reaches the bottom of the block, the program execution jumps back to the `while` statement on line 8 and checks the condition again. If it's still `True`, the execution jumps into the start of the block and runs the code in the block again.

If the `while` statement's condition evaluates to `False`, the program execution skips the code inside the following block and jumps down to the first line after the block (which is line 12).

"Growing" a String

Keep in mind that on line 7, the `i` variable is first set to the length of the `message` minus 1, and the `while` loop on line 8 keeps executing the lines inside the following block until the condition `i >= 0` is `False`:

```
 7. i = len(message) - 1
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     i = i - 1
11.
12. print(translated)
```

Line 9 is an assignment statement that stores a value in the `translated` variable. The value that is stored is the current value of `translated` concatenated with the character at the index `i` in `message`. As a result, the string value stored in `translated` "grows" one character at a time until it becomes the fully encrypted string.

Line 10 is also an assignment statement. It takes the current integer value in `i` and subtracts 1 from it (this is called *decrementing* the variable). Then it stores this value as the new value of `i`.

The next line is 12, but because this line has less indentation, Python knows that the `while` statement's block has ended. So rather than moving on to line 12, the program execution jumps back to line 8 where the `while` loop's condition is checked again. If the condition is `True`, the lines inside the block (lines 9 and 10) are executed again. This keeps happening until the condition is `False` (that is, when `i` is less than `0`), in which case the program execution goes to the first line after the block (line 12).

Let's think about the behavior of this loop to understand how many times it runs the code in the block. The variable `i` starts with the value of the last index of `message`, and the `translated` variable starts as a blank

string. Then inside the loop, the value of `message[i]` (which is the last character in the `message` string, because `i` will have the value of the last index) is added to the end of the `translated` string.

Then the value in `i` is decremented (that is, reduced) by `1`, meaning that `message[i]` will be the second to last character. So while `i` as an index keeps moving from the back of the string in `message` to the front, the string `message[i]` is added to the end of `translated`. This is how `translated` ends up holding the reverse of the string in the `message`. When `i` is finally set to `-1`, which happens when we reach index `0` of the message, the `while` loop's condition is `False`, and the execution jumps to line 12:

```
12. print(translated)
```

At the end of the program on line 12, we print the contents of the `translated` variable (that is, the string `'.daed era meht fo owt fi ,terces a peek nac eerhT'`) to the screen. This shows the user what the reversed string looks like.

If you're still having trouble understanding how the code in the `while` loop reverses the string, try adding the new line (shown in bold) to the loop's block:

```
 8. while i >= 0:
 9.     translated = translated + message[i]
10.     print('i is', i, ', message[i] is', message[i], ', translated is',
          translated)
11.     i = i - 1
12.
13. print(translated)
```

Line 10 prints the values of `i`, `message[i]`, and `translated` along with string labels each time the execution goes through the loop (that is, on each *iteration* of the loop). This time, we aren't using string concatenation but something new. The commas tell the `print()` function that we're printing six separate things, so the function adds a space between them. Now when you run the program, you can see how the `translated` variable "grows." The output looks like this:

```
i is 48 , message[i] is . , translated is .
i is 47 , message[i] is d , translated is .d
i is 46 , message[i] is a , translated is .da
i is 45 , message[i] is e , translated is .dae
i is 44 , message[i] is d , translated is .daed
i is 43 , message[i] is   , translated is .daed
i is 42 , message[i] is e , translated is .daed e
i is 41 , message[i] is r , translated is .daed er
i is 40 , message[i] is a , translated is .daed era
i is 39 , message[i] is   , translated is .daed era
i is 38 , message[i] is m , translated is .daed era m
i is 37 , message[i] is e , translated is .daed era me
i is 36 , message[i] is h , translated is .daed era meh
```

```
i is 35 , message[i] is t , translated is .daed era meht
i is 34 , message[i] is   , translated is .daed era meht
i is 33 , message[i] is f , translated is .daed era meht f
i is 32 , message[i] is o , translated is .daed era meht fo
i is 31 , message[i] is   , translated is .daed era meht fo
i is 30 , message[i] is o , translated is .daed era meht fo o
i is 29 , message[i] is w , translated is .daed era meht fo ow
i is 28 , message[i] is t , translated is .daed era meht fo owt
i is 27 , message[i] is   , translated is .daed era meht fo owt
i is 26 , message[i] is f , translated is .daed era meht fo owt f
i is 25 , message[i] is i , translated is .daed era meht fo owt fi
i is 24 , message[i] is   , translated is .daed era meht fo owt fi
i is 23 , message[i] is , , translated is .daed era meht fo owt fi ,
i is 22 , message[i] is t , translated is .daed era meht fo owt fi ,t
i is 21 , message[i] is e , translated is .daed era meht fo owt fi ,te
i is 20 , message[i] is r , translated is .daed era meht fo owt fi ,ter
i is 19 , message[i] is c , translated is .daed era meht fo owt fi ,terc
i is 18 , message[i] is e , translated is .daed era meht fo owt fi ,terce
i is 17 , message[i] is s , translated is .daed era meht fo owt fi ,terces
i is 16 , message[i] is   , translated is .daed era meht fo owt fi ,terces
i is 15 , message[i] is a , translated is .daed era meht fo owt fi ,terces a
i is 14 , message[i] is   , translated is .daed era meht fo owt fi ,terces a
i is 13 , message[i] is p , translated is .daed era meht fo owt fi ,terces a p
i is 12 , message[i] is e , translated is .daed era meht fo owt fi ,terces a pe
i is 11 , message[i] is e , translated is .daed era meht fo owt fi ,terces a pee
i is 10 , message[i] is k , translated is .daed era meht fo owt fi ,terces a peek
i is 9 , message[i] is   , translated is .daed era meht fo owt fi ,terces a peek
i is 8 , message[i] is n , translated is .daed era meht fo owt fi ,terces a peek n
i is 7 , message[i] is a , translated is .daed era meht fo owt fi ,terces a peek na
i is 6 , message[i] is c , translated is .daed era meht fo owt fi ,terces a peek nac
i is 5 , message[i] is   , translated is .daed era meht fo owt fi ,terces a peek nac
i is 4 , message[i] is e , translated is .daed era meht fo owt fi ,terces a peek nac e
i is 3 , message[i] is e , translated is .daed era meht fo owt fi ,terces a peek nac ee
i is 2 , message[i] is r , translated is .daed era meht fo owt fi ,terces a peek nac eer
i is 1 , message[i] is h , translated is .daed era meht fo owt fi ,terces a peek nac eerh
i is 0 , message[i] is T , translated is .daed era meht fo owt fi ,terces a peek nac eerhT
```

The line of output, `"i is 48 , message[i] is . , translated is ."`, shows what the expressions `i`, `message[i]`, and `translated` evaluate to after the string `message[i]` has been added to the end of `translated` but before `i` is decremented. You can see that the first time the program execution goes through the loop, `i` is set to `48`, so `message[i]` (that is, `message[48]`) is the string `'.'`. The `translated` variable started as a blank string, but when `message[i]` was added to the end of it on line 9, it became the string value `'.'`.

On the next iteration of the loop, the output is `"i is 47 , message[i] is d , translated is .d"`. You can see that `i` has been decremented from `48` to `47`, so now `message[i]` is `message[47]`, which is the `'d'` string. (That's the second `'d'` in `'dead'`.) This `'d'` gets added to the end of `translated`, so `translated` is now the value `'.d'`.

Now you can see how the `translated` variable's string is slowly "grown" from a blank string to the reversed `message`.

Improving the Program with an input() Prompt

The programs in this book are all designed so the strings that are being encrypted or decrypted are typed directly into the source code as assignment statements. This is convenient while we're developing the programs, but you shouldn't expect users to be comfortable modifying the source code themselves. To make the programs easier to use and share, you can modify the assignment statements so they call the `input()` function. You can also pass a string to `input()` so it will display a prompt for the user to enter a string to encrypt. For example, change line 4 in *reverseCipher.py* to this:

```
4. message = input('Enter message: ')
```

When you run the program, it prints the prompt to the screen and waits for the user to enter a message. The message that the user enters will be the string value that is stored in the `message` variable. When you run the program now, you can put in any string you'd like and get output like this:

```
Enter message: Hello, world!
!dlrow ,olleH
```

Summary

We've just completed our second program, which manipulates a string into a new string using techniques from Chapter 3, such as indexing and concatenation. A key part of the program was the `len()` function, which takes a string argument and returns an integer of how many characters are in the string.

You also learned about the Boolean data type, which has only two values, `True` and `False`. Comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=` can compare two values and evaluate to a Boolean value.

Conditions are expressions that use comparison operators and evaluate to a Boolean data type. They are used in `while` loops, which will execute code in the block following the `while` statement until the condition evaluates as `False`. A block is made up of lines with the same level of indentation, including any blocks inside them.

Now that you've learned how to manipulate text, you can start making programs that the user can run and interact with. This is important because text is the main way the user and the computer communicate with each other.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What does the following piece of code print to the screen?

```
print(len('Hello') + len('Hello'))
```

2. What does this code print?

```
i = 0
while i < 3:
    print('Hello')
    i = i + 1
```

3. How about this code?

```
i = 0
spam = 'Hello'
while i < 5:
    spam = spam + spam[i]
    i = i + 1
print(spam)
```

4. And this?

```
i = 0
while i < 4:
    while i < 6:
        i = i + 2
        print(i)
```

5

THE CAESAR CIPHER

"BIG BROTHER IS WATCHING YOU."
—George Orwell, Nineteen Eighty-Four

In Chapter 1, we used a cipher wheel and a chart of letters and numbers to implement the Caesar cipher. In this chapter, we'll implement the Caesar cipher in a computer program.

The reverse cipher we made in Chapter 4 always encrypts the same way. But the Caesar cipher uses keys, which encrypt the message differently depending on which key is used. The keys for the Caesar cipher are the integers from 0 to 25. Even if a cryptanalyst knows the Caesar cipher was used, that alone doesn't give them enough information to break the cipher. They must also know the key.

TOPICS COVERED IN THIS CHAPTER

- The `import` statement
- Constants
- `for` loops
- `if`, `else`, and `elif` statements
- The `in` and `not in` operators
- The `find()` string method

Source Code for the Caesar Cipher Program

Enter the following code into the file editor and save it as *caesarCipher.py*. Then download the *pyperclip.py* module from *https://www.nostarch.com/crackingcodes/* and place it in the same directory (that is, the same folder) as the file *caesarCipher.py*. This module will be imported by *caesarCipher.py*; we'll discuss this in more detail in "Importing Modules and Setting Up Variables" on page 56.

When you're finished setting up the files, press F5 to run the program. If you run into any errors or problems with your code, you can compare it to the code in the book using the online diff tool at *https://www.nostarch.com/crackingcodes/*.

caesarCipher.py

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
14.
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
17.
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
20.
```

```
21. for symbol in message:
22.     # Note: Only symbols in the SYMBOLS string can be
          encrypted/decrypted.
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
25.
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
31.
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
37.
38.         translated = translated + SYMBOLS[translatedIndex]
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
42.
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

Sample Run of the Caesar Cipher Program

When you run the *caesarCipher.py* program, the output looks like this:

```
guv6Jv6Jz!J6rp5r7Jzr66ntrM
```

The output is the string `'This is my secret message.'` encrypted with the Caesar cipher using a key of 13. The Caesar cipher program you just ran automatically copies this encrypted string to the clipboard so you can paste it in an email or text file. As a result, you can easily send the encrypted output from the program to another person.

You might see the following error message when you run the program:

```
Traceback (most recent call last):
  File "C:\caesarCipher.py", line 4, in <module>
    import pyperclip
ImportError: No module named pyperclip
```

If so, you probably haven't downloaded the *pyperclip.py* module into the right folder. If you confirm that *pyperclip.py* is in the folder with *caesarCipher.py* but still can't get the module to work, just comment out the code on lines 4 and 45 (which have the text `pyperclip` in them) from the *caesarCipher.py* program by placing a `#` in front of them. This makes Python ignore the code

that depends on the *pyperclip.py* module and should allow the program to run successfully. Note that if you comment out that code, the encrypted or decrypted text won't be copied to the clipboard at the end of the program. You can also comment out the `pyperclip` code from the programs in future chapters, which will remove the copy-to-clipboard functionality from those programs, too.

To decrypt the message, just paste the output text as the new value stored in the `message` variable on line 7. Then change the assignment statement on line 13 to store the string `'decrypt'` in the variable `mode`:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'decrypt' # Set to either 'encrypt' or 'decrypt'.
```

When you run the program now, the output looks like this:

```
This is my secret message.
```

Importing Modules and Setting Up Variables

Although Python includes many built-in functions, some functions exist in separate programs called modules. *Modules* are Python programs that contain additional functions that your program can use. We import modules with the appropriately named `import` statement, which consists of the `import` keyword followed by the module name.

Line 4 contains an `import` statement:

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
```

In this case, we're importing a module named `pyperclip` so we can call the `pyperclip.copy()` function later in this program. The `pyperclip.copy()` function will automatically copy strings to your computer's clipboard so you can conveniently paste them into other programs.

The next few lines in *caesarCipher.py* set three variables:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
```

```
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
```

The `message` variable stores the string to be encrypted or decrypted, and the `key` variable stores the integer of the encryption key. The `mode` variable stores either the string `'encrypt'`, which makes code later in the program encrypt the string in `message`, or `'decrypt'`, which makes the program decrypt rather than encrypt.

Constants and Variables

Constants are variables whose values shouldn't be changed when the program runs. For example, the Caesar cipher program needs a string that contains every possible character that can be encrypted with this Caesar cipher. Because that string shouldn't change, we store it in the constant variable named `SYMBOLS` in line 16:

```
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
```

Symbol is a common term used in cryptography for a single character that a cipher can encrypt or decrypt. A *symbol set* is every possible symbol a cipher is set up to encrypt or decrypt. Because we'll use the symbol set many times in this program, and because we don't want to type the full string value each time it appears in the program (we might make typos, which would cause errors), we use a constant variable to store the symbol set. We enter the code for the string value once and place it in the `SYMBOLS` constant.

Note that `SYMBOLS` is in all uppercase letters, which is the naming convention for constants. Although we *could* change `SYMBOLS` just like any other variable, the all uppercase name reminds the programmer not to write code that does so.

As with all conventions, we don't *have* to follow this one. But doing so makes it easier for other programmers to understand how these variables are used. (It can even help you when you're looking at your own code later.)

On line 19, the program stores a blank string in a variable named `translated` that will later store the encrypted or decrypted message:

```
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
```

Just as in the reverse cipher in Chapter 5, by the end of the program, the `translated` variable will contain the completely encrypted (or decrypted) message. But for now it starts as a blank string.

The for Loop Statement

At line 21, we use a type of loop called a `for` loop:

```
21. for symbol in message:
```

Recall that a `while` loop will loop as long as a certain condition is `True`. The `for` loop has a slightly different purpose and doesn't have a condition like the `while` loop. Instead, it loops over a string or a group of values. Figure 5-1 shows the six parts of a `for` loop.

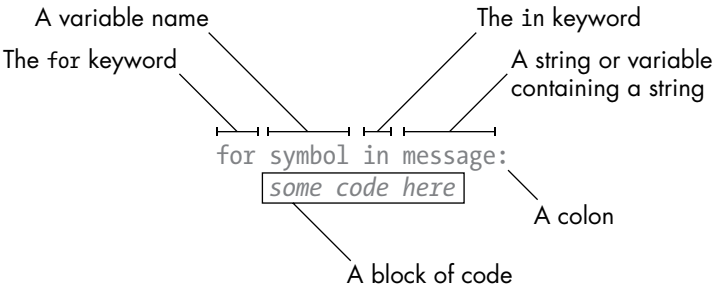


Figure 5-1: The six parts of a for loop statement

Each time the program execution goes through the loop (that is, on each iteration through the loop) the variable in the `for` statement (which in line 21 is `symbol`) takes on the value of the next character in the variable containing a string (which in this case is `message`). The `for` statement is similar to an assignment statement because the variable is created and assigned a value except the `for` statement cycles through different values to assign the variable.

An Example for Loop

For example, type the following into the interactive shell. Note that after you type the first line, the `>>>` prompt will disappear (represented in our code as `...`) because the shell is expecting a block of code after the `for` statement's colon. In the interactive shell, the block will end when you enter a blank line:

```
>>> for letter in 'Howdy':
...     print('The letter is ' + letter)
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

This code loops over each character in the string `'Howdy'`. When it does, the variable `letter` takes on the value of each character in `'Howdy'` one at a time in order. To see this in action, we've written code in the loop that prints the value of `letter` for each iteration.

A while Loop Equivalent of a for Loop

The `for` loop is very similar to the `while` loop, but when you only need to iterate over characters in a string, using a `for` loop is more efficient. You could make a `while` loop act like a `for` loop by writing a bit more code:

```
❶ >>> i = 0
❷ >>> while i < len('Howdy'):
❸ ...     letter = 'Howdy'[i]
❹ ...     print('The letter is ' + letter)
❺ ...     i = i + 1
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

Notice that this `while` loop works the same as the `for` loop but is not as short and simple as the `for` loop. First, we set a new variable `i` to `0` before the `while` statement ❶. This statement has a condition that will evaluate to `True` as long as the variable `i` is less than the length of the string `'Howdy'` ❷. Because `i` is an integer and only keeps track of the current position in the string, we need to declare a separate `letter` variable to hold the character in the string at the `i` position ❸. Then we can print the current value of `letter` to get the same output as the `for` loop ❹. When the code is finished executing, we need to increment `i` by adding `1` to it to move to the next position ❺.

To understand lines 23 and 24 in *caesarCipher.py*, you need to learn about the `if`, `elif`, and `else` statements, the `in` and `not in` operators, and the `find()` string method. We'll look at these in the following sections.

The if Statement

Line 23 in the Caesar cipher has another kind of Python instruction—the `if` statement:

```
23.     if symbol in SYMBOLS:
```

You can read an `if` statement as, "If this condition is `True`, execute the code in the following block. Otherwise, if it is `False`, skip the block." An `if` statement is formatted using the keyword `if` followed by a condition, followed by a colon (`:`). The code to execute is indented in a block just as with loops.

An Example if Statement

Let's try an example of an `if` statement. Open a new file editor window, enter the following code, and save it as *checkPw.py*:

checkPw.py

```
print('Enter your password.')
❶ typedPassword = input()
❷ if typedPassword == 'swordfish':
❸     print('Access Granted')
❹ print('Done')
```

When you run this program, it displays the text `Enter your password.` and lets the user type in a password. The password is then stored in the variable `typedPassword` ❶. Next, the `if` statement checks whether the password is equal to the string `'swordfish'` ❷. If it is, the execution moves inside the block following the `if` statement to display the text `Access Granted` to the user ❸; otherwise, if `typedPassword` isn't equal to `'swordfish'`, the execution skips the `if` statement's block. Either way, the execution continues on to the code after the `if` block to display `Done` ❹.

The else Statement

Often, we want to test a condition and execute one block of code if the condition is `True` and another block of code if it's `False`. We can use an `else` statement after an `if` statement's block, and the `else` statement's block of code will be executed if the `if` statement's condition is `False`. For an `else` statement, you just write the keyword `else` and a colon (`:`). It doesn't need a condition because it will be run if the `if` statement's condition isn't true. You can read the code as, "If this condition is `True`, execute this block, or else, if it is `False`, execute this other block."

Modify the *checkPw.py* program to look like the following (the new lines are in bold):

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
      print('Access Granted')
else:
❷     print('Access Denied')
❸ print('Done')
```

This version of the program works almost the same as the previous version. The text `Access Granted` will still display if the `if` statement's condition is `True` ❶. But now if the user types something other than `swordfish`, the `if` statement's condition will be `False`, causing the execution to enter the `else` statement's block and display `Access Denied` ❷. Either way, the execution will still continue and display `Done` ❸.

The elif Statement

Another statement, called the `elif` statement, can also be paired with `if`. Like an `if` statement, it has a condition. Like an `else` statement, it follows an `if` (or another `elif`) statement and executes if the previous `if` (or `elif`) statement's condition is `False`. You can read `if`, `elif`, and `else` statements as, "If this condition is `True`, run this block. Or else, check if this next condition is `True`. Or else, just run this last block." Any number of `elif` statements can follow an `if` statement. Modify the *checkPw.py* program again to make it look like the following:

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
❷     print('Access Granted')
❸ elif typedPassword == 'mary':
      print('Hint: the password is a fish.')
❹ elif typedPassword == '12345':
      print('That is a really obvious password.')
else:
      print('Access Denied')
print('Done')
```

This code contains four blocks for the `if`, `elif`, and `else` statements. If the user enters `12345`, then `typedPassword == 'swordfish'` evaluates to `False` ❶, so the first block with `print('Access Granted')` ❷ is skipped. The execution next checks the `typedPassword == 'mary'` condition, which also evaluates to `False` ❸, so the second block is also skipped. The `typedPassword == '12345'` condition is `True` ❹, so the execution enters the block following this `elif` statement to run the code `print('That is a really obvious password.')` and skips any remaining `elif` and `else` statements. *Notice that one and only one of these blocks will be executed.*

You can have zero or more `elif` statements following an `if` statement. You can have zero or one but not multiple `else` statements, and the `else` statement always comes last because it only executes if none of the conditions evaluate to `True`. The first statement with a `True` condition has its block executed. The rest of the conditions (even if they're also `True`) aren't checked.

The in and not in Operators

Line 23 in *caesarCipher.py* also uses the `in` operator:

```
23.     if symbol in SYMBOLS:
```

An `in` operator can connect two strings, and it will evaluate to `True` if the first string is inside the second string or evaluate to `False` if not. The `in`

operator can also be paired with `not`, which will do the opposite. Enter the following into the interactive shell:

```
>>> 'hello' in 'hello world!'
True
>>> 'hello' not in 'hello world!'
False
>>> 'ello' in 'hello world!'
True
❶ >>> 'HELLO' in 'hello world!'
False
❷ >>> '' in 'Hello'
True
```

Notice that the `in` and `not in` operators are case sensitive ❶. Also, a blank string is always considered to be in any other string ❷.

Expressions using the `in` and `not in` operators are handy to use as conditions of `if` statements to execute some code if a string exists inside another string.

Returning to *caesarCipher.py*, line 23 checks whether the string in `symbol` (which the `for` loop on line 21 set to a single character from the `message` string) is in the `SYMBOLS` string (the symbol set of all characters that can be encrypted or decrypted by this cipher program). If `symbol` is in `SYMBOLS`, the execution enters the block that follows starting on line 24. If it isn't, the execution skips this block and instead enters the block following line 39's `else` statement. The cipher program needs to run different code depending on whether the symbol is in the symbol set.

The find() String Method

Line 24 finds the index in the `SYMBOLS` string where `symbol` is:

```
24.         symbolIndex = SYMBOLS.find(symbol)
```

This code includes a method call. *Methods* are just like functions except they're attached to a value with a period (or in line 24, a variable containing a value). The name of this method is `find()`, and it's being called on the string value stored in `SYMBOLS`.

Most data types (such as strings) have methods. The `find()` method takes one string argument and returns the integer index of where the argument appears in the method's string. Enter the following into the interactive shell:

```
>>> 'hello'.find('e')
1
>>> 'hello'.find('o')
4
>>> spam = 'hello'
>>> spam.find('h')
❶ 0
```

You can use the `find()` method on either a string or a variable containing a string value. Remember that indexing in Python starts with `0`, so when the index returned by `find()` is for the first character in the string, a `0` is returned ❶.

If the string argument can't be found, the `find()` method returns the integer `-1`. Enter the following into the interactive shell:

```
>>> 'hello'.find('x')
-1
❶ >>> 'hello'.find('H')
-1
```

Notice that the `find()` method is also case sensitive ❶.

The string you pass as an argument to `find()` can be more than one character. The integer that `find()` returns will be the index of the first character where the argument is found. Enter the following into the interactive shell:

```
>>> 'hello'.find('ello')
1
>>> 'hello'.find('lo')
3
>>> 'hello hello'.find('e')
1
```

The `find()` string method is like a more specific version of using the `in` operator. It not only tells you whether a string exists in another string but also tells you where.

Encrypting and Decrypting Symbols

Now that you understand `if`, `elif`, and `else` statements; the `in` operator; and the `find()` string method, it will be easier to understand how the rest of the Caesar cipher program works.

The cipher program can only encrypt or decrypt symbols that are in the symbol set:

```
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
```

So before running the code on line 24, the program must figure out whether `symbol` is in the symbol set. Then it can find the index in `SYMBOLS` where `symbol` is located. The index returned by the `find()` call is stored in `symbolIndex`.

Now that we have the current symbol's index stored in `symbolIndex`, we can do the encryption or decryption math on it. The Caesar cipher adds the key number to the symbol's index to encrypt it or subtracts the key number

from the symbol's index to decrypt it. This value is stored in `translatedIndex` because it will be the index in `SYMBOLS` of the translated symbol.

caesarCipher.py

```
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
```

The `mode` variable contains a string that tells the program whether it should be encrypting or decrypting. If this string is `'encrypt'`, then the condition for line 27's `if` statement will be `True`, and line 28 will be executed to add the `key` to `symbolIndex` (and the block after the `elif` statement will be skipped). Otherwise, if `mode` is `'decrypt'`, then line 30 is executed to subtract the `key`.

Handling Wraparound

When we were implementing the Caesar cipher with paper and pencil in Chapter 1, sometimes adding or subtracting the key would result in a number greater than or equal to the size of the symbol set or less than zero. In those cases, we have to add or subtract the length of the symbol set so that it will "wrap around," or return to the beginning or end of the symbol set. We can use the code `len(SYMBOLS)` to do this, which returns `66`, the length of the `SYMBOLS` string. Lines 33 to 36 handle this wraparound in the cipher program.

```
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
```

If `translatedIndex` is greater than or equal to `66`, the condition on line 33 is `True` and line 34 is executed (and the `elif` statement on line 35 is skipped). Subtracting the length of `SYMBOLS` from `translatedIndex` points the index of the variable back to the beginning of the `SYMBOLS` string. Otherwise, Python will check whether `translatedIndex` is less than `0`. If that condition is `True`, line 36 is executed, and `translatedIndex` wraps around to the end of the `SYMBOLS` string.

You might be wondering why we didn't just use the integer value `66` directly instead of `len(SYMBOLS)`. By using `len(SYMBOLS)` instead of `66`, we can add to or remove symbols from `SYMBOLS` and the rest of the code will still work.

Now that you have the index of the translated symbol in `translatedIndex`, `SYMBOLS[translatedIndex]` will evaluate to the translated symbol. Line 38 adds this encrypted/decrypted symbol to the end of the `translated` string using string concatenation:

```
38.         translated = translated + SYMBOLS[translatedIndex]
```

Eventually, the `translated` string will be the whole encoded or decoded message.

Handling Symbols Outside of the Symbol Set

The `message` string might contain characters that are not in the `SYMBOLS` string. These characters are outside of the cipher program's symbol set and can't be encrypted or decrypted. Instead, they will just be appended to the `translated` string as is, which happens in lines 39 to 41:

```
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
```

The `else` statement on line 39 has four spaces of indentation. If you look at the indentation of the lines above, you'll see that it's paired with the `if` statement on line 23. Although there's a lot of code in between this `if` and `else` statement, it all belongs in the same block of code.

If line 23's `if` statement's condition were `False`, the block would be skipped, and the program execution would enter the `else` statement's block starting at line 41. This `else` block has just one line in it. It adds the unchanged `symbol` string to the end of `translated`. As a result, symbols outside of the symbol set, such as `'%'` or `'('`, are added to the translated string without being encrypted or decrypted.

Displaying and Copying the Translated String

Line 43 has no indentation, which means it's the first line after the block that started on line 21 (the `for` loop's block). By the time the program execution reaches line 44, it has looped through each character in the `message` string, encrypted (or decrypted) the characters, and added them to `translated`:

```
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

Line 44 calls the `print()` function to display the `translated` string on the screen. Notice that this is the only `print()` call in the entire program. The computer does a lot of work encrypting every letter in `message`, handling wraparound, and handling non-letter characters. But the user doesn't need to see this. The user just needs to see the final string in `translated`.

Line 45 calls `copy()`, which takes one string argument and copies it to the clipboard. Because `copy()` is a function in the `pyperclip` module, we must tell Python this by putting `pyperclip.` in front of the function name. If we type `copy(translated)` instead of `pyperclip.copy(translated)`, Python will give us an error message because it won't be able to find the function.

Python will also give an error message if you forget the `import pyperclip` line (line 4) before trying to call `pyperclip.copy()`.

That's the entire Caesar cipher program. When you run it, notice how your computer can execute the entire program and encrypt the string in less than a second. Even if you enter a very long string to store in the `message`

variable, your computer can encrypt or decrypt the message within a second or two. Compare this to the several minutes it would take to do this with a cipher wheel. The program even automatically copies the encrypted text to the clipboard so the user can simply paste it into an email to send to someone.

Encrypting Other Symbols

One problem with the Caesar cipher that we've implemented is that it can't encrypt characters outside its symbol set. For example, if you encrypt the string `'Be sure to bring the $$$.'` with the key `20`, the message will encrypt to `'VyQ?A!yQ.9Qv!381Q.2yQ$$$T'`. This encrypted message doesn't hide that you are referring to $$$. However, we can modify the program to encrypt other symbols.

By changing the string that is stored in `SYMBOLS` to include more characters, the program will encrypt them as well, because on line 23, the condition `symbol in SYMBOLS` will be `True`. The value of `symbolIndex` will be the index of `symbol` in this new, larger `SYMBOLS` constant variable. The "wraparound" will need to add or subtract the number of characters in this new string, but that's already handled because we use `len(SYMBOLS)` instead of typing `66` directly into the code (which is why we programmed it this way).

For example, you could expand line 16 to be:

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
  67890 !?.`~@#$%^&*()_+-=[]{}|;:<>,/'
```

Keep in mind that a message must be encrypted and decrypted with the same symbol set to work.

Summary

You've learned several programming concepts and read through quite a few chapters to get to this point, but now you have a program that implements a secret cipher. And more important, you understand how this code works.

Modules are Python programs that contain useful functions. To use these functions, you must first import them using an `import` statement. To call functions in an imported module, put the module name and a period before the function name, like so: *`module.function`*`()`.

Constant variables are written in uppercase letters by convention. These variables are not meant to have their values changed (although nothing prevents the programmer from writing code that does so). Constants are helpful because they give a "name" to specific values in your program.

Methods are functions that are attached to a value of a certain data type. The `find()` string method returns an integer of the position of the string argument passed to it inside the string it is called on.

You learned about several new ways to manipulate which lines of code run and how many times each line runs. A `for` loop iterates over all the characters in a string value, setting a variable to each character on each iteration. The `if`, `elif`, and `else` statements execute blocks of code based on whether a condition is `True` or `False`.

The `in` and `not in` operators check whether one string is or isn't in another string and evaluate to `True` or `False` accordingly.

Knowing how to program gives you the ability to write down a process like encrypting or decrypting with the Caesar cipher in a language that a computer can understand. And once the computer understands how to execute the process, it can do it much faster than any human can and with no mistakes (unless mistakes are in your programming). Although this is an incredibly useful skill, it turns out the Caesar cipher can easily be broken by someone who knows how to program. In Chapter 6, you'll use the skills you've learned to write a Caesar cipher hacker so you can read ciphertext that other people have encrypted. Let's move on and learn how to hack encryption.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Using *caesarCipher.py*, encrypt the following sentences with the given keys:
   a. `'"You can show black is white by argument," said Filby, "but you will never convince me."'` with key `8`
   b. `'1234567890'` with key `21`
2. Using *caesarCipher.py*, decrypt the following ciphertexts with the given keys:
   a. `'Kv?uqwpfu?rncwukdng?gpqwijB'` with key `2`
   b. `'XCBSw88S18A1S 2SB41SE .8zSEwAS5OD5A5x81V'` with key `22`
3. Which Python instruction would import a module named *watermelon.py*?
4. What do the following pieces of code display on the screen?

   a.

```
spam = 'foo'
for i in spam:
    spam = spam + i
print(spam)
```

(continued)

b.

```
if 10 < 5:
    print('Hello')
elif False:
    print('Alice')
elif 5 != 5:
    print('Bob')
else:
    print('Goodbye')
```

c.

```
print('f' not in 'foo')
```

d.

```
print('foo' in 'f')
```

e.

```
print('hello'.find('oo'))
```

6

HACKING THE CAESAR CIPHER WITH BRUTE-FORCE

"Arab scholars . . . invented cryptanalysis, the science of unscrambling a message without knowledge of the key."
—Simon Singh, The Code Book

We can hack the Caesar cipher by using a cryptanalytic technique called *brute-force*. A *brute-force attack* tries every possible decryption key for a cipher. Nothing stops a cryptanalyst from guessing one key, decrypting the ciphertext with that key, looking at the output, and then moving on to the next key if they didn't find the secret message. Because the brute-force technique is so effective against the Caesar cipher, you shouldn't actually use the Caesar cipher to encrypt secret information.

Ideally, the ciphertext would never fall into anyone's hands. But *Kerckhoffs's principle* (named after the 19th-century cryptographer Auguste Kerckhoffs) states that a cipher should still be secure even if everyone knows how the cipher works and someone else has the ciphertext. This principle was restated by the 20th-century mathematician Claude Shannon as *Shannon's maxim*: "The enemy knows the system." The part of the cipher that keeps the message secret is the key, and for the Caesar cipher this information is very easy to find.

TOPICS COVERED IN THIS CHAPTER

- Kerckhoffs's principle and Shannon's maxim
- The brute-force technique
- The `range()` function
- String formatting (string interpolation)

Source Code for the Caesar Cipher Hacker Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *caesarHacker.py*. Then download the *pyperclip.py* module if you haven't already (*https://www.nostarch.com/crackingcodes/*) and place it in the same directory (that is, the same folder) as the *caesarCipher.py* file. This module will be imported by *caesarCipher.py*.

When you're finished setting up the files, press F5 to run the program. If you run into any errors or problems with your code, you can compare it to the code in the book using the online diff tool at *https://www.nostarch.com/crackingcodes/*.

caesarHacker.py

```
 1. # Caesar Cipher Hacker
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
 6.
 7. # Loop through every possible key:
 8. for key in range(len(SYMBOLS)):
 9.     # It is important to set translated to the blank string so that the
10.     # previous iteration's value for translated is cleared:
11.     translated = ''
12.
13.     # The rest of the program is almost the same as the Caesar program:
14.
```

```
    # Loop through each symbol in message:
    for symbol in message:
        if symbol in SYMBOLS:
            symbolIndex = SYMBOLS.find(symbol)
            translatedIndex = symbolIndex - key

            # Handle the wraparound:
            if translatedIndex < 0:
                translatedIndex = translatedIndex + len(SYMBOLS)

            # Append the decrypted symbol:
            translated = translated + SYMBOLS[translatedIndex]

        else:
            # Append the symbol without encrypting/decrypting:
            translated = translated + symbol

    # Display every possible decryption:
    print('Key #%s: %s' % (key, translated))
```

Notice that much of this code is the same as the code in the original Caesar cipher program. This is because the Caesar cipher hacker program uses the same steps to decrypt the message.

Sample Run of the Caesar Cipher Hacker Program

The Caesar cipher hacker program prints the following output when you run it. It breaks the ciphertext `guv6Jv6Jz!J6rp5r7Jzr66ntrM` by decrypting the ciphertext with all 66 possible keys:

```
Key #0: guv6Jv6Jz!J6rp5r7Jzr66ntrM
Key #1: ftu5Iu5Iy I5qo4q6Iyq55msqL
Key #2: est4Ht4HxOH4pn3p5Hxp44lrpK
Key #3: drs3Gs3Gw9G3om2o4Gwo33kqoJ
Key #4: cqr2Fr2Fv8F2nl1n3Fvn22jpnI
--snip--
Key #11: Vjku?ku?o1?ugetgv?oguucigB
Key #12: Uijt!jt!nz!tfdsfu!nfttbhfA
Key #13: This is my secret message.
Key #14: SghrOhrOlxOrdbqdsOldrrZfd?
Key #15: Rfgq9gq9kw9qcapcr9kcqqYec!
--snip--
Key #61: lz1 O1 O5CO wuOw!O5w  sywR
Key #62: kyzONzON4BNOvt9v N4vOOrxvQ
Key #63: jxy9My9M3AM9us8uOM3u99qwuP
Key #64: iwx8Lx8L2.L8tr7t9L2t88pvtO
Key #65: hvw7Kw7K1?K7sq6s8K1s77ousN
```

Because the decrypted output for key 13 is plain English, we know the original encryption key must have been 13.

Setting Up Variables

The hacker program will create a `message` variable that stores the ciphertext string the program tries to decrypt. The `SYMBOLS` constant variable contains every character that the cipher can encrypt:

```
1. # Caesar Cipher Hacker
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
     67890 !?.'
```

The value for `SYMBOLS` needs to be the same as the value for `SYMBOLS` used in the Caesar cipher program that encrypted the ciphertext we're trying to hack; otherwise, the hacker program won't work. Note that there is a single space between the `0` and `!` in the string value.

Looping with the range() Function

Line 8 is a `for` loop that doesn't iterate over a string value but instead iterates over the return value from a call to the `range()` function:

```
7. # Loop through every possible key:
8. for key in range(len(SYMBOLS)):
```

The `range()` function takes one integer argument and returns a value of the `range` data type. Range values can be used in `for` loops to loop a specific number of times according to the integer you give the function. Let's try an example. Enter the following into the interactive shell:

```
>>> for i in range(3):
...   print('Hello')
...
Hello
Hello
Hello
```

The `for` loop will loop three times because we passed the integer `3` to `range()`.

More specifically, the range value returned from the `range()` function call will set the `for` loop's variable to the integers from `0` to (but not including) the argument passed to `range()`. For example, enter the following into the interactive shell:

```
>>> for i in range(6):
...   print(i)
...
0
1
```

```
2
3
4
5
```

This code sets the variable `i` to the values from `0` to (but not including) `6`, which is similar to what line 8 in *caesarHacker.py* does. Line 8 sets the `key` variable with the values from `0` to (but not including) `66`. Instead of hard-coding the value `66` directly into our program, we use the return value from `len(SYMBOLS)` so the program will still work if we modify `SYMBOLS`.

The first time the program execution goes through this loop, `key` is set to `0`, and the ciphertext in `message` is decrypted with key `0`. (Of course, if `0` is not the real key, `message` just "decrypts" to nonsense.) The code inside the `for` loop from lines 9 through 31, which we'll explain next, are similar to the original Caesar cipher program and do the decrypting. On the next iteration of line 8's `for` loop, `key` is set to `1` for the decryption.

Although we won't use it in this program, you can also pass two integer arguments to the `range()` function instead of just one. The first argument is where the range should start, and the second argument is where the range should stop (up to but not including the second argument). The arguments are separated by a comma:

```
>>> for i in range(2, 6):
...   print(i)
...
2
3
4
5
```

The variable `i` will take the value from `2` (including `2`) up to the value `6` (but not including `6`).

Decrypting the Message

The decryption code in the next few lines adds the decrypted text to the end of the string in `translated`. On line 11, `translated` is set to a blank string:

```
 7. # Loop through every possible key:
 8. for key in range(len(SYMBOLS)):
 9.     # It is important to set translated to the blank string so that the
10.     # previous iteration's value for translated is cleared:
11.     translated = ''
```

It's important that we reset `translated` to a blank string at the beginning of this `for` loop; otherwise, the text that was decrypted with the

current key will be added to the decrypted text in `translated` from the last iteration in the loop.

Lines 16 to 30 are almost the same as the code in the Caesar cipher program in Chapter 5 but are slightly simpler because this code only has to decrypt:

```
13.     # The rest of the program is almost the same as the Caesar program:
14.
15.     # Loop through each symbol in message:
16.     for symbol in message:
17.         if symbol in SYMBOLS:
18.             symbolIndex = SYMBOLS.find(symbol)
```

In line 16, we loop through every symbol in the ciphertext string stored in `message`. On each iteration of this loop, line 17 checks whether `symbol` exists in the `SYMBOLS` constant variable and, if so, decrypts it. Line 18's `find()` method call locates the index where `symbol` is in `SYMBOLS` and stores it in a variable called `symbolIndex`.

Then we subtract the `key` from `symbolIndex` on line 19 to decrypt:

```
19.             translatedIndex = symbolIndex - key
20.
21.             # Handle the wraparound:
22.             if translatedIndex < 0:
23.                 translatedIndex = translatedIndex + len(SYMBOLS)
```

This subtraction operation may cause `translatedIndex` to become less than zero and require us to "wrap around" the `SYMBOLS` constant when we find the position of the character in `SYMBOLS` to decrypt to. Line 22 checks for this case, and line 23 adds `66` (which is what `len(SYMBOLS)` returns) if `translatedIndex` is less than `0`.

Now that `translatedIndex` has been modified, `SYMBOLS[translatedIndex]` will evaluate to the decrypted symbol. Line 26 adds this symbol to the end of the string stored in `translated`:

```
25.             # Append the decrypted symbol:
26.             translated = translated + SYMBOLS[translatedIndex]
27.
28.         else:
29.             # Append the symbol without encrypting/decrypting:
30.             translated = translated + symbol
```

Line 30 just adds the unmodified `symbol` to the end of `translated` if the value was not found in the `SYMBOL` set.

Using String Formatting to Display the Key and Decrypted Messages

Although line 33 is the only `print()` function call in our Caesar cipher hacker program, it will execute several lines because it gets called once per iteration of the `for` loop in line 8:

```
32.     # Display every possible decryption:
33.     print('Key #%s: %s' % (key, translated))
```

The argument for the `print()` function call is a string value that uses *string formatting* (also called *string interpolation*). String formatting with the `%s` text places one string inside another one. The first `%s` in the string gets replaced by the first value in the parentheses at the end of the string.

Enter the following into the interactive shell:

```
>>> 'Hello %s!' % ('world')
'Hello world!'
>>> 'Hello ' + 'world' + '!'
'Hello world!'
>>> 'The %s ate the %s that ate the %s.' % ('dog', 'cat', 'rat')
'The dog ate the cat that ate the rat.'
```

In this example, first the string `'world'` is inserted into the string `'Hello %s!'` in place of the `%s`. It works as though you had concatenated the part of the string before the `%s` with the interpolated string and the part of the string after the `%s`. When you interpolate multiple strings, they replace each `%s` in order.

String formatting is often easier to type than string concatenation using the `+` operator, especially for large strings. And, unlike with string concatenation, you can insert non-string values such as integers into the string. Enter the following into the interactive shell:

```
>>> '%s had %s pies.' % ('Alice', 42)
'Alice had 42 pies.'
>>> 'Alice' + ' had ' + 42 + ' pies.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The integer `42` is inserted into the string without any issues when you use interpolation, but when you try to concatenate the integer, it causes an error.

Line 33 of *caesarHacker.py* uses string formatting to create a string that has the values in both the `key` and `translated` variables. Because `key` stores an integer value, we use string formatting to put it in a string value that is passed to `print()`.

Summary

The critical weakness of the Caesar cipher is that there aren't many possible keys that can be used to encrypt. Any computer can easily decrypt with all 66 possible keys, and it takes a cryptanalyst only a few seconds to look through the decrypted messages to find the one in English. To make our messages more secure, we need a cipher that has more potential keys. The transposition cipher discussed in Chapter 7 can provide this security for us.

PRACTICE QUESTION

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Break the following ciphertext, decrypting one line at a time because each line has a different key. Remember to escape any quote characters:

```
qeFIP?eGSeECNNS,
5coOMXXcoPSZIWoQI,
avnl1olyD4l'ylDohww6DhzDjhuDil,

z.GM?.cEQc. 7Oc.7KcKMKHA9AGFK,
?MFYp2pPJJUpZSIJWpRdpMFY,
ZqH8sl5HtqHTH4s3lyvH5zH5spH4t pHzqHlH3l5K

Zfbi,!tif!xpvme!qspcbcmz!fbu!nfA
```

7

ENCRYPTING WITH THE TRANSPOSITION CIPHER

"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."
—Edward Snowden, 2015

The Caesar cipher isn't secure; it doesn't take much for a computer to brute-force through all 66 possible keys. The transposition cipher, on the other hand, is more difficult to brute-force because the number of possible keys depends on the message's length. There are many different types of transposition ciphers, including the rail fence cipher, route cipher, Myszkowski transposition cipher, and disrupted transposition cipher. This chapter covers a simple transposition cipher called the *columnar transposition cipher*.

TOPICS COVERED IN THIS CHAPTER

- Creating functions with `def` statements
- Arguments and parameters
- Variables in global and local scopes
- `main()` functions
- The list data type
- Similarities in lists and strings
- Lists of lists
- Augmented assignment operators (`+=`, `-=`, `*=`, `/=`)
- The `join()` string method
- Return values and the `return` statement
- The `__name__` variable

How the Transposition Cipher Works

Instead of substituting characters with other characters, the transposition cipher rearranges the message's symbols into an order that makes the original message unreadable. Because each key creates a different ordering, or *permutation*, of the characters, a cryptanalyst doesn't know how to rearrange the ciphertext back into the original message.

The steps for encrypting with the transposition cipher are as follows:

1. Count the number of characters in the message and the key.
2. Draw a row of a number of boxes equal to the key (for example, 8 boxes for a key of 8).
3. Start filling in the boxes from left to right, entering one character per box.
4. When you run out of boxes but still have more characters, add another row of boxes.
5. When you reach the last character, shade in the unused boxes in the last row.
6. Starting from the top left and going down each column, write out the characters. When you get to the bottom of a column, move to the next column to the right. Skip any shaded boxes. This will be the ciphertext.

To see how these steps work in practice, we'll encrypt a message by hand and then translate the process into a program.

Encrypting a Message by Hand

Before we start writing code, let's encrypt the message "Common sense is not so common." using pencil and paper. Including the spaces and punctuation, this message has 30 characters. For this example, you'll use the number 8 as the key. The range for possible keys for this cipher type is from 2 to half the message size, which is 15. But the longer the message, the more keys are possible. Encrypting an entire book using the columnar transposition cipher would allow for thousands of possible keys.

The first step is to draw eight boxes in a row to match the key number, as shown in Figure 7-1.

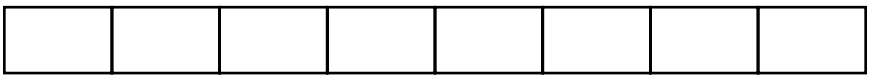


Figure 7-1: The number of boxes in the first row should match the key number.

The second step is to start writing the message you want to encrypt into the boxes, placing one character into each box, as shown in Figure 7-2. Remember that spaces are also characters (indicated here with ■).

C	o	m	m	o	n	■	s

Figure 7-2: Fill in one character per box, including spaces.

You have only eight boxes, but there are 30 characters in the message. When you run out of boxes, draw another row of eight boxes under the first row. Continue creating new rows until you've written the entire message, as shown in Figure 7-3.

1st	2nd	3rd	4th	5th	6th	7th	8th
C	o	m	m	o	n	■	s
e	n	s	e	■	i	s	■
n	o	t	■	s	o	■	c
o	m	m	o	n	.		

Figure 7-3: Add more rows until the entire message is filled in.

Shade in the two boxes in the last row as a reminder to ignore them. The ciphertext consists of the letters read from the top-left box going down the column. *C*, *e*, *n*, and *o* are from the 1st column, as labeled in the diagram. When you get to the last row of a column, move to the top row of the next column to the right. The next characters are *o*, *n*, *o*, *m*. Ignore the shaded boxes.

The ciphertext is "Cenoonommstmme oo snnio. s s c", which is sufficiently scrambled to prevent someone from figuring out the original message by looking at it.

Creating the Encryption Program

To make a program for encrypting, you need to translate these paper-and-pencil steps into Python code. Let's look again at how to encrypt the string `'Common sense is not so common.'` using the key `8`. To Python, a character's position inside a string is its numbered index, so add the index of each letter in the string to the boxes in your original encrypting diagram, as shown in Figure 7-4. (Remember that indexes begin with `0`, not `1`.)

1st	2nd	3rd	4th	5th	6th	7th	8th
C 0	o 1	m 2	m 3	o 4	n 5	■ 6	s 7
e 8	n 9	s 10	e 11	■ 12	i 13	s 14	■ 15
n 16	o 17	t 18	■ 19	s 20	o 21	■ 22	c 23
o 24	m 25	m 26	o 27	n 28	. 29		

Figure 7-4: Add the index number to each box, starting with 0.

These boxes show that the first column has the characters at indexes `0`, `8`, `16`, and `24` (which are `'C'`, `'e'`, `'n'`, and `'o'`). The next column has the characters at indexes `1`, `9`, `17`, and `25` (which are `'o'`, `'n'`, `'o'`, and `'m'`). Notice the pattern emerging: the nth column has all the characters in the string at indexes $0 + (n - 1)$, $8 + (n - 1)$, $16 + (n - 1)$, and $24 + (n - 1)$, as shown in Figure 7-5.

1st	2nd	3rd	4th	5th	6th	7th	8th
C 0+0=0	o 1+0=1	m 2+0=2	m 3+0=3	o 4+0=4	n 5+0=5	■ 6+0=6	s 7+0=7
e 0+8=8	n 1+8=9	s 2+8=10	e 3+8=11	■ 4+8=12	i 5+8=13	s 6+8=14	■ 7+8=15
n 0+16=16	o 1+16=17	t 2+16=18	■ 3+16=19	s 4+16=20	o 5+16=21	■ 6+16=22	c 7+16=23
o 0+24=24	m 1+24=25	m 2+24=26	o 3+24=27	n 4+24=28	. 5+24=29		

Figure 7-5: The index of each box follows a predictable pattern.

There is an exception for the last row in the 7th and 8th columns, because 24 + (7 – 1) and 24 + (8 – 1) would be greater than 29, which is the largest index in the string. In those cases, you only add 0, 8, and 16 to *n* (and skip 24).

What's so special about the numbers 0, 8, 16, and 24? These are the numbers you get when, starting from 0, you add the key (which in this example is 8). So, 0 + 8 is 8, 8 + 8 is 16, 16 + 8 is 24. The result of 24 + 8 would be 32, but because 32 is larger than the length of the message, you'll stop at 24.

For the *n*th column's string, start at index (*n* – 1) and continue adding 8 (the key) to get the next index. Keep adding 8 as long as the index is less than 30 (the message length), at which point move to the next column.

If you imagine each column is a string, the result would be a list of eight strings, like this: `'Ceno'`, `'onom'`, `'mstm'`, `'me o'`, `'o sn'`, `'nio.'`, `' s '`, `'s c'`. If you concatenated the strings together in order, the result would be the ciphertext: `'Cenoonommstmme oo snnio. s s c'`. You'll learn about a concept called *lists* later in the chapter that will let you do exactly this.

Source Code for the Transposition Cipher Encryption Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and then save it as *transpositionEncrypt.py*. Remember to place the *pyperclip.py* module in the same directory as the *transpositionEncrypt.py* file. Then press F5 to run the program.

transpositionEncrypt.py

```
1. # Transposition Cipher Encryption
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
5.
6. def main():
7.     myMessage = 'Common sense is not so common.'
8.     myKey = 8
9.
10.    ciphertext = encryptMessage(myKey, myMessage)
11.
12.    # Print the encrypted string in ciphertext to the screen, with
13.    # a | ("pipe" character) after it in case there are spaces at
14.    # the end of the encrypted message:
15.    print(ciphertext + '|')
16.
17.    # Copy the encrypted string in ciphertext to the clipboard:
18.    pyperclip.copy(ciphertext)
19.
20.
21. def encryptMessage(key, message):
22.    # Each string in ciphertext represents a column in the grid:
23.    ciphertext = [''] * key
24.
```

```
    # Loop through each column in ciphertext:
    for column in range(key):
        currentIndex = column

        # Keep looping until currentIndex goes past the message length:
        while currentIndex < len(message):
            # Place the character at currentIndex in message at the
            # end of the current column in the ciphertext list:
            ciphertext[column] += message[currentIndex]

            # Move currentIndex over:
            currentIndex += key

    # Convert the ciphertext list into a single string value and return it:
    return ''.join(ciphertext)


# If transpositionEncrypt.py is run (instead of imported as a module) call
# the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Transposition Cipher Encryption Program

When you run the *transpositionEncrypt.py* program, it produces this output:

```
Cenoonommstmme oo snnio. s s c|
```

The vertical pipe character (|) marks the end of the ciphertext in case there are spaces at the end. This ciphertext (without the pipe character at the end) is also copied to the clipboard, so you can paste it into an email to someone. If you want to encrypt a different message or use a different key, change the value assigned to the `myMessage` and `myKey` variables on lines 7 and 8. Then run the program again.

Creating Your Own Functions with def Statements

After importing the `pyperclip` module, you'll use a `def` statement to create a custom function, `main()`, on line 6.

```
1. # Transposition Cipher Encryption
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip
6. def main():
7.     myMessage = 'Common sense is not so common.'
8.     myKey = 8
```

The `def` statement means you're creating, or *defining*, a new function that you can call later in the program. The block of code after the `def`

statement is the code that will run when the function is called. When you *call* this function, the execution moves inside the block of code following the function's `def` statement.

As you learned in Chapter 3, in some cases, functions will accept *arguments*, which are values that the function can use with its code. For example, `print()` can take a string value as an argument between its parentheses. When you define a function that takes arguments, you put a variable name between its parentheses in its `def` statement. These variables are called *parameters*. The `main()` function defined here has no parameters, so it takes no arguments when it's called. If you try to call a function with too many or too few arguments for the number of parameters the function has, Python will raise an error message.

Defining a Function that Takes Arguments with Parameters

Let's create a function with a parameter and then call it with an argument. Open a new file editor window and enter the following code into it:

hello Function.py

```
❶ def hello(name):
❷     print('Hello, ' + name)
❸ print('Start.')
❹ hello('Alice')
❺ print('Call the function again:')
❻ hello('Bob')
❼ print('Done.')
```

Save this program as *helloFunction.py* and run it by pressing F5. The output looks like this:

```
Start.
Hello, Alice
Call the function again:
Hello, Bob
Done.
```

When the *helloFunction.py* program runs, the execution starts at the top. The `def` statement ❶ defines the `hello()` function with one parameter, which is the variable `name`. The execution skips the block after the `def` statement ❷ because the block is only run when the function is called. Next, it executes `print('Start.')` ❸, which is why `'Start.'` is the first string printed when you run the program.

The next line after `print('Start.')` is the first function call to `hello()`. The program execution jumps to the first line in the `hello()` function's block ❷. The string `'Alice'` is passed as the argument and is assigned to the parameter `name`. This function call prints the string `'Hello, Alice'` to the screen.

When the program execution reaches the bottom of the `def` statement's block, the execution jumps back to the line with the function call ❹ and continues executing the code from there, so `'Call the function again:'` is printed ❺.

Next is a second call to `hello()` ❻. The program execution jumps back to the `hello()` function's definition ❷ and executes the code there again, displaying `'Hello, Bob'` on the screen. Then the function returns and the execution goes to the next line, which is the `print('Done.')` statement ❼, and executes it. This is the last line in the program, so the program exits.

Changes to Parameters Exist Only Inside the Function

Enter the following code into the interactive shell. This code defines and then calls a function named `func()`. Note that the interactive shell requires you to enter a blank line after `param = 42` to close the `def` statement's block:

```
>>> def func(param):
        param = 42

>>> spam = 'Hello'
>>> func(spam)
>>> print(spam)
Hello
```

The `func()` function takes a parameter called `param` and sets its value to `42`. The code outside the function creates a `spam` variable and sets it to a string value, and then the function is called on `spam` and `spam` is printed.

When you run this program, the `print()` call on the last line will print `'Hello'`, not `42`. When `func()` is called with `spam` as the argument, only the value inside `spam` is being copied and assigned to `param`. Any changes made to `param` inside the function will *not* change the value in the `spam` variable. (There is an exception to this rule when you are passing a list or dictionary value, but this is explained in "List Variables Use References" on page 119.)

Every time a function is called, a *local scope* is created. Variables created during a function call exist in this local scope and are called *local variables*. Parameters always exist in a local scope (they are created and assigned a value when the function is called). Think of a *scope* as a container the variables exist inside. When the function returns, the local scope is destroyed, and the local variables that were contained in the scope are forgotten.

Variables created outside of every function exist in the *global scope* and are called *global variables*. When the program exits, the global scope is destroyed, and all the variables in the program are forgotten. (All the variables in the reverse cipher and Caesar cipher programs in Chapters 5 and 6, respectively, were global.)

A variable must be local or global; it cannot be both. Two different variables can have the same name as long as they're in different scopes. They're still considered two different variables, similar to how Main Street in San Francisco is a different street from Main Street in Birmingham.

The important idea to understand is that the argument value that is "passed" into a function call is *copied* to the parameter. So even if the parameter is changed, the variable that provided the argument value is not changed.

Defining the main() Function

In lines 6 through 8 in *transpositionEncrypt.py*, you can see that we've defined a `main()` function that will set values for the variables `myMessage` and `myKey` when called:

```
6. def main():
7.     myMessage = 'Common sense is not so common.'
8.     myKey = 8
```

The rest of the programs in this book will also have a function named `main()` that is called at the start of each program. The reason we have a `main()` function is explained at the end of this chapter, but for now just know that `main()` is always called soon after the programs in this book are run.

Lines 7 and 8 are the first two lines in the block of code defining `main()`. In these lines, the variables `myMessage` and `myKey` store the plaintext message to encrypt and the key used to do the encryption. Line 9 is a blank line but is still part of the block and separates lines 7 and 8 from line 10 to make the code more readable. Line 10 assigns the variable `ciphertext` as the encrypted message by calling a function that takes two arguments:

```
10.     ciphertext = encryptMessage(myKey, myMessage)
```

The code that does the actual encrypting is in the `encryptMessage()` function defined later on line 21. This function takes two arguments: an integer value for the key and a string value for the message to encrypt. In this case, we pass the variables `myMessage` and `myKey`, which we just defined in lines 7 and 8. When passing multiple arguments to a function call, separate the arguments with a comma.

The return value of `encryptMessage()` is a string value of the encrypted ciphertext. This string is stored in `ciphertext`.

The ciphertext message is printed to the screen on line 15 and copied to the clipboard on line 18:

```
12.     # Print the encrypted string in ciphertext to the screen, with
13.     # a | ("pipe" character) after it in case there are spaces at
14.     # the end of the encrypted message:
15.     print(ciphertext + '|')
16.
17.     # Copy the encrypted string in ciphertext to the clipboard:
18.     pyperclip.copy(ciphertext)
```

The program prints a pipe character (`|`) at the end of the message so the user can see any empty space characters at the end of the ciphertext.

Line 18 is the last line of the `main()` function. After it executes, the program execution returns to the line after the line that called it.

Passing the Key and Message As Arguments

The `key` and `message` variables between the parentheses on line 21 are parameters:

```
21. def encryptMessage(key, message):
```

When the `encryptMessage()` function is called in line 10, two argument values are passed (the values in `myKey` and `myMessage`). These values get assigned to the parameters `key` and `message` when the execution moves to the top of the function.

You might wonder why you even have the `key` and `message` parameters, since you already have the variables `myKey` and `myMessage` in the `main()` function. We need different variables because `myKey` and `myMessage` are in the `main()` function's local scope and can't be used outside of `main()`.

The List Data Type

Line 23 in the *transpositionEncrypt.py* program uses a data type called a *list*:

```
22.     # Each string in ciphertext represents a column in the grid:
23.     ciphertext = [''] * key
```

Before we can move on, you need to understand how lists work and what you can do with them. A list value can contain other values. Similar to how strings begin and end with quotes, a list value begins with an open bracket, `[`, and ends with a closed bracket, `]`. The values stored inside the list are between the brackets. If more than one value is in the list, the values are separated by commas.

To see a list in action, enter the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals
['aardvark', 'anteater', 'antelope', 'albert']
```

The `animals` variable stores a list value, and in this list value are four string values. The individual values inside a list are also called *items* or *elements*. Lists are ideal to use when you have to store multiple values in one variable.

Many of the operations you can do with strings also work with lists. For example, indexing and slicing work on list values the same way they work on string values. Instead of individual characters in a string, the index refers to an item in a list. Enter the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'albert']
❶ >>> animals[0]
'aardvark'
>>> animals[1]
'anteater'
```

```
>>> animals[2]
'albert'
❷ >>> animals[1:3]
['anteater', 'albert']
```

Keep in mind that the first index is `0`, not `1` ❶. Similar to how using slices with a string gives you a new string that is part of the original string, using slices with a list gives you a list that is part of the original list. And remember that if a slice has a second index, the slice only goes *up to but doesn't include* the item at the second index ❷.

A `for` loop can also iterate over the values in a list, just like it can iterate over the characters in a string. The value stored in the `for` loop's variable is a single value from the list. Enter the following into the interactive shell:

```
>>> for spam in ['aardvark', 'anteater', 'albert']:
...   print('For dinner we are cooking ' + spam)
...
For dinner we are cooking aardvark
For dinner we are cooking anteater
For dinner we are cooking albert
```

Each time the loop iterates, the `spam` variable is assigned a new value from the list starting with the list's `0` index until the end of the list.

Reassigning the Items in Lists

You can also modify the items inside a list by using the list's index with a normal assignment statement. Enter the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'albert']
❶ >>> animals[2] = 9999
>>> animals
❷ ['aardvark', 'anteater', 9999]
```

To modify the third member of the `animals` list, we use the index to get the third value with `animals[2]` and then use an assignment statement to change its value from `'albert'` to the value `9999` ❶. When we check the contents of the list again, `'albert'` is no longer contained in the list ❷.

REASSIGNING CHARACTERS IN STRINGS

Although you can reassign items in a list, you can't reassign a character in a string value. Enter the following code into the interactive shell:

```
>>> 'Hello world!'[6] = 'X'
```

(continued)

You'll see the following error:

```
Traceback (most recent call last):
  File <pyshell#0>, line 1, in <module>
    'Hello world!'[6] = 'X'
TypeError: 'str' object does not support item assignment
```

The reason you see this error is that Python doesn't let you use assignment statements on a string's index value. Instead, to change a character in a string, you need to create a new string using slices. Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam = spam[:6] + 'X' + spam[7:]
>>> spam
'Hello Xorld!'
```

You would first take a slice that starts at the beginning of the string and goes up to the character to change. Then you could concatenate that to the string of the new character and a slice from the character after the new character to the end of the string. This results in the original string with just one changed character.

Lists of Lists

List values can even contain other lists. Enter the following into the interactive shell:

```
>>> spam = [['dog', 'cat'], [1, 2, 3]]
>>> spam[0]
['dog', 'cat']
>>> spam[0][0]
'dog'
>>> spam[0][1]
'cat'
>>> spam[1][0]
1
>>> spam[1][1]
2
```

The value of `spam[0]` evaluates to the list `['dog', 'cat']`, which has its own indexes. The double index brackets used for `spam[0][0]` indicates that we're taking the first item from the first list: `spam[0]` evaluates to `['dog', 'cat']` and `['dog', 'cat'][0]` evaluates to `'dog'`.

Using len() and the in Operator with Lists

You've used `len()` to indicate the number of characters in a string (that is, the length of the string). The `len()` function also works on list values and returns an integer of the number of items in a list.

Enter the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> len(animals)
4
```

Similarly, you've used the `in` and `not in` operators to indicate whether a string exists inside another string value. The `in` operator also works for checking whether a value exists in a list, and the `not in` operator checks whether a value does not exist in a list. Enter the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'anteater' in animals
True
>>> 'anteater' not in animals
False
❶ >>> 'anteat' in animals
False
❷ >>> 'anteat' in animals[1]
True
>>> 'delicious spam' in animals
False
```

Why does the expression at ❶ return `False` while the expression at ❷ returns `True`? Remember that `animals` is a list value, while `animals[1]` evaluates to the string value `'anteater'`. The expression at ❶ evaluates to `False` because the string `'anteat'` does not exist in the `animals` list. However, the expression at ❷ evaluates to `True` because `animals[1]` is the string `'anteater'` and `'anteat'` exists within that string.

Similar to how a set of empty quotes represents a blank string value, a set of empty brackets represents a blank list. Enter the following into the interactive shell:

```
>>> animals = []
>>> len(animals)
0
```

The `animals` list is empty, so its length is `0`.

List Concatenation and Replication with the + and * Operators

You know that the `+` and `*` operators can concatenate and replicate strings; the same operators can also concatenate and replicate lists. Enter the following into the interactive shell.

```
>>> ['hello'] + ['world']
['hello', 'world']
>>> ['hello'] * 5
['hello', 'hello', 'hello', 'hello', 'hello']
```

That's enough about the similarities between strings and lists. Just remember that most operations you can do with string values also work with list values.

The Transposition Encryption Algorithm

We'll use lists in our encryption algorithm to create our ciphertext. Let's return to the code in the *transpositionEncrypt.py* program. In line 23, which we saw earlier, the `ciphertext` variable is a list of empty string values:

```
22.     # Each string in ciphertext represents a column in the grid:
23.     ciphertext = [''] * key
```

Each string in the `ciphertext` variable represents a column of the transposition cipher's grid. Because the number of columns is equal to the key, you can use list replication to multiply a list with one blank string value in it by the value in `key`. This is how line 23 evaluates to a list with the correct number of blank strings. The string values will be assigned all the characters that go into one column of the grid. The result will be a list of string values that represent each column, as discussed earlier in the chapter. Because list indexes start with 0, you'll need to also label each column starting at 0. So `ciphertext[0]` is the leftmost column, `ciphertext[1]` is the column to the right of that, and so on.

To see how this would work, let's again look at the grid from the "Common sense is not so common." example from earlier in this chapter (with column numbers corresponding to the list indexes added to the top), as shown in Figure 7-6.

0	1	2	3	4	5	6	7
C	o	m	m	o	n	■	s
e	n	s	e	■	i	s	■
n	o	t	■	s	o	■	c
o	m	m	o	n	.		

Figure 7-6: The example message grid with list indexes for each column

If we manually assigned the string values to the ciphertext variable for this grid, it would look like this:

```
>>> ciphertext = ['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']
>>> ciphertext[0]
'Ceno'
```

The next step adds text to each string in ciphertext, as we just did in the manual example, except this time we added some code to make the computer do it programmatically:

```
25.     # Loop through each column in ciphertext:
26.     for column in range(key):
27.         currentIndex = column
```

The for loop on line 26 iterates once for each column, and the column variable has the integer value to use for the index to ciphertext. On the first iteration through the for loop, the column variable is set to 0; on the second iteration, it's set to 1; then 2; and so on. We have the index for the string values in ciphertext that we want to access later using the expression ciphertext[column].

Meanwhile, the currentIndex variable holds the index for the message string the program looks at on each iteration of the for loop. On each iteration through the loop, line 27 sets currentIndex to the same value as column. Next, we'll create the ciphertext by concatenating the scrambled message together one character at a time.

Augmented Assignment Operators

So far, when we've concatenated or added values to each other, we've used the + operator to add the new value to the variable. Often, when you're assigning a new value to a variable, you want it to be based on the variable's current value, so you use the variable as the part of the expression that is evaluated and assigned to the variable, as in this example in the interactive shell:

```
>>> spam = 40
>>> spam = spam + 2
>>> print(spam)
42
```

There are other ways to manipulate values in variables based on the variable's current value. For example, you can do this by using *augmented assignment operators*. The statement spam += 2, which uses the += augmented assignment operator, does the *same thing* as spam = spam + 2. It's just a little shorter to type. The += operator works with integers to do addition, strings to do string concatenation, and lists to do list concatenation. Table 7-1 shows the augmented assignment operators and their equivalent assignment statements.

Table 7-1: Augmented Assignment Operators

Augmented assignment	Equivalent normal assignment
`spam += 42`	`spam = spam + 42`
`spam -= 42`	`spam = spam - 42`
`spam *= 42`	`spam = spam * 42`
`spam /= 42`	`spam = spam / 42`

We'll use augmented assignment operators to concatenate characters to our ciphertext.

Moving currentIndex Through the Message

The `currentIndex` variable holds the index of the next character in the `message` string that will be concatenated to the `ciphertext` lists. The `key` is added to `currentIndex` on each iteration of line 30's `while` loop to point to different characters in `message` and, at each iteration of line 26's `for` loop, `currentIndex` is set to the value in the `column` variable.

To scramble the string in the `message` variable, we need to take the first character of `message`, which is `'C'`, and put it into the first string of `ciphertext`. Then, we would skip eight characters into `message` (because `key` is equal to `8`) and concatenate that character, which is `'e'`, to the first string of the ciphertext. We would continue to skip characters according to the key and concatenate each character until we reach the end of the message. Doing so would create the string `'Ceno'`, which is the first column of the ciphertext. Then we would do this again but start at the second character in `message` to make the second column.

Inside the `for` loop that starts on line 26 is a `while` loop that starts on line 30. This `while` loop finds and concatenates the right character in `message` to make each column. It loops while `currentIndex` is less than the length of `message`:

```
29.         # Keep looping until currentIndex goes past the message length:
30.         while currentIndex < len(message):
31.             # Place the character at currentIndex in message at the
32.             # end of the current column in the ciphertext list:
33.             ciphertext[column] += message[currentIndex]
34.
35.             # Move currentIndex over:
36.             currentIndex += key
```

For each column, the `while` loop iterates through the original `message` variable and picks out characters in intervals of `key` by adding `key` to `currentIndex`. On line 27 for the first iteration of the `for` loop, `currentIndex` was set to the value of `column`, which starts at `0`.

As you can see in Figure 7-7, `message[currentIndex]` is the first character of `message` on its first iteration. The character at `message[currentIndex]`

is concatenated to `ciphertext[column]` to start the first column at line 33. Line 36 adds the value in `key` (which is `8`) to `currentIndex` each time through the loop. The first time it is `message[0]`, the second time `message[8]`, the third time `message[16]`, and the fourth time `message[24]`.

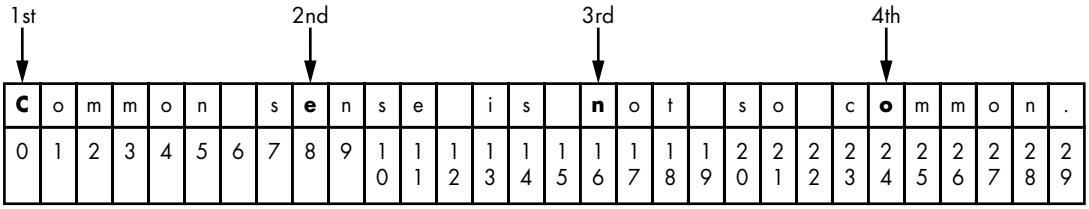


Figure 7-7: Arrows pointing to what `message[currentIndex]` refers to during the first iteration of the for loop when `column` is set to `0`

Although the value in `currentIndex` is less than the length of the `message` string, you want to continue concatenating the character at `message[currentIndex]` to the end of the string at the `column` index in `ciphertext`. When `currentIndex` is greater than the length of `message`, the execution exits the `while` loop and goes back to the `for` loop. Because there isn't code in the `for` block after the `while` loop, the `for` loop iterates, `column` is set to `1`, and `currentIndex` starts at the same value as `column`.

Now when line 36 adds `8` to `currentIndex` on each iteration of line 30's `while` loop, the indexes will be `1`, `9`, `17`, and `25`, as shown in Figure 7-8.

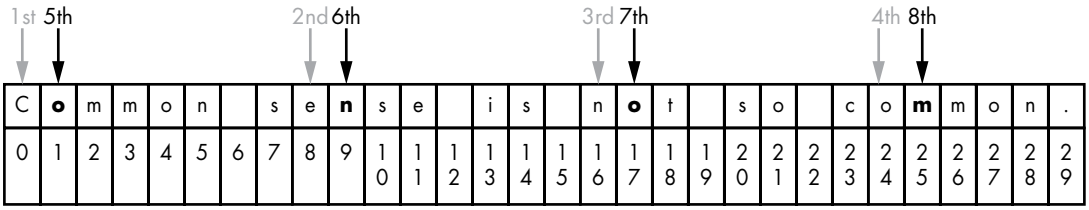


Figure 7-8: Arrows pointing to what `message[currentIndex]` refers to during the second iteration of the for loop when `column` is set to `1`

As `message[1]`, `message[9]`, `message[17]`, and `message[25]` are concatenated to the end of `ciphertext[1]`, they form the string `'onom'`. This is the second column of the grid.

When the `for` loop has finished looping through the rest of the columns, the value in `ciphertext` is `['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']`. After we have the list of string columns, we need to join them together to make one string that is the whole ciphertext: `'Cenoonommstmme oo snnio. s s c'`.

The join() String Method

The `join()` method is used on line 39 to join the individual column strings of `ciphertext` into one string. The `join()` method is called on a string value and takes a list of strings. It returns one string that has all of the members

in the list joined by the string that `join()` is called on. (This is a blank string if you just want to join the strings together.) Enter the following into the interactive shell:

```
>>> eggs = ['dogs', 'cats', 'moose']
❶ >>> ''.join(eggs)
'dogscatsmoose'
❷ >>> ', '.join(eggs)
'dogs, cats, moose'
❸ >>> 'XYZ'.join(eggs)
'dogsXYZcatsXYZmoose'
```

When you call `join()` on an empty string and join the list `eggs` ❶, you get the list's strings concatenated with no string between them. In some cases, you might want to separate each member in a list to make it more readable, which we've done at ❷ by calling `join()` on the string `', '`. This inserts the string `', '` between each member of the list. You can insert any string you want between list members, as you can see at ❸.

Return Values and return Statements

A function (or method) call always evaluates to a value. This is the value *returned* by the function or method call, also called the *return value* of the function. When you create your own functions using a `def` statement, a `return` statement tells Python what the return value for the function is. Line 39 is a `return` statement:

```
38.     # Convert the ciphertext list into a single string value and return it:
39.     return ''.join(ciphertext)
```

Line 39 calls `join()` on the blank string and passes `ciphertext` as the argument so the strings in the `ciphertext` list are joined into a single string.

A return Statement Example

A `return` statement is the `return` keyword followed by the value to be returned. You can use an expression instead of a value, as in line 39. When you do so, the return value is whatever that expression evaluates to. Open a new file editor window, enter the following program, save it as *addNumbers.py*, and then press F5 to run it:

addNumbers.py

```
1. def addNumbers(a, b):
2.     return a + b
3.
4. print(addNumbers(2, 40))
```

When you run the *addNumbers.py* program, this is the output:

```
42
```

That's because the function call `addNumbers(2, 40)` at line 4 evaluates to `42`. The `return` statement in `addNumbers()` at line 2 evaluates the expression `a + b` and then returns the evaluated value.

Returning the Encrypted Ciphertext

In the *transpositionEncrypt.py* program, the `encryptMessage()` function's `return` statement returns a string value that is created by joining all of the strings in the `ciphertext` list. If the list in ciphertext is `['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']`, the `join()` method call will return `'Cenoonommstmme oo snnio. s s c'`. This final string, the result of the encryption code, is returned by our `encryptMessage()` function.

The great advantage of using functions is that a programmer has to know what the function does but doesn't need to know how the function's code works. A programmer can understand that when they call the `encryptMessage()` function and pass it an integer as well as a string for the `key` and `message` parameters, the function call evaluates to an encrypted string. They don't need to know anything about how the code in `encryptMessage()` actually does this, which is similar to how you know that when you pass a string to `print()`, it will print the string even though you've never seen the `print()` function's code.

The __name__ Variable

You can turn the transposition encryption program into a module using a special trick involving the `main()` function and a variable named `__name__`.

When you run a Python program, `__name__` (that's two underscores before `name` and two underscores after) is assigned the string value `'__main__'` (again, two underscores before and after `main`) even before the first line of your program runs. The double underscore is often referred to as *dunder* in Python, and `__main__` is called *dunder main dunder*.

At the end of the script file (and, more important, after all of the `def` statements), you want to have some code that checks whether the `__name__` variable has the `'__main__'` string assigned to it. If so, you want to call the `main()` function.

The `if` statement on line 44 is actually one of the first lines of code executed when you run the program (after the `import` statement on line 4 and the `def` statements on lines 6 and 21).

```
42. # If transpositionEncrypt.py is run (instead of imported as a module) call
43. # the main() function:
44. if __name__ == '__main__':
45.     main()
```

The reason the code is set up this way is that although Python sets `__name__` to `'__main__'` when the program is run, it sets it to the string `'transpositionEncrypt'` if the program is imported by another Python program. Similar to how the program imports the `pyperclip` module to call the

functions in it, other programs might want to import *transpositionEncrypt.py* to call its `encryptMessage()` function without the `main()` function running. When an `import` statement is executed, Python looks for the module's file by adding *.py* to the end of the filename (which is why `import pyperclip` imports the *pyperclip.py* file). This is how our program knows whether it's being run as the main program or imported by a different program as a module. (You'll import *transpositionEncrypt.py* as a module in Chapter 9.)

When you import a Python program and before the program is executed, the `__name__` variable is set to the filename part before *.py*. When the *transpositionEncrypt.py* program is imported, all the `def` statements are run (to define the `encryptMessage()` function that the importing program wants to use), but the `main()` function isn't called, so the encryption code for `'Common sense is not so common.'` with key `8` isn't executed.

That's why the code that encrypts the `myMessage` string with the `myKey` key is inside a function (which by convention is named `main()`). This code inside `main()` won't run when *transpositionEncrypt.py* is imported by other programs, but these other programs can still call its `encryptMessage()` function. This is how the function's code can be reused by other programs.

NOTE *One useful way of learning how a program works is by following its execution step-by-step as it runs. You can use an online program-tracing tool to view traces of the Hello Function and Transposition Cipher Encryption programs at* https://www.nostarch.com/crackingcodes/. *The tracing tool will give you a visual representation of what the programs are doing as each line of code is executed.*

Summary

Whew! You learned several new programming concepts in this chapter. The transposition cipher program is more complicated (but much more secure) than the Caesar cipher program in Chapter 6. The new concepts, functions, data types, and operators you've learned in this chapter let you manipulate data in more sophisticated ways. Just remember that much of what goes into understanding a line of code is evaluating it step-by-step in the same way Python will.

You can organize code into groups called functions, which you create with `def` statements. Argument values can be passed to functions as the function's parameters. Parameters are local variables. Variables outside of all functions are global variables. Local variables are different from global variables, even if they have the same name as the global variable. Local variables in one function are also separate from local variables in another function, even if they have the same name.

List values can store multiple other values, including other list values. Many of the operations you can use on strings (such as indexing, slicing, and using the `len()` function) can be used on lists. And augmented assignment operators provide a nice shortcut to regular assignment operators. The `join()` method can join a list that contains multiple strings to return a single string.

It might be best to review this chapter if you're not yet comfortable with these programming concepts. In Chapter 8, you'll learn how to decrypt using the transposition cipher.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. With paper and pencil, encrypt the following messages with the key 9 using the transposition cipher. The number of characters has been provided for your convenience.
   - Underneath a huge oak tree there was of swine a huge company, (61 characters)
   - That grunted as they crunched the mast: For that was ripe and fell full fast. (77 characters)
   - Then they trotted away for the wind grew high: One acorn they left, and no more might you spy. (94 characters)
2. In the following program, is each `spam` a global or local variable?

```
spam = 42
def foo():
    global spam
    spam = 99
    print(spam)
```

3. What value does each of the following expressions evaluate to?

```
[0, 1, 2, 3, 4][2]
[[1, 2], [3, 4]][0]
[[1, 2], [3, 4]][0][1]
['hello'][0][1]
[2, 4, 6, 8, 10][1:3]
list('Hello world!')
list(range(10))[2]
```

4. What value does each of the following expressions evaluate to?

```
len([2, 4])
len([])
len(['', '', ''])
[4, 5, 6] + [1, 2, 3]
3 * [1, 2, 3] + [9]
42 in [41, 42, 42, 42]
```

5. What are the four augmented assignment operators?

8

DECRYPTING WITH THE TRANSPOSITION CIPHER

"Weakening encryption or creating backdoors to encrypted devices and data for use by the good guys would actually create vulnerabilities to be exploited by the bad guys."
—Tim Cook, CEO of Apple, 2015

Unlike the Caesar cipher, the decryption process for the transposition cipher is different from the encryption process. In this chapter, you'll create a separate program named *transpositionDecrypt.py* to handle decryption.

TOPICS COVERED IN THIS CHAPTER

- Decrypting with the transposition cipher
- The `round()`, `math.ceil()`, and `math.floor()` functions
- The `and` and `or` Boolean operators
- Truth tables

How to Decrypt with the Transposition Cipher on Paper

Pretend you've sent the ciphertext "Cenoonommstmme oo snnio. s s c" to a friend (and they already know that the secret key is 8). The first step for them to decrypt the ciphertext is to calculate the number of boxes they need to draw. To determine this number, they must divide the length of the ciphertext message by the key and round up to the nearest whole number if the result isn't already a whole number. The length of the ciphertext is 30 characters (the same as the plaintext) and the key is 8, so 30 divided by 8 is 3.75.

Rounding up 3.75 to 4, your friend will draw a grid of boxes with four columns (the number they just calculated) and eight rows (the key).

Your friend also needs to calculate the number of boxes to shade in. Using the total number of boxes (32), they subtract the length of the ciphertext (which is 30): 32 – 30 = 2. They shade in the *bottom* two boxes in the *rightmost* column.

Then they start filling in the boxes, placing one character of the ciphertext in each box. Starting at the top left, they fill in toward the right, as you did when you were encrypting. The ciphertext is "Cenoonommstmme oo snnio. s s c", so "Ceno" goes in the first row, "onom" goes in the second row, and so on. When they're done, the boxes will look like Figure 8-1 (a ■ represents a space).

C	e	n	o
o	n	o	m
m	s	t	m
m	e	■	o
o	■	s	n
n	i	o	.
■	s	■	
s	■	c	

Figure 8-1: Decrypting the message by reversing the grid

Your friend who received the ciphertext notices that when they read the text going down the columns, the original plaintext is restored: "Common sense is not so common."

To recap, the steps for decrypting are as follows:

1. Calculate the number of columns you need by dividing the length of the message by the key and then rounding up.
2. Draw boxes in columns and rows. Use the number of columns you calculated in step 1. The number of rows is the same as the key.
3. Calculate the number of boxes to shade in by taking the total number of boxes (the number of rows multiplied by the number of columns) and subtracting the length of the ciphertext message.
4. Shade in the number of boxes you calculated in step 3 at the bottom of the rightmost column.

5. Fill in the characters of the ciphertext starting at the top row and going from left to right. Skip any of the shaded boxes.
6. Get the plaintext by reading the leftmost column from top to bottom, and continuing to do the same in each column.

Note that if you used a different key, you'd draw the wrong number of rows. Even if you followed the other steps in the decryption process correctly, the plaintext would be random garbage (similar to if you used the wrong key with the Caesar cipher).

Source Code for the Transposition Cipher Decryption Program

Open a new file editor window by clicking **File ▸ New File**. Enter the following code into the file editor and then save it as *transpositionDecrypt.py*. Remember to place *pyperclip.py* in the same directory. Press F5 to run the program.

transpositionDecrypt.py

```
 1. # Transposition Cipher Decryption
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import math, pyperclip
 5.
 6. def main():
 7.     myMessage = 'Cenoonommstmme oo snnio. s s c'
 8.     myKey = 8
 9.
10.     plaintext = decryptMessage(myKey, myMessage)
11.
12.     # Print with a | (called "pipe" character) after it in case
13.     # there are spaces at the end of the decrypted message:
14.     print(plaintext + '|')
15.
16.     pyperclip.copy(plaintext)
17.
18.
19. def decryptMessage(key, message):
20.     # The transposition decrypt function will simulate the "columns" and
21.     # "rows" of the grid that the plaintext is written on by using a list
22.     # of strings. First, we need to calculate a few values.
23.
24.     # The number of "columns" in our transposition grid:
25.     numOfColumns = int(math.ceil(len(message) / float(key)))
26.     # The number of "rows" in our grid:
27.     numOfRows = key
28.     # The number of "shaded boxes" in the last "column" of the grid:
29.     numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
30.
31.     # Each string in plaintext represents a column in the grid:
32.     plaintext = [''] * numOfColumns
33.
```

```
    # The column and row variables point to where in the grid the next
    # character in the encrypted message will go:
    column = 0
    row = 0

    for symbol in message:
        plaintext[column] += symbol
        column += 1 # Point to the next column.

        # If there are no more columns OR we're at a shaded box, go back
        # to the first column and the next row:
        if (column == numOfColumns) or (column == numOfColumns - 1 and
          row >= numOfRows - numOfShadedBoxes):
            column = 0
            row += 1

    return ''.join(plaintext)


# If transpositionDecrypt.py is run (instead of imported as a module),
# call the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Transposition Cipher Decryption Program

When you run the *transpositionDecrypt.py* program, it produces this output:

```
Common sense is not so common.|
```

If you want to decrypt a different message or use a different key, change the value assigned to the `myMessage` and `myKey` variables on lines 7 and 8.

Importing Modules and Setting Up the main() Function

The first part of the *transpositionDecrypt.py* program is similar to the first part of *transpositionEncrypt.py*:

```
1. # Transposition Cipher Decryption
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import math, pyperclip
5.
6. def main():
7.     myMessage = 'Cenoonommstmme oo snnio. s s c'
8.     myKey = 8
9.
10.    plaintext = decryptMessage(myKey, myMessage)
11.
```

```
12.     # Print with a | (called "pipe" character) after it in case
13.     # there are spaces at the end of the decrypted message:
14.     print(plaintext + '|')
15.
16.     pyperclip.copy(plaintext)
```

The `pyperclip` module is imported along with another module named `math` on line 4. If you separate the module names with commas, you can import multiple modules with one `import` statement.

The `main()` function, which we start defining on line 6, creates variables named `myMessage` and `myKey` and then calls the decryption function `decryptMessage()`. The return value of `decryptMessage()` is the decrypted plaintext of the ciphertext and key. This is stored in a variable named `plaintext`, which is printed to the screen (with a pipe character at the end in case there are spaces at the end of the message) and then copied to the clipboard.

Decrypting the Message with the Key

The `decryptMessage()` function follows the six decrypting steps described on page 100 and then returns the results of decryption as a string. To make decryption easier, we'll use functions from the `math` module, which we imported earlier in the program.

The round(), math.ceil(), and math.floor() Functions

Python's `round()` function will round a floating-point number (a number with a decimal point) to the closest integer. The `math.ceil()` and `math.floor()` functions (in Python's `math` module) will round a number up and down, respectively.

When you divide numbers using the `/` operator, the expression returns a floating-point number (a number with a decimal point). This happens even if the number divides evenly. For example, enter the following into the interactive shell:

```
>>> 21 / 7
3.0
>>> 22 / 5
4.4
```

If you want to round a number to the nearest integer, you can use the `round()` function. To see how the function works, enter the following:

```
>>> round(4.2)
4
>>> round(4.9)
5
>>> round(5.0)
5
>>> round(22 / 5)
4
```

If you only want to round up, you need to use the `math.ceil()` function, which represents "ceiling." If you only want to round down, use `math.floor()`. These functions exist in the `math` module, which you need to import before calling them. Enter the following into the interactive shell:

```
>>> import math
>>> math.floor(4.0)
4
>>> math.floor(4.2)
4
>>> math.floor(4.9)
4
>>> math.ceil(4.0)
4
>>> math.ceil(4.2)
5
>>> math.ceil(4.9)
5
```

The `math.floor()` function will always remove the decimal point from the float and convert it to an integer to round down, and `math.ceil()` will instead increment the ones place of the float and convert it to an integer to round up.

The decryptMessage() Function

The `decryptMessage()` function implements each of the decryption steps as Python code. It takes an integer `key` and a `message` string as arguments. The `math.ceil()` function is used for the transposition decryption in `decryptMessage()` when the columns are calculated to determine the number of boxes that need to be made:

```
19. def decryptMessage(key, message):
20.     # The transposition decrypt function will simulate the "columns" and
21.     # "rows" of the grid that the plaintext is written on by using a list
22.     # of strings. First, we need to calculate a few values.
23.
24.     # The number of "columns" in our transposition grid:
25.     numOfColumns = int(math.ceil(len(message) / float(key)))
26.     # The number of "rows" in our grid:
27.     numOfRows = key
28.     # The number of "shaded boxes" in the last "column" of the grid:
29.     numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
```

Line 25 calculates the number of columns by dividing `len(message)` by the integer in `key`. This value is passed to the `math.ceil()` function, and that return value is stored in `numOfColumns`. To make this program compatible with Python 2, we call `float()` so the `key` becomes a floating-point value. In Python 2, the result of dividing two integers is automatically rounded down. Calling `float()` avoids this behavior without affecting the behavior under Python 3.

Line 27 calculates the number of rows, which is the integer stored in `key`. This value gets stored in the variable `numOfRows`.

Line 29 calculates the number of shaded boxes in the grid, which is the number of columns times rows, minus the length of the message.

If you're decrypting "Cenoonommstmme oo snnio. s s c" with a key of 8, `numOfColumns` is set to `4`, `numOfRows` is set to `8`, and `numOfShadedBoxes` is set to `2`.

Just like the encryption program had a variable named `ciphertext` that was a list of strings to represent the grid of ciphertext, `decryptMessage()` also has a list-of-strings variable named `plaintext`:

```
31.     # Each string in plaintext represents a column in the grid:
32.     plaintext = [''] * numOfColumns
```

These strings are blank at first, with one string for each column of the grid. Using list replication, you can multiply a list of one blank string by `numOfColumns` to make a list of several blank strings equal to the number of columns needed.

Keep in mind that this `plaintext` is different from the `plaintext` in the `main()` function. Because the `decryptMessage()` function and the `main()` function each has its own local scope, the functions' `plaintext` variables are different and just happen to have the same name.

Remember that the grid for the `'Cenoonommstmme oo snnio. s s c'` example looks like Figure 8-1 on page 100.

The `plaintext` variable will have a list of strings, and each string in the list will be a single column of this grid. For this decryption, you want `plaintext` to end up with the following value:

```
['Common s', 'ense is ', 'not so c', 'ommon.']
```

That way, you can join all the list's strings together to return the `'Common sense is not so common.'` string value.

To make the list, we first need to place each symbol in `message` in the correct string inside the `plaintext` list one at a time. We'll create two variables named `column` and `row` to track the column and row where the next character in `message` should go; these variables should start at `0` to begin at the first column and first row. Lines 36 and 37 do this:

```
34.     # The column and row variables point to where in the grid the next
35.     # character in the encrypted message will go:
36.     column = 0
37.     row = 0
```

Line 39 starts a `for` loop that iterates over the characters in the `message` string. Inside this loop, the code will adjust the `column` and `row` variables so it concatenates `symbol` to the correct string in the `plaintext` list:

```
39.     for symbol in message:
40.         plaintext[column] += symbol
41.         column += 1 # Point to the next column.
```

Line 40 concatenates `symbol` to the string at index `column` in the `plaintext` list, because each string in `plaintext` represents a column. Then line 41 adds `1` to `column` (that is, it *increments* `column`) so that on the next iteration of the loop, `symbol` will be concatenated to the next string in the `plaintext` list.

We've handled incrementing `column` and `row`, but we'll also need to reset the variables to `0` in some cases. To understand the code that does that, you'll need to understand Boolean operators.

Boolean Operators

Boolean operators compare Boolean values (or expressions that evaluate to a Boolean value) and evaluate to a Boolean value. The Boolean operators `and` and `or` can help you form more complicated conditions for `if` and `while` statements. The `and` operator connects two expressions and evaluates to `True` if both expressions evaluate to `True`. The `or` operator connects two expressions and evaluates to `True` if one or both expressions evaluate to `True`; otherwise, these expressions evaluate to `False`. Enter the following into the interactive shell to see how the `and` operator works:

```
>>> 10 > 5 and 2 < 4
True
>>> 10 > 5 and 4 != 4
False
```

The first expression evaluates to `True` because the expressions on either side of the `and` operator both evaluate to `True`. In other words, the expression `10 > 5 and 2 < 4` evaluates to `True and True`, which in turn evaluates to `True`.

However, in the second expression, although `10 > 5` evaluates to `True`, the expression `4 != 4` evaluates to `False`. This means the expression evaluates to `True and False`. Because both expressions have to be `True` for the `and` operator to evaluate to `True`, the whole expression evaluates to `False`.

If you ever forget how a Boolean operator works, you can look at its *truth table*, which shows what different combinations of Boolean values evaluate to based on the operator used. Table 8-1 is a truth table for the `and` operator.

Table 8-1: The `and` Operator Truth Table

A and B	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

To see how the or operator works, enter the following into the interactive shell:

```
>>> 10 > 5 or 4 != 4
True
>>> 10 < 5 or 4 != 4
False
```

When you're using the or operator, only one side of the expression must be True for the or operator to evaluate the whole expression as True, which is why `10 > 5 or 4 != 4` evaluates to True. However, because both the expression `10 < 5` and the expression `4 != 4` are False, the second expression evaluates to `False or False`, which in turn evaluates to False.

The or operator's truth table is shown in Table 8-2.

Table 8-2: The or Operator Truth Table

A or B	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The third Boolean operator is not. The not operator evaluates to the opposite Boolean value of the value it operates on. So `not True` is False and `not False` is True. Enter the following into the interactive shell:

```
>>> not 10 > 5
False
>>> not 10 < 5
True
>>> not False
True
>>> not not False
False
>>> not not not not not False
True
```

As you can see in the last two expressions, you can even use multiple not operators. The not operator's truth table is shown in Table 8-3.

Table 8-3: The not Operator Truth Table

not A	Evaluates to
not True	False
not False	True

The and and or Operators Are Shortcuts

Similar to how `for` loops let you do the same task as `while` loops but with less code, the `and` and `or` operators also let you shorten your code. Enter the following two pieces of code, which have the same result, into the interactive shell:

```
>>> if 10 > 5:
...   if 2 < 4:
...     print('Hello!')
...
Hello!
>>> if 10 > 5 and 2 < 4:
...   print('Hello!')
...
Hello!
```

The `and` operator can take the place of two `if` statements that check each part of the expression separately (where the second `if` statement is inside the first `if` statement's block).

You can also replace an if and `elif` statement with the `or` operator. To give this a try, enter the following into the interactive shell:

```
>>> if 4 != 4:
...   print('Hello!')
... elif 10 > 5:
...   print('Hello!')
...
Hello!
>>> if 4 != 4 or 10 > 5:
...   print('Hello!')
...
Hello!
```

The `if` and `elif` statements will each check a different part of the expression, whereas the `or` operator can check both statements in one line.

Order of Operations for Boolean Operators

You know that math operators have an order of operations, and so do the `and`, `or`, and `not` operators. First, `not` is evaluated, then `and`, and then `or`. Enter the following into the interactive shell:

```
>>> not False and False    # not False evaluates first
False
>>> not (False and False)  # (False and False) evaluates first
True
```

In the first line of code, `not False` is evaluated first, so the expression becomes `True and False`, which evaluates to `False`. In the second line, parentheses are evaluated first, even before the `not` operator, so `False and False` is evaluated as `False`, and the expression becomes `not (False)`, which is `True`.

Adjusting the column and row Variables

Now that you know how Boolean operators work, you can learn how the `column` and `row` variables are reset in *transpositionDecrypt.py*.

There are two cases in which you'll want to reset `column` to `0` so that on the next iteration of the loop, `symbol` is added to the first string in the list in `plaintext`. In the first case, you want to do this if `column` is incremented past the last index in `plaintext`. In this situation, the value in `column` will be equal to `numOfColumns`. (Remember that the last index in `plaintext` will be `numOfColumns` minus one. So when `column` is equal to `numOfColumns`, it's already past the last index.)

The second case is if `column` is at the last index and the `row` variable is pointing to a row that has a shaded box in the last column. As a visual example of that, the decryption grid with the column indexes along the top and the row indexes down the side is shown in Figure 8-2.

	0	1	2	3
0	C 0	e 1	n 2	o 3
1	o 4	n 5	o 6	m 7
2	m 8	s 9	t 10	m 11
3	m 12	e 13	■ 14	o 15
4	o 16	■ 17	s 18	n 19
5	n 20	i 21	o 22	. 23
6	■ 24	s 25	■ 26	
7	s 27	■ 28	c 29	

Figure 8-2: Decryption grid with column and row indexes

You can see that the shaded boxes are in the last column (whose index will be `numOfColumns - 1`) in rows 6 and 7. To calculate which row indexes potentially have shaded boxes, use the expression `row >= numOfRows - numOfShadedBoxes`. In our example with eight rows (with indexes 0 to 7), rows 6 and 7 are shaded. The number of unshaded boxes is the total number of rows (in our example, 8) minus the number of shaded boxes (in our example, 2). If the current `row` is equal to or greater than this number (8 – 2 = 6), we can know we have a shaded box. If this expression is `True` and `column` is also equal to `numOfColumns - 1`, then Python has encountered a shaded box; at this point, you want to reset `column` to `0` for the next iteration:

```
43.         # If there are no more columns OR we're at a shaded box, go back
44.         # to the first column and the next row:
45.         if (column == numOfColumns) or (column == numOfColumns - 1 and
              row >= numOfRows - numOfShadedBoxes):
46.             column = 0
47.             row += 1
```

These two cases are why the condition on line 45 is `(column == numOfColumns) or (column == numOfColumns - 1 and row >= numOfRows - numOfShadedBoxes)`. Although that looks like a big, complicated expression, remember that you can break it down into smaller parts. The expression `(column == numOfColumns)` checks whether the column variable is past the index range, and the second part of the expression checks whether we're at a `column` and `row` index that is a shaded box. If either of these two expressions is true, the block of code that executes will reset `column` to the first column by setting it to `0`. You'll also increment the `row` variable.

By the time the `for` loop on line 39 has finished looping over every character in `message`, the `plaintext` list's strings have been modified so they're now in the decrypted order (if the correct key was used). The strings in the `plaintext` list are joined together (with a blank string between each string) by the `join()` string method on line 49:

```
49.     return ''.join(plaintext)
```

Line 49 also returns the string that the `decryptMessage()` function returns.

For decryption, `plaintext` will be `['Common s', 'ense is ', 'not so c', 'ommon.']`, so `''.join(plaintext)` will evaluate to `'Common sense is not so common.'`

Calling the main() Function

The first line that our program runs after importing modules and executing the `def` statements is the `if` statement on line 54.

```
52. # If transpositionDecrypt.py is run (instead of imported as a module),
53. # call the main() function:
54. if __name__ == '__main__':
55.     main()
```

As with the transposition encryption program, Python checks whether this program has been run (instead of imported by a different program) by checking whether the `__name__` variable is set to the string value `'__main__'`. If so, the code executes the `main()` function.

Summary

That's it for the decryption program. Most of the program is in the `decryptMessage()` function. The programs we've written can encrypt and decrypt the message "Common sense is not so common." with the key 8; however, you should try several other messages and keys to check that a message that is encrypted and then decrypted results in the same original message. If you don't get the results you expect, you'll know that either the encryption code or the decryption code doesn't work. In Chapter 9, we'll automate this process by writing a program to test our programs.

If you'd like to see a step-by-step trace of the transposition cipher decryption program's execution, visit *https://www.nostarch.com/crackingcodes/*.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Using paper and pencil, decrypt the following messages with the key 9. The ■ marks a single space. The total number of characters has been counted for you.
   - H■cb■■irhdeuousBdi■■■prrtyevdgp■nir■■eerit■eatoreechadihf■pak en■ge■b■te■dih■aoa.da■tts■tn (89 characters)
   - A■b■■drottthawa■nwar■eci■t■nlel■ktShw■leec,hheat■.na■■e■soog mah■a■■ateniAcgakh■dmnor■■ (86 characters)
   - Bmmsrl■dpnaua!toeboo'ktn■uknrwos.■yaregonr■w■nd,tu■■oiady■h gtRwt■■■A■hhanhhasthtev■■e■†■e■■eo (93 characters)
2. When you enter the following code into the interactive shell, what does each line print?

```
>>> math.ceil(3.0)
>>> math.floor(3.1)
>>> round(3.1)
>>> round(3.5)
>>> False and False
>>> False or False
>>> not not True
```

3. Draw the complete truth tables for the `and`, `or`, and `not` operators.
4. Which of the following is correct?

```
if __name__ == '__main__':
if __main__ == '__name__':
if _name_ == '_main_':
if _main_ == '_name_':
```

9

PROGRAMMING A PROGRAM TO TEST YOUR PROGRAM

"It is poor civic hygiene to install technologies that could someday facilitate a police state."
—Bruce Schneier, Secrets and Lies

The transposition programs seem to work pretty well at encrypting and decrypting different messages with various keys, but how do you know they *always* work? You can't be absolutely sure the programs always work unless you test the `encryptMessage()` and `decryptMessage()` functions with all sorts of `message` and `key` parameter values. But this would take a lot of time because you'd have to type a message in the encryption program, set the key, run the encryption program, paste the ciphertext into the decryption program, set the key, and then run the decryption program. You'd also need to repeat that process with several different keys and messages, resulting in a lot of boring work!

Instead, let's write another program that generates a random message and a random key to test the cipher programs. This new program will encrypt the message with `encryptMessage()` from *transpositionEncrypt.py* and then pass the ciphertext to `decryptMessage()` from *transpositionDecrypt.py*. If the plaintext returned by `decryptMessage()` is the same as the original message, the program will know that the encryption and decryption programs work. The process of testing a program automatically using another program is called *automated testing*.

Several different message and key combinations need to be tried, but it takes the computer only a minute or so to test thousands of combinations. If all of those tests pass, you can be more certain that your code works.

TOPICS COVERED IN THIS CHAPTER

- The `random.randint()` function
- The `random.seed()` function
- List references
- The `copy.deepcopy()` functions
- The `random.shuffle()` function
- Randomly scrambling a string
- The `sys.exit()` function

Source Code for the Transposition Cipher Tester Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *transpositionTest.py*. Then press F5 to run the program.

transpositionTest.py

```
 1. # Transposition Cipher Test
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import random, sys, transpositionEncrypt, transpositionDecrypt
 5.
 6. def main():
 7.     random.seed(42) # Set the random "seed" to a static value.
 8.
 9.     for i in range(20): # Run 20 tests.
10.         # Generate random messages to test.
11.
12.         # The message will have a random length:
13.         message = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' * random.randint(4, 40)
14.
15.         # Convert the message string to a list to shuffle it:
16.         message = list(message)
```

```
        random.shuffle(message)
        message = ''.join(message) # Convert the list back to a string.

        print('Test #%s: "%s..."' % (i + 1, message[:50]))

        # Check all possible keys for each message:
        for key in range(1, int(len(message)/2)):
            encrypted = transpositionEncrypt.encryptMessage(key, message)
            decrypted = transpositionDecrypt.decryptMessage(key, encrypted)

            # If the decryption doesn't match the original message, display
            # an error message and quit:
            if message != decrypted:
                print('Mismatch with key %s and message %s.' % (key,
                  message))
                print('Decrypted as: ' + decrypted)
                sys.exit()

    print('Transposition cipher test passed.')


# If transpositionTest.py is run (instead of imported as a module) call
# the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Transposition Cipher Tester Program

When you run the *transpositionTest.py* program, the output should look like this:

```
Test #1: "JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQ..."
Test #2: "SWRCLUCRDOMLWZKOMAGVOTXUVVEPIOJMSBEQRQOFRGCCKENINV..."
Test #3: "BIZBPZUIWDUFXAPJTHCMDWEGHYOWKWWWSJYKDQVSFWCJNCOZZA..."
Test #4: "JEWBCEXVZAILLCHDZJCUTXASSZZRKRPMYGTGHBXPQPBEBVCODM..."
--snip--
Test #17: "KPKHHLPUWPSSIOULGKVEFHZOKBFHXUKVSEOWOENOZSNIDELAWR..."
Test #18: "OYLFXXZENDFGSXTEAHGHPBNORCFEPBMITILSSJRGDVMNSOMURV..."
Test #19: "SOCLYBRVDPLNVJKAFDGHCQMXIOPEJSXEAAXNWCCYAGZGLZGZHK..."
Test #20: "JXJGRBCKZXPUIEXOJUNZEYYSEAEGVOJWIRTSSGPUWPNZUBQNDA..."
Transposition cipher test passed.
```

The tester program works by importing the *transpositionEncrypt.py* and *transpositionDecrypt.py* programs as modules. Then the tester program calls `encryptMessage()` and `decryptMessage()` from the encryption and decryption programs. The tester program creates a random message and chooses a random key. It doesn't matter that the message is just random letters, because the program only needs to check that encrypting and then decrypting the message results in the original message.

Using a loop, the program repeats this test 20 times. If at any point the string returned from `transpositionDecrypt()` isn't the same as the original message, the program prints an error and exits.

Let's explore the source code in more detail.

Importing the Modules

The program starts by importing modules, including two you've already seen that come with Python, `random` and `sys`:

```
1. # Transposition Cipher Test
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import random, sys, transpositionEncrypt, transpositionDecrypt
```

We also need to import the transposition cipher programs (that is, *transpositionEncrypt.py* and *transpositionDecrypt.py*) by just typing their names without the *.py* extension.

Creating Pseudorandom Numbers

To create random numbers to generate the messages and keys, we'll use the `random` module's `seed()` function. Before we delve into what the seed does, let's look at how random numbers work in Python by trying out the `random.randint()` function. The `random.randint()` function that we'll use later in the program takes two integer arguments and returns a random integer between those two integers (including the integers). Enter the following into the interactive shell:

```
>>> import random
>>> random.randint(1, 20)
20
>>> random.randint(1, 20)
18
>>> random.randint(100, 200)
107
```

Of course, the numbers you get will probably be different from those shown here because they're random numbers.

But the numbers generated by Python's `random.randint()` function are not truly random. They're produced from a pseudorandom number generator algorithm, which takes an initial number and produces other numbers based on a formula.

The initial number that the pseudorandom number generator starts with is called the *seed*. If you know the seed, the rest of the numbers the generator produces are predictable, because when you set the seed to a

specific number, the same numbers will be generated in the same order. These random-looking but predictable numbers are called *pseudorandom numbers*. Python programs for which you don't set a seed use the computer's current clock time to set a seed. You can reset Python's random seed by calling the `random.seed()` function.

To see proof that the pseudorandom numbers aren't completely random, enter the following into the interactive shell:

```
>>> import random
❶ >>> random.seed(42)
❷ >>> numbers = []
>>> for i in range(20):
...   numbers.append(random.randint(1, 10))
...
❸ [2, 1, 5, 4, 4, 3, 2, 9, 2, 10, 7, 1, 1, 2, 4, 4, 9, 10, 1, 9]
>>> random.seed(42)
>>> numbers = []
>>> for i in range(20):
...   numbers.append(random.randint(1, 10))
...
❹ [2, 1, 5, 4, 4, 3, 2, 9, 2, 10, 7, 1, 1, 2, 4, 4, 9, 10, 1, 9]
```

In this code, we generate 20 numbers twice using the same seed. First, we import `random` and set the seed to `42` ❶. Then we set up a list called `numbers` ❷ where we'll store our generated numbers. We use a `for` loop to generate 20 numbers and append each one to the `numbers` list, which we print so we can see every number that was generated ❸.

When the seed for Python's pseudorandom number generator is set to `42`, the first "random" number between `1` and `10` will always be `2`. The second number will always be `1`, the third number will always be `5`, and so on. When you reset the seed to `42` and generate numbers with the seed again, the same set of pseudorandom numbers is returned from `random.randint()`, as you can see by comparing the `numbers` list at ❸ and ❹.

Random numbers will become important for ciphers in later chapters, because they're used not only for testing ciphers but also for encrypting and decrypting in more complex ciphers. Random numbers are so important that one common security flaw in encryption software is using predictable random numbers. If the random numbers in your programs can be predicted, a cryptanalyst can use this information to break your cipher.

Selecting encryption keys in a truly random manner is necessary for the security of a cipher, but for other uses, such as this code test, pseudorandom numbers are fine. We'll use pseudorandom numbers to generate test strings for our tester program. You can generate truly random numbers with Python by using the `random.SystemRandom().randint()` function, which you can learn more about at *https://www.nostarch.com/crackingcodes/*.

Creating a Random String

Now that you've learned how to use `random.randint()` and `random.seed()` to create random numbers, let's return to the source code. To completely automate our encryption and decryption programs, we'll need to automatically generate random string messages.

To do this, we'll take a string of characters to use in the messages, duplicate it a random number of times, and store that as a string. Then, we'll take the string of the duplicated characters and scramble them to make them more random. We'll generate a new random string for each test so we can try many different letter combinations.

First, let's set up the `main()` function, which contains code that tests the cipher programs. It starts by setting a seed for the pseudorandom string:

```
6. def main():
7.     random.seed(42) # Set the random "seed" to a static value.
```

Setting the random seed by calling `random.seed()` is useful for the tester program because you want predictable numbers so the same pseudorandom messages and keys are chosen each time the program is run. As a result, if you notice one message fails to encrypt and decrypt properly, you'll be able to reproduce this failing test case.

Next, we'll duplicate a string using a `for` loop.

Duplicating a String a Random Number of Times

We'll use a `for` loop to run 20 tests and to generate our random message:

```
 9.     for i in range(20): # Run 20 tests.
10.         # Generate random messages to test.
11.
12.         # The message will have a random length:
13.         message = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' * random.randint(4, 40)
```

Each time the `for` loop iterates, the program will create and test a new message. We want this program to run multiple tests because the more tests we try, the more certain we'll be that the programs work.

Line 13 is the first line of testing code and creates a message of a random length. It takes a string of uppercase letters and uses `randint()` and string replication to duplicate the string a random number of times between `4` and `40`. Then it stores the new string in the `message` variable.

If we leave the `message` string as it is now, it will always just be the alphabet string repeated a random number of times. Because we want to test different combinations of characters, we'll need to take things a step further and scramble the characters in `message`. To do that, let's first learn a bit more about lists.

List Variables Use References

Variables store lists differently than they store other values. A variable will contain a reference to the list, rather than the list itself. A *reference* is a value that points to some bit of data, and a *list reference* is a value that points to a list. This results in slightly different behavior for your code.

You already know that variables store strings and integer values. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

We assign `42` to the `spam` variable, and then copy the value in `spam` and assign it to the variable `cheese`. When we later change the value in `spam` to `100`, the new number doesn't affect the value in `cheese` because `spam` and `cheese` are different variables that store different values.

But lists don't work this way. When we assign a list to a variable, we are actually assigning a list reference to the variable. The following code makes this distinction easier to understand. Enter this code into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

This code might look odd to you. The code changed only the `cheese` list, but both the `cheese` and `spam` lists have changed.

When we create the list ❶, we assign a reference to it in the `spam` variable. But the next line ❷ copies only the list reference in `spam` to `cheese`, not the list value. This means the values stored in `spam` and `cheese` now both refer to the same list. There is only one underlying list because the actual list was never actually copied. So when we modify the first element of `cheese` ❸, we are modifying the same list that `spam` refers to.

Remember that variables are like boxes that contain values. But list variables don't actually contain lists—they contain references to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 9-1 shows what happens when a list is assigned to the `spam` variable.

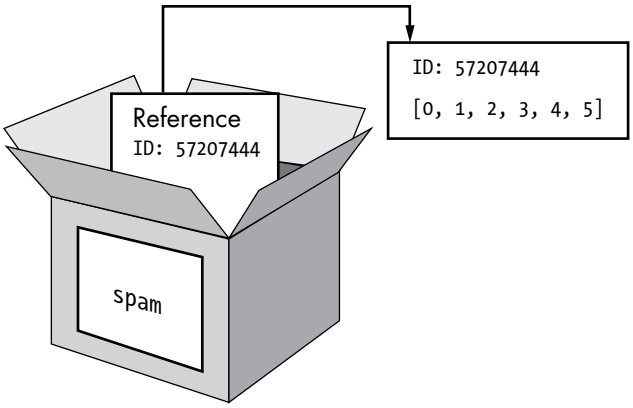


Figure 9-1: `spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.

Then, in Figure 9-2, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new list. Notice that both references refer to the same list.

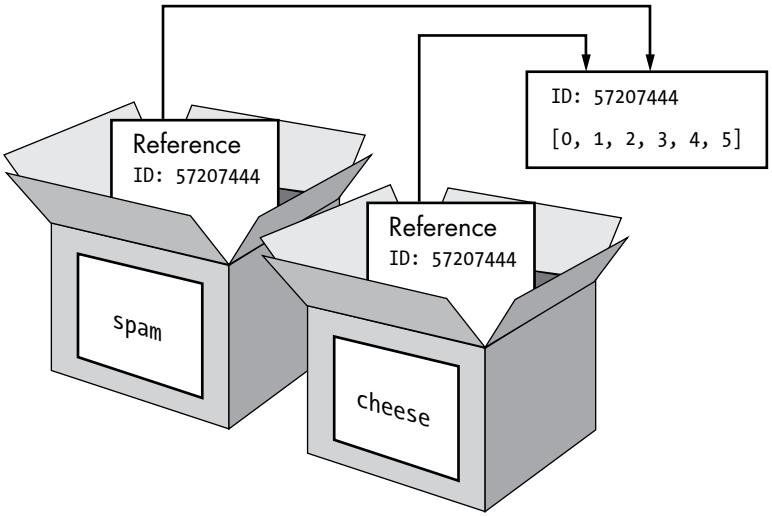


Figure 9-2: `spam = cheese` copies the reference, not the list.

When we alter the list that `cheese` refers to, the list that `spam` refers to also changes, because both `cheese` and `spam` refer to the same list. You can see this in Figure 9-3.

Although Python variables technically contain references to list values, people often casually say that the variable "contains the list."

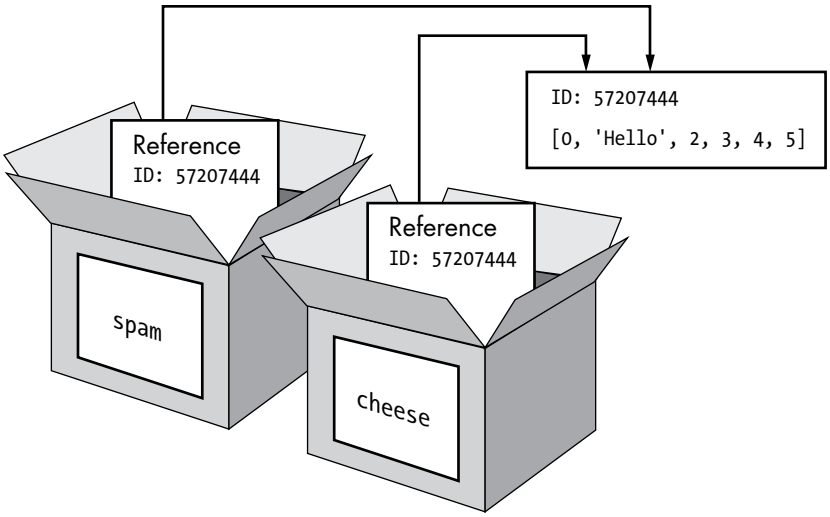


Figure 9-3: `cheese[1] = 'Hello!'` modifies the list that both variables refer to.

Passing References

References are particularly important for understanding how arguments are passed to functions. When a function is called, the arguments' values are copied to the parameter variables. For lists, this means a copy of the reference is used for the parameter. To see the consequences of this action, open a new file editor window, enter the following code, and save it as *passingReference.py*. Press F5 to run the code.

passingReference.py

```
def eggs(someParameter):
    someParameter.append('Hello')
spam = [1, 2, 3]
eggs(spam)
print(spam)
```

When you run the code, notice that when `eggs()` is called, a return value isn't used to assign a new value to `spam`. Instead, the list is modified directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: forgetting that Python handles list variables this way can lead to confusing bugs.

Using copy.deepcopy() to Duplicate a List

If you want to copy a list value, you can import the `copy` module to call the `copy.deepcopy()` function, which returns a separate copy of the list it is passed:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> import copy
>>> cheese = copy.deepcopy(spam)
>>> cheese[1] = 'Hello!'
>>> spam
[0, 1, 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Because the `copy.deepcopy()` function was used to copy the list in `spam` to `cheese`, when an item in `cheese` is changed, `spam` is unaffected.

We'll use this function in Chapter 17 when we hack the simple substitution cipher.

The random.shuffle() Function

With a foundation in how references work, you can now understand how the `random.shuffle()` function that we'll use next works. The `random.shuffle()` function is part of the `random` module and accepts a list argument whose items it randomly rearranges. Enter the following into the interactive shell to see how `random.shuffle()` works:

```
>>> import random
>>> spam = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> spam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle(spam)
>>> spam
[3, 0, 5, 9, 6, 8, 2, 4, 1, 7]
>>> random.shuffle(spam)
>>> spam
[1, 2, 5, 9, 4, 7, 0, 3, 6, 8]
```

An important detail to note is that `shuffle()` *does not return a list value.* Instead, it changes the list value that is passed to it (because `shuffle()` modifies the list directly from the list reference value it is passed). The `shuffle()` function modifies the list *in place*, which is why we execute `random.shuffle(spam)` instead of `spam = random.shuffle(spam)`.

Randomly Scrambling a String

Let's return to *transpositionTest.py*. To shuffle the characters in a string value, we first need to convert the string to a list using `list()`:

```
15.         # Convert the message string to a list to shuffle it:
16.         message = list(message)
17.         random.shuffle(message)
18.         message = ''.join(message) # Convert the list back to a string.
```

The return value from `list()` is a list value with one-character strings of each character in the string passed to it; so on line 16, we're reassigning `message` to be a list of its characters. Next, `shuffle()` randomizes the order of the items in `message`. Then the program converts the list of strings back to a string value using the `join()` string method. This shuffling of the `message` string allows us to test many different messages.

Testing Each Message

Now that the random message has been made, the program tests the encryption and decryption functions with it. We'll have the program print some feedback so we can see what it's doing while it's testing:

```
20.         print('Test #%s: "%s..."' % (i + 1, message[:50]))
```

Line 20 has a `print()` call that displays which test number the program is on (we need to add 1 to `i` because `i` starts at `0` and the test numbers should start at `1`). Because the string in `message` can be long, we use string slicing to show only the first 50 characters of `message`.

Line 20 also uses string interpolation. The value that `i + 1` evaluates to replaces the first `%s` in the string, and the value that `message[:50]` evaluates to replaces the second `%s`. When using string interpolation, be sure the number of `%s` in the string matches the number of values that are between the parentheses after it.

Next, we'll test all the possible keys. Although the key for the Caesar cipher could be an integer from `0` to `65` (the length of the symbol set), the key for the transposition cipher can be between `1` and half the length of the message. The `for` loop on line 23 runs the test code with the keys `1` up to (but not including) the length of the message.

```
22.         # Check all possible keys for each message:
23.         for key in range(1, int(len(message)/2)):
24.             encrypted = transpositionEncrypt.encryptMessage(key, message)
25.             decrypted = transpositionDecrypt.decryptMessage(key, encrypted)
```

Line 24 encrypts the string in `message` using the `encryptMessage()` function. Because this function is inside the *transpositionEncrypt.py* file, we need

to add `transpositionEncrypt.` (with the period at the end) to the front of the function name.

The encrypted string that is returned from `encryptMessage()` is then passed to `decryptMessage()`. We need to use the same key for both function calls. The return value from `decryptMessage()` is stored in a variable named `decrypted`. If the functions worked, the string in `message` should be the same as the string in `decrypted`. We'll look at how the program checks this next.

Checking Whether the Cipher Worked and Ending the Program

After we've encrypted and decrypted the message, we need to check whether both processes worked correctly. To do that, we simply need to check whether the original message is the same as the decrypted message.

```
27.             # If the decryption doesn't match the original message, display
28.             # an error message and quit:
29.             if message != decrypted:
30.                 print('Mismatch with key %s and message %s.' % (key,
                      message))
31.                 print('Decrypted as: ' + decrypted)
32.                 sys.exit()
33.
34.     print('Transposition cipher test passed.')
```

Line 29 tests whether `message` and `decrypted` are equal. If they aren't, Python displays an error message on the screen. Lines 30 and 31 print the `key`, `message`, and `decrypted` values as feedback to help us figure out what went wrong. Then the program exits.

Normally, programs exit when the execution reaches the end of the code and there are no more lines to execute. However, when `sys.exit()` is called, the program ends immediately and stops testing new messages (because you'll want to fix your cipher programs if even one test fails!).

But if the values in `message` and `decrypted` are equal, the program execution skips the `if` statement's block and the call to `sys.exit()`. The program continues looping until it finishes running all of its tests. After the loop ends, the program runs line 34, which you know is outside of line 9's loop because it has different indentation. Line 34 prints `'Transposition cipher test passed.'`.

Calling the main() Function

As with our other programs, we want to check whether the program is being imported as a module or being run as the main program.

```
37. # If transpositionTest.py is run (instead of imported as a module) call
38. # the main() function:
39. if __name__ == '__main__':
40.     main()
```

Lines 39 and 40 do the trick, checking whether the special variable `__name__` is set to `'__main__'` and if so, calling the `main()` function.

Testing the Test Program

We've written a program that tests the transposition cipher programs, but how do we know that the test program works? What if the test program has a bug, and it just indicates that the transposition cipher programs work when they really don't?

We can test the test program by purposely adding bugs to the encryption or decryption functions. Then, if the test program doesn't detect a problem, we know that it isn't running as expected.

To add a bug to the program, we open *transpositionEncrypt.py* and add `+ 1` to line 36:

transpositionEncrypt.py

```
35.             # Move currentIndex over:
36.             currentIndex += key + 1
```

Now that the encryption code is broken, when we run the test program, it should print an error, like this:

```
Test #1: "JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQ..."
Mismatch with key 1 and message
JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQTRGJHDVCZECRESZJARAVIPFOBWZ
XXTBFOFHVSIGBWIBBHGKUWHEUUDYONYTZVKNVVTYZPDDMIDKBHTYJAHBNDVJUZDCEMFMLUXEONCZX
WAWGXZSFTMJNLJOKKIJXLWAPCQNYCIQOFTEAUHRJODKLGRIZSJBXQPBMQPPFGMVUZHKFWPGNMRYXR
OMSCEEXLUSCFHNELYPYKCNYTOUQGBFSRDDMVIGXNYPHVPQISTATKVKM.
Decrypted as:
JQDKZACYCPTRLHBQEWLWGQRIITGHVZCEZAAIFBZXBOHSGWBHKWEUYNTVNVYPDIKHYABDJZCMMUENZ
WWXSTJLOKJLACNCQFEUROKGISBQBQPGVZKWGMYRMCELSFNLPKNTUGFRDVGNPVQSAKK
```

The test program failed at the first message after we purposely inserted a bug, so we know that it's working exactly as we planned!

Summary

You can use your new programming skills for more than just writing programs. You can also program the computer to test programs you write to make sure they work for different inputs. Writing code to test code is a common practice.

In this chapter, you learned how to use the `random.randint()` function to produce pseudorandom numbers and how to use `random.seed()` to reset the seed to create more pseudorandom numbers. Although pseudorandom numbers aren't random enough to use in cryptography programs, they're good enough to use in this chapter's testing program.

You also learned the difference between a list and list reference and that the `copy.deepcopy()` function will create copies of list values instead of

reference values. Additionally, you learned how the `random.shuffle()` function can scramble the order of items in a list value by shuffling list items in place using references.

All of the programs we've created so far encrypt only short messages. In Chapter 10, you'll learn how to encrypt and decrypt entire files.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. If you ran the following program and it printed the number `8`, what would it print the next time you ran it?

```
import random
random.seed(9)
print(random.randint(1, 10))
```

2. What does the following program print?

```
spam = [1, 2, 3]
eggs = spam
ham = eggs
ham[0] = 99
print(ham == spam)
```

3. Which module contains the `deepcopy()` function?
4. What does the following program print?

```
import copy
spam = [1, 2, 3]
eggs = copy.deepcopy(spam)
ham = copy.deepcopy(eggs)
ham[0] = 99
print(ham == spam)
```

10

ENCRYPTING AND DECRYPTING FILES

"Why do security police grab people and torture them? To get their information. And hard disks put up no resistance to torture. You need to give the hard disk a way to resist. That's cryptography."
—Patrick Ball, Human Rights Data Analysis Group

In previous chapters, our programs have only worked on small messages that we type directly into the source code as string values.

The cipher program we'll make in this chapter will allow you to encrypt and decrypt entire files, which can be millions of characters in size.

TOPICS COVERED IN THIS CHAPTER

- The `open()` function
- Reading and writing files
- The `write()`, `close()`, and `read()` file object methods
- The `os.path.exists()` function
- The `upper()`, `lower()`, and `title()` string methods
- The `startswith()` and `endswith()` string methods
- The `time` module and `time.time()` function

Plain Text Files

The transposition file cipher program encrypts and decrypts plain (unformatted) text files. These are the kind of files that only have text data and usually have the *.txt* file extension. You can write your own text files with programs such as Notepad on Windows, TextEdit on macOS, and gedit on Linux. (Word processing programs can produce plain text files as well, but keep in mind that they won't save any font, size, color, or other formatting.) You can even use IDLE's file editor by saving the files with a *.txt* extension instead of the usual *.py* extension.

For some samples, you can download text files from *https://www.nostarch.com/crackingcodes/*. These sample text files are of books that are now in the public domain and legal to download and use. For example, Mary Shelley's classic novel *Frankenstein* has more than 78,000 words in its text file! To type this book into an encryption program would take a lot of time, but by using the downloaded file, the program can do the encryption in a couple of seconds.

Source Code for the Transposition File Cipher Program

As with the transposition cipher–testing program, the transposition file cipher program imports the *transpositionEncrypt.py* and *transpositionDecrypt.py* files so it can call the `encryptMessage()` and `decryptMessage()` functions. As a result, you don't have to retype the code for these functions in the new program.

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *transpositionFileCipher.py*. Then download *frankenstein.txt* from *https://www.nostarch.com/crackingcodes/* and place this file in the same folder as the *transpositionFileCipher.py* file. Press F5 to run the program.

transposition FileCipher.py

```
# Transposition Cipher Encrypt/Decrypt File
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import time, os, sys, transpositionEncrypt, transpositionDecrypt

def main():
    inputFilename = 'frankenstein.txt'
    # BE CAREFUL! If a file with the outputFilename name already exists,
    # this program will overwrite that file:
    outputFilename = 'frankenstein.encrypted.txt'
    myKey = 10
    myMode = 'encrypt' # Set to 'encrypt' or 'decrypt'.

    # If the input file does not exist, the program terminates early:
    if not os.path.exists(inputFilename):
        print('The file %s does not exist. Quitting...' % (inputFilename))
        sys.exit()

    # If the output file already exists, give the user a chance to quit:
    if os.path.exists(outputFilename):
        print('This will overwrite the file %s. (C)ontinue or (Q)uit?' %
          (outputFilename))
        response = input('> ')
        if not response.lower().startswith('c'):
            sys.exit()

    # Read in the message from the input file:
    fileObj = open(inputFilename)
    content = fileObj.read()
    fileObj.close()

    print('%sing...' % (myMode.title()))

    # Measure how long the encryption/decryption takes:
    startTime = time.time()
    if myMode == 'encrypt':
        translated = transpositionEncrypt.encryptMessage(myKey, content)
    elif myMode == 'decrypt':
        translated = transpositionDecrypt.decryptMessage(myKey, content)
    totalTime = round(time.time() - startTime, 2)
    print('%sion time: %s seconds' % (myMode.title(), totalTime))

    # Write out the translated message to the output file:
    outputFileObj = open(outputFilename, 'w')
    outputFileObj.write(translated)
    outputFileObj.close()

    print('Done %sing %s (%s characters).' % (myMode, inputFilename,
      len(content)))
    print('%sed file is %s.' % (myMode.title(), outputFilename))


```

```
51. # If transpositionCipherFile.py is run (instead of imported as a module),
52. # call the main() function:
53. if __name__ == '__main__':
54.     main()
```

Sample Run of the Transposition File Cipher Program

When you run the *transpositionFileCipher.py* program, it should produce this output:

```
Encrypting...
Encryption time: 1.21 seconds
Done encrypting frankenstein.txt (441034 characters).
Encrypted file is frankenstein.encrypted.txt.
```

A new *frankenstein.encrypted.txt* file is created in the same folder as *transpositionFileCipher.py*. When you open this file with IDLE's file editor, you'll see the encrypted contents of *frankenstein.py*. It should look something like this:

```
PtFiyedleo  a arnvmt eneeGLchongnes MmuyedlsuO#uiSHTGA r sy,n t ys
s nuaoGeL
sc7s,
--snip--
```

Once you have an encrypted text, you can send it to someone else to decrypt it. The recipient will also need to have the transposition file cipher program.

To decrypt the text, make the following changes to the source code (in bold) and run the transposition file cipher program again:

```
 7.     inputFilename = 'frankenstein.encrypted.txt'
 8.     # BE CAREFUL! If a file with the outputFilename name already exists,
 9.     # this program will overwrite that file:
10.     outputFilename = 'frankenstein.decrypted.txt'
11.     myKey = 10
12.     myMode = 'decrypt' # Set to 'encrypt' or 'decrypt'.
```

This time when you run the program, a new file named *frankenstein.decrypted.txt* that is identical to the original *frankenstein.txt* file will appear in the folder.

Working with Files

Before we dive into the code for *transpositionFileCipher.py*, let's examine how Python works with files. The three steps to reading the contents of a file are opening the file, reading the file content into a variable, and closing the file. Similarly, to write new content in a file, you must open (or create) the file, write the new content, and close the file.

Opening Files

Python can open a file to read from or write to using the `open()` function. The `open()` function's first parameter is the name of the file to open. If the file is in the same folder as the Python program, you can just use the file's name, such as `'thetimemachine.txt'`. The command to open *thetimemachine.txt* if it existed in the same folder as your Python program would look like this:

```
fileObj = open('thetimemachine.txt')
```

A file object is stored in the `fileObj` variable, which will be used to read from or write to the file.

You can also specify the *absolute path* of the file, which includes the folders and parent folders that the file is in. For example, `'C:\\Users\\Al\\frankenstein.txt'` (on Windows) and `'/Users/Al/frankenstein.txt'` (on macOS and Linux) are absolute paths. Remember that on Windows the backslash (\) must be escaped by typing another backslash before it.

For example, if you wanted to open the *frankenstein.txt* file, you would pass the path of the file as a string for the `open()` function's first parameter (and format the absolute path according to your operating system):

```
fileObj = open('C:\\Users\\Al\\frankenstein.txt')
```

The file object has several methods for writing to, reading from, and closing the file.

Writing to and Closing Files

For the encryption program, after reading in the text file's content, you'll need to write the encrypted (or decrypted) content to a new file, which you'll do by using the `write()` method.

To use `write()` on a file object, you need to open the file object in write mode, which you do by passing `open()` the string `'w'` as a second argument. (This second argument is an *optional parameter* because the `open()` function can still be used without passing two arguments.) For example, enter the following line of code into the interactive shell:

```
>>> fileObj = open('spam.txt', 'w')
```

This line creates a file named *spam.txt* in write mode so you can edit it. If a file of the same name exists where the `open()` function creates the new file, the old file is overwritten, so be careful when using `open()` in write mode.

With *spam.txt* now open in write mode, you can write to the file by calling the `write()` method on it. The `write()` method takes one argument: a string of text to write to the file. Enter the following into the interactive shell to write `'Hello, world!'` to *spam.txt*:

```
>>> fileObj.write('Hello, world!')
13
```

Passing the string `'Hello, world!'` to the `write()` method writes that string to the *spam.txt* file and then prints `13`, the number of characters in the string written to the file.

When you're finished working with a file, you need to tell Python you're done with the file by calling the `close()` method on the file object:

```
>>> fileObj.close()
```

There is also an append mode, which is like write mode except append mode doesn't overwrite the file. Instead, strings are written to the end of the content already in the file. Although we won't use it in this program, you can open a file in append mode by passing the string `'a'` as the second argument to `open()`.

If you get an `io.UnsupportedOperation: not readable` error message when you try calling `write()` on a file object, you might not have opened the file in write mode. When you don't include the `open()` function's optional parameter, it automatically opens the file object in read mode (`'r'`) instead, which allows you to use only the `read()` method on the file object.

Reading from a File

The `read()` method returns a string containing all the text in the file. To try it out, we'll read the *spam.txt* file we created earlier with the `write()` method. Run the following code from the interactive shell:

```
>>> fileObj = open('spam.txt', 'r')
>>> content = fileObj.read()
>>> print(content)
Hello world!
>>> fileObj.close()
```

The file is opened, and the file object that is created is stored in the `fileObj` variable. Once you have the file object, you can read the file using the `read()` method and store it in the `content` variable, which you then print. When you're done with the file object, you need to close it with `close()`.

If you get the error message `IOError: [Errno 2] No such file or directory`, make sure the file actually is where you think it is and double-check that you typed the filename and folder name correctly. (*Directory* is another word for *folder*.)

We'll use `open()`, `read()`, `write()`, and `close()` on the files that we open to encrypt or decrypt in *transpositionFileCipher.py*.

Setting Up the main() Function

The first part of the *transpositionFileCipher.py* program should look familiar. Line 4 is an `import` statement for the programs *transpositionEncrypt.py* and *transpositionDecrypt.py* as well as Python's `time`, `os`, and `sys` modules. Then we start `main()` by setting up some variables to use in the program.

```
 1. # Transposition Cipher Encrypt/Decrypt File
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import time, os, sys, transpositionEncrypt, transpositionDecrypt
 5.
 6. def main():
 7.     inputFilename = 'frankenstein.txt'
 8.     # BE CAREFUL! If a file with the outputFilename name already exists,
 9.     # this program will overwrite that file:
10.     outputFilename = 'frankenstein.encrypted.txt'
11.     myKey = 10
12.     myMode = 'encrypt' # Set to 'encrypt' or 'decrypt'.
```

The `inputFilename` variable holds a string of the file to read, and the encrypted (or decrypted) text is written to the file named in `outputFilename`. The transposition cipher uses an integer for a key, which is stored in `myKey`. The program expects `myMode` to store `'encrypt'` or `'decrypt'` to tell it to encrypt or decrypt the `inputFilename` file. But before we can read from the `inputFilename` file, we need to check that it exists using `os.path.exists()`.

Checking Whether a File Exists

Reading files is always harmless, but you need to be careful when writing to files. Calling the `open()` function in write mode on a filename that already exists overwrites the original content. Using the `os.path.exists()` function, your programs can check whether or not that file already exists.

The os.path.exists() Function

The `os.path.exists()` function takes a single string argument for a filename or a path to a file and returns `True` if the file already exists and `False` if it doesn't. The `os.path.exists()` function exists inside the `path` module, which exists inside the `os` module, so when we import the `os` module, the `path` module is imported, too.

Enter the following into the interactive shell:

```
>>> import os
❶ >>> os.path.exists('spam.txt')
False
>>> os.path.exists('C:\\Windows\\System32\\calc.exe') # Windows
True
>>> os.path.exists('/usr/local/bin/idle3') # macOS
False
>>> os.path.exists('/usr/bin/idle3') # Linux
False
```

In this example, the `os.path.exists()` function confirms that the *calc.exe* file exists in Windows. Of course, you'll only get these results if you're running Python on Windows. Remember to escape the backslash in a Windows file path by typing another backslash before it. If you're using macOS, only

the macOS example will return `True`, and only the last example will return `True` for Linux. If the full file path isn't given ❶, Python will check the current working directory. For IDLE's interactive shell, this is the folder that Python is installed in.

Checking Whether the Input File Exists with the os.path.exists() Function

We use the `os.path.exists()` function to check that the filename in `inputFilename` exists. Otherwise, we have no file to encrypt or decrypt. We do this in lines 14 to 17:

```
14.     # If the input file does not exist, then the program terminates early:
15.     if not os.path.exists(inputFilename):
16.         print('The file %s does not exist. Quitting...' % (inputFilename))
17.         sys.exit()
```

If the file doesn't exist, we display a message to the user and then quit the program.

Using String Methods to Make User Input More Flexible

Next, the program checks whether a file with the same name as `outputFilename` exists, and if so, it asks the user to type `C` if they want to continue running the program or `Q` to quit the program. Because a user might type various responses, such as `'c'`, `'C'`, or even the word `'Continue'`, we want to make sure the program will accept all of these versions. To do this, we'll use more string methods.

The upper(), lower(), and title() String Methods

The `upper()` and `lower()` string methods will return the string they are called on in all uppercase or all lowercase letters, respectively. Enter the following into the interactive shell to see how the methods work on the same string:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.lower()
'hello'
```

Just as the `lower()` and `upper()` string methods return a string in lowercase or uppercase, the `title()` string method returns a string in title case. *Title case* is where the first character of every word is uppercase and the rest of the characters are lowercase. Enter the following into the interactive shell:

```
>>> 'hello'.title()
'Hello'
>>> 'HELLO'.title()
'Hello'
>>> 'extra! extra! man bites shark!'.title()
'Extra! Extra! Man Bites Shark!'
```

We'll use `title()` a little later in the program to format messages we output for the user.

The startswith() and endswith() String Methods

The `startswith()` method returns `True` if its string argument is found at the beginning of the string. Enter the following into the interactive shell:

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.startswith('H')
False
>>> spam = 'Albert'
❶ >>> spam.startswith('Al')
True
```

The `startswith()` method is case sensitive and can also be used on strings with multiple characters ❶.

The `endswith()` string method is used to check whether a string value ends with another specified string value. Enter the following into the interactive shell:

```
>>> 'Hello world!'.endswith('world!')
True
❷ >>> 'Hello world!'.endswith('world')
False
```

The string values must match perfectly. Notice that the lack of the exclamation mark in `'world'` ❷ causes `endswith()` to return `False`.

Using These String Methods in the Program

As noted, we want the program to accept any response that starts with a *C* regardless of capitalization. This means that we want the file to be overwritten whether the user types `c`, `continue`, `C`, or another string that begins with *C*. We'll use the string methods `lower()` and `startswith()` to make the program more flexible when taking user input:

```
19.     # If the output file already exists, give the user a chance to quit:
20.     if os.path.exists(outputFilename):
21.         print('This will overwrite the file %s. (C)ontinue or (Q)uit?' %
              (outputFilename))
22.         response = input('> ')
23.         if not response.lower().startswith('c'):
24.             sys.exit()
```

On line 23, we take the first letter of the string and check whether it is a *C* using the `startswith()` method. The `startswith()` method that we use is case sensitive and checks for a lowercase `'c'`, so we use the `lower()` method to modify the `response` string's capitalization to always be lowercase. If the user didn't enter a response starting with a *C*, then `startswith()` returns `False`,

which makes the `if` statement evaluate to `True` (because of the `not` in the `if` statement), and `sys.exit()` is called to end the program. Technically, the user doesn't have to enter `Q` to quit; any string that doesn't begin with *C* causes the `sys.exit()` function to be called to quit the program.

Reading the Input File

On line 27, we start using the file object methods discussed at the beginning of this chapter.

```
26.     # Read in the message from the input file:
27.     fileObj = open(inputFilename)
28.     content = fileObj.read()
29.     fileObj.close()
30.
31.     print('%sing...' % (myMode.title()))
```

Lines 27 to 29 open the file stored in `inputFilename`, read its contents into the `content` variable, and then close the file. After reading in the file, line 31 outputs a message for the user telling them that the encryption or decryption has begun. Because `myMode` should either contain the string `'encrypt'` or `'decrypt'`, calling the `title()` string method capitalizes the first letter of the string in `myMode` and splices the string into the `'%sing'` string, so it displays either `'Encrypting...'` or `'Decrypting...'`.

Measuring the Time It Took to Encrypt or Decrypt

Encrypting or decrypting an entire file can take much longer than a short string. A user might want to know how long the process takes for a file. We can measure the length of the encryption or decryption process by using the `time` module.

The time Module and time.time() Function

The `time.time()` function returns the current time as a float value of the number of seconds since January 1, 1970. This moment is called the *Unix Epoch*. Enter the following into the interactive shell to see how this function works:

```
>>> import time
>>> time.time()
1540944000.7197928
>>> time.time()
1540944003.4817972
```

Because `time.time()` returns a float value, it can be precise to a *millisecond* (that is, 1/1000 of a second). Of course, the numbers that

time.time() displays depend on the moment in time that you call this function and may be difficult to interpret. It might not be clear that 1540944000.7197928 is Tuesday, October 30, 2018, at approximately 5 PM. However, the time.time() function is useful for comparing the number of seconds between calls to time.time(). We can use this function to determine how long a program has been running.

For example, if you subtract the floating-point values returned when I called time.time() previously in the interactive shell, you would get the amount of time in between those calls while I was typing:

```
>>> 1540944003.4817972 - 1540944000.7197928
2.7620043754577637
```

If you need to write code that handles dates and times, see *https://www.nostarch.com/crackingcodes/* for information on the datetime module.

Using the time.time() Function in the Program

On line 34, time.time() returns the current time to store in a variable named startTime. Lines 35 to 38 call encryptMessage() or decryptMessage(), depending on whether 'encrypt' or 'decrypt' is stored in the myMode variable.

```
33.     # Measure how long the encryption/decryption takes:
34.     startTime = time.time()
35.     if myMode == 'encrypt':
36.         translated = transpositionEncrypt.encryptMessage(myKey, content)
37.     elif myMode == 'decrypt':
38.         translated = transpositionDecrypt.decryptMessage(myKey, content)
39.     totalTime = round(time.time() - startTime, 2)
40.     print('%sion time: %s seconds' % (myMode.title(), totalTime))
```

Line 39 calls time.time() again after the program decrypts or encrypts and subtracts startTime from the current time. The result is the number of seconds between the two calls to time.time(). The time.time() - startTime expression evaluates to a value that is passed to the round() function, which rounds to the nearest two decimal points, because we don't need millisecond precision for the program. This value is stored in totalTime. Line 40 uses string splicing to print the program mode and displays to the user the amount of time it took for the program to encrypt or decrypt.

Writing the Output File

The encrypted (or decrypted) file contents are now stored in the translated variable. But this string is forgotten when the program terminates, so we want to store the string in a file to have even after the program has finished running. The code on lines 43 to 45 does this by

opening a new file (and passing 'w' to the open() function) and then calling the write() file object method:

```
42.     # Write out the translated message to the output file:
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
```

Then, lines 47 and 48 print more messages to the user indicating that the process is done and the name of the written file:

```
47.     print('Done %sing %s (%s characters).' % (myMode, inputFilename,
          len(content)))
48.     print('%sed file is %s.' % (myMode.title(), outputFilename))
```

Line 48 is the last line of the main() function.

Calling the main() Function

Lines 53 and 54 (which are executed after the def statement on line 6 is executed) call the main() function if this program is being run instead of being imported:

```
51. # If transpositionCipherFile.py is run (instead of imported as a module),
52. # call the main() function:
53. if __name__ == '__main__':
54.     main()
```

This is explained in detail in "The __name__ Variable" on page 95.

Summary

Congratulations! There wasn't much to the *transpositionFileCipher.py* program aside from the open(), read(), write(), and close() functions, which let us encrypt large text files on a hard drive. You learned how to use the os.path.exists() function to check whether a file already exists. As you've seen, we can extend our programs' capabilities by importing their functions for use in new programs. This greatly increases our ability to use computers to encrypt information.

You also learned some useful string methods to make a program more flexible when accepting user input and how to use the time module to measure how fast your program runs.

Unlike the Caesar cipher program, the transposition file cipher has too many possible keys to attack by simply using brute force. But if we can write a program that recognizes English (as opposed to strings of gibberish), the computer could examine the output of thousands of decryption attempts and determine which key can successfully decrypt a message to English. You'll learn how to do this in Chapter 11.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Which is correct: `os.exists()` or `os.path.exists()`?
2. When is the Unix Epoch?
3. What do the following expressions evaluate to?

```
'Foobar'.startswith('Foo')
'Foo'.startswith('Foobar')
'Foobar'.startswith('foo')
'bar'.endswith('Foobar')
'Foobar'.endswith('bar')
'The quick brown fox jumped over the yellow lazy dog.'.title()
```

11

DETECTING ENGLISH PROGRAMMATICALLY

The gaffer says something longer and more complicated. After a while, Waterhouse (now wearing his cryptoanalyst hat, searching for meaning midst apparent randomness, his neural circuits exploiting the redundancies in the signal) realizes that the man is speaking heavily accented English.
—Neal Stephenson, Cryptonomicon

Previously, we used the transposition file cipher to encrypt and decrypt entire files, but we haven't tried writing a brute-force program to hack the cipher yet. Messages encrypted with the transposition file cipher can have thousands of possible keys, which your computer can still easily brute-force, but you'd then have to look through thousands of decryptions to find the one correct plaintext. As you can imagine, this can be a big problem, but there is a work-around.

When the computer decrypts a message using the wrong key, the resulting string is garbage text instead of English text. We can program the computer to recognize when a decrypted message is English. That way, if the computer decrypts using the wrong key, it knows to go on and try the next possible key. Eventually, when the computer tries a key that decrypts to English text, it can stop and bring that key to your attention, sparing you from having to look through thousands of incorrect decryptions.

TOPICS COVERED IN THIS CHAPTER

- The dictionary data type
- The `split()` method
- The `None` value
- Divide-by-zero errors
- The `float()`, `int()`, and `str()` functions and integer division
- The `append()` list method
- Default arguments
- Calculating percentages

How Can a Computer Understand English?

A computer can't understand English, at least, not in the way that human beings understand English. Computers don't understand math, chess, or human rebellions either, any more than a clock understands lunchtime. Computers just execute instructions one after another. But these instructions can mimic complex behaviors to solve math problems, win at chess, or hunt down the future leaders of the human resistance.

Ideally, what we need to create is a Python function (let's call it the `isEnglish()` function) that we can pass a string to and get a return value of `True` if the string is English text or `False` if it's random gibberish. Let's look at some English text and some garbage text to see what patterns they might have:

```
Robots are your friends. Except for RX-686. She will try to eat you.
ai-pey  e. xrx ne augur iirl6 Rtiyt fhubE6d hrSei t8..ow eo.telyoosEs  t
```

Notice that the English text is made up of words that you would find in a dictionary, but the garbage text isn't. Because words are usually separated by spaces, one way of checking whether a message string is English is to split the message into smaller strings at each space and to check whether each substring is a dictionary word. To split the message strings into substrings, we can use the Python string method named `split()`, which checks where each word begins and ends by looking for spaces between characters. ("The `split()` Method" on page 150 covers this in more detail.) We can then compare each substring to each word in the dictionary using an `if` statement, as in the following code:

```
if word == 'aardvark' or word == 'abacus' or word == 'abandon' or word ==
'abandoned' or word == 'abbreviate' or word == 'abbreviation' or word ==
'abdomen' or ...
```

We *could* write code like that, but we probably wouldn't because it would be tedious to type it all out. Fortunately, we can use *English dictionary files*, which are text files that contain nearly every word in English. I'll provide you with a dictionary file to use, so we just need to write the `isEnglish()` function that checks whether the substrings in the message are in the dictionary file.

Not every word exists in our dictionary file. The dictionary file might be incomplete; for example, it might not have the word *aardvark*. There are also perfectly good decryptions that might have non-English words in them, such as *RX-686* in our example English sentence. The plaintext could also be in a different language, but we'll assume it's in English for now.

So the `isEnglish()` function won't be foolproof, but if *most* of the words in the string argument are English words, it's a good bet the string is English text. There's a very low probability that a ciphertext decrypted using the wrong key will decrypt to English.

You can download the dictionary file we'll use for this book (which has more than 45,000 words) from *https://www.nostarch.com/crackingcodes/*. The dictionary text file lists one word per line in uppercase. Open it, and you'll see something like this:

```
AARHUS
AARON
ABABA
ABACK
ABAFT
ABANDON
ABANDONED
ABANDONING
ABANDONMENT
ABANDONS
--snip--
```

Our `isEnglish()` function will split a decrypted string into individual substrings and check whether each substring exists as a word in the dictionary file. If a certain number of the substrings are English words, we'll identify that text as English. And if the text is in English, there's a good chance that we'll have successfully decrypted the ciphertext with the correct key.

Source Code for the Detect English Module

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and then save it as *detectEnglish.py*. Make sure *dictionary.txt* is in the same directory as *detectEnglish.py* or this code won't work. Press F5 to run the program.

detectEnglish.py

```
1. # Detect English module
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. # To use, type this code:
5. #   import detectEnglish
```

```
 6. #   detectEnglish.isEnglish(someString) # Returns True or False
 7. # (There must be a "dictionary.txt" file in this directory with all
 8. # English words in it, one word per line. You can download this from
 9. # https://www.nostarch.com/crackingcodes/.)
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower() + ' \t\n'
12.
13. def loadDictionary():
14.     dictionaryFile = open('dictionary.txt')
15.     englishWords = {}
16.     for word in dictionaryFile.read().split('\n'):
17.         englishWords[word] = None
18.     dictionaryFile.close()
19.     return englishWords
20.
21. ENGLISH_WORDS = loadDictionary()
22.
23.
24. def getEnglishCount(message):
25.     message = message.upper()
26.     message = removeNonLetters(message)
27.     possibleWords = message.split()
28.
29.     if possibleWords == []:
30.         return 0.0 # No words at all, so return 0.0
31.
32.     matches = 0
33.     for word in possibleWords:
34.         if word in ENGLISH_WORDS:
35.             matches += 1
36.     return float(matches) / len(possibleWords)
37.
38.
39. def removeNonLetters(message):
40.     lettersOnly = []
41.     for symbol in message:
42.         if symbol in LETTERS_AND_SPACE:
43.             lettersOnly.append(symbol)
44.     return ''.join(lettersOnly)
45.
46.
47. def isEnglish(message, wordPercentage=20, letterPercentage=85):
48.     # By default, 20% of the words must exist in the dictionary file, and
49.     # 85% of all the characters in the message must be letters or spaces
50.     # (not punctuation or numbers).
51.     wordsMatch = getEnglishCount(message) * 100 >= wordPercentage
52.     numLetters = len(removeNonLetters(message))
53.     messageLettersPercentage = float(numLetters) / len(message) * 100
54.     lettersMatch = messageLettersPercentage >= letterPercentage
55.     return wordsMatch and lettersMatch
```

Sample Run of the Detect English Module

The *detectEnglish.py* program we'll write in this chapter won't run by itself. Instead, other encryption programs will import *detectEnglish.py* so they can call the `detectEnglish.isEnglish()` function, which returns `True` when the string is determined to be English. This is why we don't give *detectEnglish.py* a `main()` function. The other functions in *detectEnglish.py* are helper functions that the `isEnglish()` function will call. All the work we'll do in this chapter will allow any program to import the `detectEnglish` module with an `import` statement and use the functions in it.

You'll also be able to use this module in the interactive shell to check whether an individual string is in English, as shown here:

```
>>> import detectEnglish
>>> detectEnglish.isEnglish('Is this sentence English text?')
True
```

In this example, the function determined that the string `'Is this sentence English text?'` is indeed in English, so it returns `True`.

Instructions and Setting Up Constants

Let's look at the first portion of the *detectEnglish.py* program. The first nine lines of code are comments that give instructions on how to use this module.

```
 1. # Detect English module
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. # To use, type this code:
 5. #   import detectEnglish
 6. #   detectEnglish.isEnglish(someString) # Returns True or False
 7. # (There must be a "dictionary.txt" file in this directory with all
 8. # English words in it, one word per line. You can download this from
 9. # https://www.nostarch.com/crackingcodes/.)
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower() + ' \t\n'
```

The first nine lines of code are comments that give instructions on how to use this module. They remind users that this module won't work unless a file named *dictionary.txt* is in the same directory as *detectEnglish.py*.

Lines 10 and 11 set up a few variables as constants, which are in uppercase. As you learned in Chapter 5, constants are variables whose values should never be changed after they're set. `UPPERLETTERS` is a constant containing the 26 uppercase letters that we set up for convenience and to save time typing. We use the `UPPERLETTERS` constant to set up `LETTERS_AND_SPACE`, which contains all the uppercase and lowercase letters of the alphabet as well as the space character, the tab character, and the newline character. Instead of typing out all the uppercase and lowercase letters, we just concatenate

UPPERLETTERS with the lowercase letters returned by UPPERLETTERS.lower() and the additional non-letter characters. The tab and newline characters are represented with the escape characters \t and \n.

The Dictionary Data Type

Before we continue with the rest of the *detectEnglish.py* code, you need to learn more about the dictionary data type to understand how to convert the text in the file into a string value. The *dictionary* data type (not to be confused with the dictionary file) stores values, which can contain multiple other values just as lists do. In lists, we use an integer index to retrieve items in the list, such as spam[42]. But for each item in the dictionary value, we instead use a key to retrieve a value. Although we can use only integers to retrieve items from a list, the key in a dictionary value can be an integer or a string, such as spam['hello'] or spam[42]. Dictionaries let us organize our program's data with more flexibility than lists and don't store items in any particular order. Instead of using square brackets as lists do, dictionaries use braces. For example, an empty dictionary looks like this {}.

NOTE *Keep in mind that dictionary files and dictionary values are completely different concepts that just happen to have similar names. A Python dictionary value can contain multiple other values. A dictionary file is a text file containing English words.*

A dictionary's items are typed as *key-value pairs*, in which the keys and values are separated by colons. Multiple key-value pairs are separated by commas. To retrieve values from a dictionary, use square brackets with the key between them, similar to when indexing with lists. To try retrieving values from a dictionary using keys, enter the following into the interactive shell:

```
>>> spam = {'key1': 'This is a value', 'key2': 42}
>>> spam['key1']
'This is a value'
```

First, we set up a dictionary called spam with two key-value pairs. We then access the value associated with the 'key1' string key, which is another string. As with lists, you can store all kinds of data types in your dictionaries.

Note that, as with lists, variables don't store dictionary values; instead, they store references to dictionaries. The following example code shows two variables with references to the same dictionary:

```
>>> spam = {'hello': 42}
>>> eggs = spam
>>> eggs['hello'] = 99
>>> eggs
{'hello': 99}
>>> spam
{'hello': 99}
```

The first line of code sets up another dictionary called spam, this time with only one key-value pair. You can see that it stores an integer value 42 associated with the 'hello' string key. The second line assigns that dictionary key-value pair to another variable called eggs. You can then use eggs to change the original dictionary value associated with the 'hello' string key to 99. Now both variables, eggs and spam, should return the same dictionary key-value pair with the updated value.

The Difference Between Dictionaries and Lists

Dictionaries are like lists in many ways, but there are a few important differences:

- Dictionary items are not in any order. There is no first or last item in a dictionary as there is in a list.
- You can't concatenate dictionaries with the + operator. If you want to add a new item, use indexing with a new key. For example, foo['a new key'] = 'a string'.
- Lists only have integer index values that range from 0 to the length of the list minus one, but dictionaries can use any key. If you have a dictionary stored in a variable spam, you can store a value in spam[3] without needing values for spam[0], spam[1], or spam[2].

Adding or Changing Items in a Dictionary

You can also add or change values in a dictionary by using the dictionary keys as indexes. Enter the following into the interactive shell to see how this works:

```
>>> spam = {42: 'hello'}
>>> print(spam[42])
hello
>>> spam[42] = 'goodbye'
>>> print(spam[42])
goodbye
```

This dictionary has an existing dictionary string value 'hello' associated with the key 42. We can reassign a new string value 'goodbye' to that key using spam[42] = 'goodbye'. Assigning a new value to an existing dictionary key overwrites the original value associated with that key. For example, when we try to access the dictionary with the key 42, we get the new value associated with it.

And just as lists can contain other lists, dictionaries can also contain other dictionaries (or lists). To see an example, enter the following into the interactive shell:

```
>>> foo = {'fizz': {'name': 'Al', 'age': 144}, 'moo':['a', 'brown', 'cow']}
>>> foo['fizz']
{'age': 144, 'name': 'Al'}
>>> foo['fizz']['name']
'Al'
```

```
>>> foo['moo']
['a', 'brown', 'cow']
>>> foo['moo'][1]
'brown'
```

This example code shows a dictionary (named `foo`) that contains two keys `'fizz'` and `'moo'`, each corresponding to a different value and data type. The `'fizz'` key holds another dictionary, and the `'moo'` key holds a list. (Remember that dictionary values don't keep their items in order. This is why `foo['fizz']` shows the key-value pairs in a different order from what you typed.) To retrieve a value from a dictionary nested within another dictionary, you first specify the key of the larger data set you want to access using square brackets, which is `'fizz'` in this example. Then you use square brackets again and enter the key `'name'` corresponding to the nested string value `'Al'` that you want to retrieve.

Using the len() Function with Dictionaries

The `len()` function shows you the number of items in a list or the number of characters in a string. It can also show you the number of items in a dictionary. Enter the following code into the interactive shell to see how to use the `len()` function to count items in a dictionary:

```
>>> spam = {}
>>> len(spam)
0
>>> spam['name'] = 'Al'
>>> spam['pet'] = 'Zophie the cat'
>>> spam['age'] = 89
>>> len(spam)
3
```

The first line of this example shows an empty dictionary called `spam`. The `len()` function correctly shows the length of this empty dictionary is `0`. However, after you introduce the following three values, `'Al'`, `'Zophie the cat'`, and `89`, into the dictionary, the `len()` function now returns `3` for the three key-value pairs you've just assigned to the variable.

Using the in Operator with Dictionaries

You can use the `in` operator to see whether a certain key exists in a dictionary. It's important to remember that the `in` operator checks keys, not values. To see this operator in action, enter the following into the interactive shell:

```
>>> eggs = {'foo': 'milk', 'bar': 'bread'}
>>> 'foo' in eggs
True
>>> 'milk' in eggs
False
```

```
>>> 'blah blah blah' in eggs
False
>>> 'blah blah blah' not in eggs
True
```

We set up a dictionary called `eggs` with some key-value pairs and then check which keys exist in the dictionary using the `in` operator. The key `'foo'` is a key in `eggs`, so `True` is returned. Whereas `'milk'` returns `False` because it is a value, not a key, `'blah blah blah'` evaluates to `False` because no such item exists in this dictionary. The `not in` operator works with dictionary values as well, which you can see in the last command.

Finding Items Is Faster with Dictionaries than with Lists

Imagine the following list and dictionary values in the interactive shell:

```
>>> listVal = ['spam', 'eggs', 'bacon']
>>> dictionaryVal = {'spam':0, 'eggs':0, 'bacon':0}
```

Python can evaluate the expression `'bacon'` in `dictionaryVal` a bit faster than `'bacon'` in `listVal`. This is because for a list, Python must start at the beginning of the list and then move through each item in order until it finds the search item. If the list is very large, Python must search through numerous items, a process that can take a lot of time.

But a dictionary, also called a *hash table*, directly translates where in the computer's memory the value for the key-value pair is stored, which is why a dictionary's items don't have an order. No matter how large the dictionary is, finding any item always takes the same amount of time.

This difference in speed is hardly noticeable when searching short lists and dictionaries. But our `detectEnglish` module will have tens of thousands of items, and the expression `word in ENGLISH_WORDS`, which we'll use in our code, will be evaluated many times when the `isEnglish()` function is called. Using dictionary values speeds up this process when handling a large number of items.

Using for Loops with Dictionaries

You can also iterate over the keys in a dictionary using `for` loops, just as you can iterate over the items in a list. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Al', 'age': 99}
>>> for k in spam:
...   print(k, spam[k])
...
Age 99
name Al
```

To use a `for` statement to iterate over the keys in a dictionary, start with the `for` keyword. Set the variable `k`, use the `in` keyword to specify that you want to loop over `spam` and end the statement with a colon. As you can see, entering `print(k, spam[k])` returns each key in the dictionary along with its corresponding value.

Implementing the Dictionary File

Now let's return to *detectEnglish.py* and set up the dictionary file. The dictionary file sits on the user's hard drive, but unless we load the text in this file as a string value, our Python code can't use it. We'll create a `loadDictionary()` helper function to do this:

```
13. def loadDictionary():
14.     dictionaryFile = open('dictionary.txt')
15.     englishWords = {}
```

First, we get the dictionary's file object by calling `open()` and passing the string of the filename `'dictionary.txt'`. Then we name the dictionary variable `englishWords` and set it to an empty dictionary.

We'll store all the words in the dictionary file (the file that stores the English words) in a dictionary value (the Python data type). The similar names are unfortunate, but the two are completely different. Even though we could have used a list to store the string values of each word in the dictionary file, we're using a dictionary instead because the `in` operator works faster on dictionaries than lists.

Next, you'll learn about the `split()` string method, which we'll use to split our dictionary file into substrings.

The split() Method

The `split()` string method takes one string and returns a list of several strings by splitting the passed string at each space. To see an example of how this works, enter the following into the interactive shell:

```
>>> 'My very energetic mother    just served us Nutella.'.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'Nutella.']
```

The result is a list of eight strings, one string for each of the words in the original string. Spaces are dropped from the items in the list, even if there is more than one space. You can pass an optional argument to the `split()` method to tell it to split on a different string other than a space. Enter the following into the interactive shell:

```
>>> 'helloXXXworldXXXhowXXXareXXyou?'.split('XXX')
['hello', 'world', 'how', 'areXXyou?']
```

Notice that the string doesn't have any spaces. Using `split('XXX')` splits the original string wherever `'XXX'` occurs, resulting in a list of four strings. The last part of the string, `'areXXyou?'`, isn't split because `'XX'` isn't the same as `'XXX'`.

Splitting the Dictionary File into Individual Words

Let's return to our source code in *detectEnglish.py* to see how we split the string in the dictionary file and store each word in a key.

```
16.     for word in dictionaryFile.read().split('\n'):
17.         englishWords[word] = None
```

Let's break down line 16. The `dictionaryFile` variable stores the file object of the opened file. The `dictionaryFile.read()` method call reads the entire file and returns it as one large string value. We then call the `split()` method on this long string and split on newline characters. Because the dictionary file has one word per line, splitting on newline characters returns a list value made up of each word in the dictionary file.

The `for` loop at the beginning of the line iterates over each word to store each one in a key. But we don't need values associated with the keys since we're using the dictionary data type, so we'll just store the `None` value for each key.

`None` is a type of value that you can assign to a variable to represent the lack of a value. Whereas the Boolean data type has only two values, the NoneType has only one value, `None`. It's always written without quotes and with a capital *N*.

For example, say you had a variable named `quizAnswer`, which holds a user's answer to a true-false pop quiz question. If the user skips a question and doesn't answer it, it makes the most sense to assign `quizAnswer` to `None` as a default value rather than to `True` or `False`. Otherwise, it might look like the user answered the question when they didn't. Likewise, function calls that exit by reaching the end of the function and not from a `return` statement evaluate to `None` because they don't return anything.

Line 17 uses the word that is being iterated over as a key in `englishWords` and stores `None` as a value for that key.

Returning the Dictionary Data

After the `for` loop finishes, the `englishWords` dictionary should have tens of thousands of keys in it. At this point, we close the file object because we're done reading from it, and then return `englishWords`:

```
18.     dictionaryFile.close()
19.     return englishWords
```

Then we call `loadDictionary()` and store the dictionary value it returns in a variable named `ENGLISH_WORDS`:

```
21. ENGLISH_WORDS = loadDictionary()
```

We want to call `loadDictionary()` before the rest of the code in the `detectEnglish` module, but Python must execute the `def` statement for `loadDictionary()` before we can call the function. This is the reason the assignment for `ENGLISH_WORDS` comes after the `loadDictionary()` function's code.

Counting the Number of English Words in message

Lines 24 through 27 of the program's code define the `getEnglishCount()` function, which takes a string argument and returns a float value indicating the ratio of recognized English words to total words. We'll represent the ratio as a value between `0.0` and `1.0`. A value of `0.0` means none of the words in `message` are English words, and `1.0` means all of the words in `message` are English words. Most likely, `getEnglishCount()` will return a float value between `0.0` and `1.0`. The `isEnglish()` function uses this return value to determine whether to evaluate as `True` or `False`.

```
24. def getEnglishCount(message):
25.     message = message.upper()
26.     message = removeNonLetters(message)
27.     possibleWords = message.split()
```

To code this function, first we create a list of individual word strings from the string in `message`. Line 25 converts the string to uppercase letters. Then line 26 removes the non-letter characters from the string, such as numbers and punctuation, by calling `removeNonLetters()`. (You'll see how this function works later.) Finally, the `split()` method on line 27 splits the string into individual words and stores them in a variable named `possibleWords`.

For example, if the string `'Hello there. How are you?'` is passed after calling `getEnglishCount()`, the value stored in `possibleWords` after lines 25 to 27 execute would be `['HELLO', 'THERE', 'HOW', 'ARE', 'YOU']`.

If the string in `message` is made up of integers, such as `'12345'`, the call to `removeNonLetters()` would return a blank string, which `split()` would be called on to return an empty list. In the program, an empty list is the equivalent of zero words being English, which could cause a divide-by-zero error.

Divide-by-Zero Errors

To return a float value between `0.0` and `1.0`, we divide the number of words in `possibleWords` recognized as English by the total number of words in `possibleWords`. Although this is mostly straightforward, we need to make sure `possibleWords` is not an empty list. If `possibleWords` is empty, it means the total number of words in `possibleWords` is `0`.

Because in mathematics dividing by zero has no meaning, dividing by zero in Python results in a divide-by-zero error. To see an example of this error, enter the following into the interactive shell:

```
>>> 42 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    42 / 0
ZeroDivisionError: division by zero
```

You can see that `42` divided by `0` results in a `ZeroDivisionError` and a message explaining what went wrong. To avoid a divide-by-zero error, we'll need to make sure the `possibleWords` list is never empty.

Line 29 checks whether `possibleWords` is an empty list, and line 30 returns `0.0` if no words are in the list.

```
29.     if possibleWords == []:
30.         return 0.0 # No words at all, so return 0.0
```

This check is necessary to avoid a divide-by-zero error.

Counting the English Word Matches

To produce the ratio of English words to total words, we'll divide the number of words in `possibleWords` that are recognized as English by the total number of words in `possibleWords`. To do this, we need to count the number of recognized English words in `possibleWords`. Line 32 sets the variable `matches` to `0`. Line 33 uses the `for` loop to iterate over each word in `possibleWords` and check whether the word exists in the `ENGLISH_WORDS` dictionary. If the word exists in the dictionary, the value in `matches` is incremented on line 35.

```
32.     matches = 0
33.     for word in possibleWords:
34.         if word in ENGLISH_WORDS:
35.             matches += 1
```

After the `for` loop is complete, the number of English words in the string is stored in the `matches` variable. Remember that we're relying on the dictionary file to be accurate and complete for the `detectEnglish` module to work correctly. If a word isn't in the dictionary text file, it won't be counted as English, even if it's a real word. Conversely, if a word is misspelled in the dictionary, words that aren't English might accidentally be counted as real words.

Right now, the number of the words in `possibleWords` that are recognized as English and the total number of words in `possibleWords` are represented by integers. To return a float value between `0.0` and `1.0` by dividing these two integers, we'll need to change one or the other into a float.

The float(), int(), and str() Functions and Integer Division

Let's look at how to change an integer into a float because the two values we'll need to divide to find the ratio are both integers. Python 3 always does regular division regardless of the value type, whereas Python 2 performs integer division when both values in the division operation are integers. Because users might use Python 2 to import *detectEnglish.py*, we'll need to pass at least one integer variable to `float()` to make sure a float is returned when doing division. Doing so ensures that regular division will be performed no matter which version of Python is used. This is an example of making the code *backward compatible* with previous versions.

Although we won't use them in this program, let's review some other functions that convert values into other data types. The `int()` function returns an integer version of its argument, and the `str()` function returns a string. To see how these functions work, enter the following into the interactive shell:

```
>>> float(42)
42.0
>>> int(42.0)
42
>>> int(42.7)
42
>>> int('42')
42
>>> str(42)
'42'
>>> str(42.7)
'42.7'
```

You can see that the `float()` function changes the integer `42` into a float value. The `int()` function can turn the floats `42.0` and `42.7` into integers by truncating their decimal values, or it can turn a string value `'42'` into an integer. The `str()` function changes numerical values into string values. These functions are helpful if you need a value's equivalent to be a different data type.

Finding the Ratio of English Words in the Message

To find the ratio of English words to total words, we divide the number of `matches` we found by the total number of `possibleWords`. Line 36 uses the `/` operator to divide these two numbers:

```
36.     return float(matches) / len(possibleWords)
```

After we pass the integer `matches` to the `float()` function, it returns a float version of that number, which we divide by the length of the `possibleWords` list.

The only way `return float(matches) / len(possibleWords)` would lead to a divide-by-zero error is if `len(possibleWords)` evaluated to `0`. The only way that

would be possible is if possibleWords were an empty list. However, lines 29 and 30 specifically check for this case and return 0.0 if the list is empty. If possibleWords were set to the empty list, the program execution would never get past line 30, so we can be confident that line 36 won't cause a ZeroDivisionError.

Removing Non-Letter Characters

Certain characters, such as numbers or punctuation marks, will cause our word detection to fail because words won't look exactly as they're spelled in our dictionary file. For example, if the last word in message is 'you.' and we didn't remove the period at the end of the string, it wouldn't be counted as an English word because 'you' wouldn't be spelled with a period in the dictionary file. To avoid such misinterpretation, numbers and punctuation marks need to be removed.

The previously explained getEnglishCount() function calls the function removeNonLetters() on a string to remove any numbers and punctuation characters from it.

```
39. def removeNonLetters(message):
40.     lettersOnly = []
41.     for symbol in message:
42.         if symbol in LETTERS_AND_SPACE:
43.             lettersOnly.append(symbol)
```

Line 40 creates a blank list called lettersOnly, and line 41 uses a for loop to loop over each character in the message argument. Next, the for loop checks whether the character exists in the string LETTERS_AND_SPACE. If the character is a number or punctuation mark, it won't exist in the LETTERS_AND_SPACE string and won't be added to the list. If the character does exist in the string, it's added to the end of the list using the append() method, which we'll look at next.

The append() List Method

When we add a value to the end of a list, we say we're *appending* the value to the list. This is done with lists so frequently in Python that there is an append() list method that takes a single argument to append to the end of the list. Enter the following into the interactive shell:

```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
```

After creating an empty list named `eggs`, we can enter `eggs.append('hovercraft')` to add the string value `'hovercraft'` to this list. Then when we enter `eggs`, it returns the only value stored in this list, which is `'hovercraft'`. If you use `append()` again to add `'eels'` to the end of the list, `eggs` now returns `'hovercraft'` followed by `'eels'`. Similarly, we can use the `append()` list method to add items to the `lettersOnly` list we created in our code earlier. This is what `lettersOnly.append(symbol)` on line 43 does in the `for` loop.

Creating a String of Letters

After finishing the `for` loop, `lettersOnly` should be a list of each letter and space character from the original `message` string. Because a list of one-character strings isn't useful for finding English words, line 44 joins the character strings in the `lettersOnly` list into one string and returns it:

```
44.     return ''.join(lettersOnly)
```

To concatenate the list elements in `lettersOnly` into one large string, we call the `join()` string method on a blank string `''`. This joins the strings in `lettersOnly` with a blank string between them. This string value is then returned as the `removeNonLetters()` function's return value.

Detecting English Words

When a message is decrypted with the wrong key, it will often produce far more non-letter and non-space characters than are found in a typical English message. Also, the words it produces will often be random and not found in a dictionary of English words. The `isEnglish()` function can check for both of these issues in a given string.

```
47. def isEnglish(message, wordPercentage=20, letterPercentage=85):
48.     # By default, 20% of the words must exist in the dictionary file, and
49.     # 85% of all the characters in the message must be letters or spaces
50.     # (not punctuation or numbers).
```

Line 47 sets up the `isEnglish()` function to accept a string argument and return a Boolean value of `True` when the string is English text and `False` when it's not. This function has three parameters: `message`, `wordPercentage=20`, and `letterPercentage=85`. The first parameter contains the string to be checked, and the second and third parameters set default percentages for words and letters, which the string must contain in order to be confirmed as English. (A *percentage* is a number between 0 and 100 that shows how much of something is proportional to the total number of those things.) We'll explore how to use default arguments and calculate percentages in the following sections.

Using Default Arguments

Sometimes a function will almost always have the same values passed to it when called. Instead of including these for every function call, you can specify a default argument in the function's `def` statement.

Line 47's `def` statement has three parameters, with default arguments of 20 and 85 provided for `wordPercentage` and `letterPercentage`, respectively. The `isEnglish()` function can be called with one to three arguments. If no arguments are passed for `wordPercentage` or `letterPercentage`, then the values assigned to these parameters will be their default arguments.

The default arguments define what percent of the `message` string needs to be made up of real English words for `isEnglish()` to determine that `message` is an English string and what percent of the `message` needs to be made up of letters or spaces instead of numbers or punctuation marks. For example, if `isEnglish()` is called with only one argument, the default arguments are used for the `wordPercentage` (the integer `20`) and `letterPercentage` (the integer `85`) parameters, which means 20 percent of the string needs to be made up of English words and 85 percent of the string needs to be made up of letters. These percentages work for detecting English in most cases, but you might want to try other argument combinations in specific cases when `isEnglish()` needs looser or more restrictive thresholds. In those situations, a program can just pass arguments for `wordPercentage` and `letterPercentage` instead of using the default arguments. Table 11-1 shows function calls to `isEnglish()` and what they're equivalent to.

Table 11-1: Function Calls with and without Default Arguments

Function call	Equivalent to
`isEnglish('Hello')`	`isEnglish('Hello', 20, 85)`
`isEnglish('Hello', 50)`	`isEnglish('Hello', 50, 85)`
`isEnglish('Hello', 50, 60)`	`isEnglish('Hello', 50, 60)`
`isEnglish('Hello', letterPercentage=60)`	`isEnglish('Hello', 20, 60)`

For instance, the third example in Table 11-1 shows that when the function is called with the second and third parameters specified, the program will use those arguments, not the default arguments.

Calculating Percentages

Once we know the percentages our program will use, we'll need to calculate the percentages for the `message` string. For example, the string value `'Hello cat MOOSE fsdkl ewpin'` has five "words," but only three are English. To calculate the percentage of English words in this string, you divide the number of English words by the total number of words and multiply the result by 100. The percentage of English words in `'Hello cat MOOSE fsdkl ewpin'` is `3 / 5 * 100`, which is 60 percent. Table 11-2 shows a few examples of calculated percentages.

Table 11-2: Calculating Percentages of English Words

Number of English words	Total number of words	Ratio of English words	× 100	=	Percentage
3	5	0.6	× 100	=	60
6	10	0.6	× 100	=	60
300	500	0.6	× 100	=	60
32	87	0.3678	× 100	=	36.78
87	87	1.0	× 100	=	100
0	10	0	× 100	=	0

The percentage will always be between 0 percent (meaning no words are English) and 100 percent (meaning all of the words are English). Our `isEnglish()` function will consider a string English if at least 20 percent of the words exist in the dictionary file and 85 percent of the characters in the string are letters or spaces. This means the message will still be detected as English even if the dictionary file isn't perfect or if some words in the message are something other than what we define as English words.

Line 51 calculates the percentage of recognized English words in `message` by passing `message` to `getEnglishCount()`, which does the division and returns a float between `0.0` and `1.0`:

```
51.     wordsMatch = getEnglishCount(message) * 100 >= wordPercentage
```

To get a percentage from this float, multiply it by `100`. If the resulting number is greater than or equal to the `wordPercentage` parameter, `True` is stored in `wordsMatch`. (Recall that the `>=` comparison operator evaluates expressions to a Boolean value.) Otherwise, `False` is stored in `wordsMatch`.

Lines 52 to 54 calculate the percentage of letter characters in the `message` string by dividing the number of letter characters by the total number of characters in `message`.

```
52.     numLetters = len(removeNonLetters(message))
53.     messageLettersPercentage = float(numLetters) / len(message) * 100
54.     lettersMatch = messageLettersPercentage >= letterPercentage
```

Earlier in the code, we wrote the `removeNonLetters()` function to find all the letter and space characters in a string, so we can just reuse it. Line 52 calls `removeNonLetters(message)` to get a string of just the letter and space characters in `message`. Passing this string to `len()` should return the total number of letter and space characters in `message`, which we store as an integer in the `numLetters` variable.

Line 53 determines the percentage of letters by getting a float version of the integer in `numLetters` and dividing it by `len(message)`. The return value of `len(message)` will be the total number of characters in `message`. As discussed

previously, the call to `float()` is made to make sure that line 53 performs regular division instead of integer division just in case the programmer who imports the `detectEnglish` module is running Python 2.

Line 54 checks whether the percentage in `messageLettersPercentage` is greater than or equal to the `letterPercentage` parameter. This expression evaluates to a Boolean value that is stored in `lettersMatch`.

We want `isEnglish()` to return `True` only if both the `wordsMatch` and `lettersMatch` variables contain `True`. Line 55 combines these values into an expression using the `and` operator:

```
55.     return wordsMatch and lettersMatch
```

If both the `wordsMatch` and `lettersMatch` variables are `True`, `isEnglish()` will declare the `message` is English and return `True`. Otherwise, `isEnglish()` will return `False`.

Summary

The transposition file cipher is an improvement over the Caesar cipher because it can have hundreds or thousands of possible keys for messages instead of just 26 different keys. Even though a computer has no problem decrypting a message with thousands of potential keys, we need to write code that can determine whether a decrypted string is valid English and therefore the original message.

In this chapter, we created an English-detecting program using a dictionary text file to create a dictionary data type. The dictionary data type is useful because it can contain multiple values just as a list does. However, unlike with a list, you can index values in a dictionary using string values as keys instead of only integers. Most of the tasks you can do with a list you can also do with a dictionary, such as passing it to `len()` or using the `in` and `not in` operators on it. However, the `in` operator executes on a very large dictionary value much faster than on a very large list. This proved particularly useful for us because our dictionary data contained thousands of values that we needed to sift through quickly.

This chapter also introduced the `split()` method, which can split strings into a list of strings, and the NoneType data type, which has only one value: `None`. This value is useful for representing a lack of a value.

You learned how to avoid divide-by-zero errors when using the `/` operator; convert values into other data types using the `int()`, `float()`, and `str()` functions; and use the `append()` list method to add a value to the end of a list.

When you define functions, you can give some of the parameters default arguments. If no argument is passed for these parameters when the function is called, the program uses the default argument value, which can be a useful shortcut in your programs. In Chapter 12, you'll learn to hack the transposition cipher using the English detection code!

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What does the following code print?

```
spam = {'name': 'Al'}
print(spam['name'])
```

2. What does this code print?

```
spam = {'eggs': 'bacon'}
print('bacon' in spam)
```

3. What `for` loop code would print the values in the following `spam` dictionary?

```
spam = {'name': 'Zophie', 'species':'cat', 'age':8}
```

4. What does the following line print?

```
print('Hello, world!'.split())
```

5. What will the following code print?

```
def spam(eggs=42):
    print(eggs)
spam()
spam('Hello')
```

6. What percentage of words in this sentence are valid English words?

```
"Whether it's flobulllar in the mind to quarfalog the slings and
arrows of outrageous guuuuuuuuur."
```

12

HACKING THE TRANSPOSITION CIPHER

"Ron Rivest, one of the inventors of RSA, thinks that restricting cryptography would be foolhardy: 'It is poor policy to clamp down indiscriminately on a technology just because some criminals might be able to use it to their advantage.'"
— *Simon Singh,* The Code Book

In this chapter, we'll use a brute-force approach to hack the transposition cipher. Of the thousands of keys that could possibly be associated with the transposition cipher, the correct key should be the only one that results in legible English. Using the *detectEnglish.py* module we wrote in Chapter 11, our transposition cipher hacker program will help us find the correct key.

TOPICS COVERED IN THIS CHAPTER

- Multiline strings with triple quotes
- The `strip()` string method

Source Code of the Transposition Cipher Hacker Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *transpositionHacker.py*. As with previous programs, make sure the *pyperclip.py* module, the *transpositionDecrypt.py* module (Chapter 8), and the *detectEnglish.py* module and *dictionary.txt* file (Chapter 11) are in the same directory as the *transpositionHacker.py* file. Then press F5 to run the program.

transpositionHacker.py

```
# Transposition Cipher Hacker
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import pyperclip, detectEnglish, transpositionDecrypt

def main():
    # You might want to copy & paste this text from the source code at
    # https://www.nostarch.com/crackingcodes/:
    myMessage = """AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e  enlh
      na  indeit n uhoretrm au ieu v er Ne2 gmanw,forwnlbsya apor tE.no
      euarisfatt  e mealefedhsppmgAnlnoe(c -or)alat r lw o eb  nglom,Ain
      one dtes ilhetcdba. t tg eturmudg,tfl1e1 v  nitiaicynhrCsaemie-sp
      ncgHt nie cetrgmnoa yc r,ieaa  toesa- e aOm82e1w shcnth  ekh
      gaecnpeutaaieetgn iodhso d ro hAe snrsfcegrt NCsLc b17m8aEheideikfr
      aBercaeu thllnrshicwsg etriebruaisss  d iorr."""

    hackedMessage = hackTransposition(myMessage)

    if hackedMessage == None:
        print('Failed to hack encryption.')
    else:
        print('Copying hacked message to clipboard:')
        print(hackedMessage)
        pyperclip.copy(hackedMessage)


def hackTransposition(message):
    print('Hacking...')

    # Python programs can be stopped at any time by pressing
    # Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux):
    print('(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux) to
      quit at any time.)')

    # Brute-force by looping through every possible key:
    for key in range(1, len(message)):
        print('Trying key #%s...' % (key))

        decryptedText = transpositionDecrypt.decryptMessage(key, message)

        if detectEnglish.isEnglish(decryptedText):
            # Ask user if this is the correct decryption:
            print()
```

```
            print('Possible encryption hack:')
            print('Key %s: %s' % (key, decryptedText[:100]))
            print()
            print('Enter D if done, anything else to continue hacking:')
            response = input('> ')

            if response.strip().upper().startswith('D'):
                return decryptedText

    return None

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

Sample Run of the Transposition Cipher Hacker Program

When you run the *transpositionHacker.py* program, the output should look like this:

```
Hacking...
(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux) to quit at any time.)
Trying key #1...
Trying key #2...
Trying key #3...
Trying key #4...
Trying key #5...
Trying key #6...
Possible encryption hack:
Key 6: Augusta Ada King-Noel, Countess of Lovelace (10 December 1815 - 27
November 1852) was an English mat
Enter D if done, anything else to continue hacking:
> D
Copying hacked message to clipboard:
Augusta Ada King-Noel, Countess of Lovelace (10 December 1815 - 27 November
1852) was an English mathematician and writer, chiefly known for her work on
Charles Babbage's early mechanical general-purpose computer, the Analytical
Engine. Her notes on the engine include what is recognised as the first
algorithm intended to be carried out by a machine. As a result, she is often
regarded as the first computer programmer.
```

After trying key #6, the program returns a snippet of the decrypted message for the user to confirm that it has found the right key. In this example, the message looks promising. When the user confirms the decryption is correct by entering `D`, the program returns the entire hacked message. You can see it's a biographical note about Ada Lovelace. (Her algorithm for calculating Bernoulli numbers, devised in 1842 and 1843, made her the first computer programmer.) If the decryption is a false positive, the user can press anything else, and the program will continue to try other keys.

Run the program again and skip the correct decryption by pressing anything other than D. The program assumes that it didn't find the correct decryption and continues its brute-force approach through the other possible keys.

```
--snip--
Trying key #417...
Trying key #418...
Trying key #419...
Failed to hack encryption.
```

Eventually, the program runs through all the possible keys and then gives up, informing the user that it was unable to hack the ciphertext.

Let's take a closer look at the source code to see how the program works.

Importing the Modules

The first few lines of the code tell the user what this program will do. Line 4 imports several modules that we've written or seen in previous chapters: *pyperclip.py*, *detectEnglish.py*, and *transpositionDecrypt.py*.

```
1. # Transposition Cipher Hacker
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, detectEnglish, transpositionDecrypt
```

The transposition cipher hacker program, containing approximately 50 lines of code, is fairly short because much of it exists in other programs that we're using as modules.

Multiline Strings with Triple Quotes

The myMessage variable stores the ciphertext we're trying to hack. Line 9 stores a string value that begins and ends with triple quotes. Notice that it's a very long string.

```
6. def main():
7.     # You might want to copy & paste this text from the source code at
8.     # https://www.nostarch.com/crackingcodes/:
9.     myMessage = """AaKoosoeDe5 b5sn ma reno ora'lhlrrceey e  enlh
         na  indeit n uhoretrm au ieu v er Ne2 gmanw,forwnlbsya apor tE.no
         euarisfatt  e mealefedhsppmgAnlnoe(c -or)alat r lw o eb  nglom,Ain
         one dtes ilhetcdba. t tg eturmudg,tfl1e1 v  nitiaicynhrCsaemie-sp
         ncgHt nie cetrgmnoa yc r,ieaa  toesa- e aOm82e1w shcnth  ekh
         gaecnpeutaaieetgn iodhso d ro hAe snrsfcegrt NCsLc b17m8aEheideikfr
         aBercaeu thllnrshicwsg etriebruaisss  d iorr."""
```

Triple quote strings are also called *multiline strings* because they span multiple lines and can contain line breaks within them. Multiline strings are

useful for putting large strings into a program's source code and because single and double quotes don't need to be escaped within them. To see an example of a multiline string, enter the following into the interactive shell:

```
>>> spam = """Dear Alice,
Why did you dress up my hamster in doll clothing?
I look at Mr. Fuzz and think, "I know this was Alice's doing."
Sincerely,
Brienne"""
>>> print(spam)
Dear Alice,
Why did you dress up my hamster in doll clothing?
I look at Mr. Fuzz and think, "I know this was Alice's doing."
Sincerely,
Brienne
```

Notice that this string value, like our ciphertext string, spans multiple lines. Everything after the opening triple quotes will be interpreted as part of the string until the program reaches the triple quotes ending it. You can make multiline strings using either three double-quote characters or three single-quote characters.

Displaying the Results of Hacking the Message

The ciphertext-hacking code exists inside the `hackTransposition()` function, which is called on line 11 and which we'll define on line 21. This function takes one string argument: the encrypted ciphertext message we're trying to hack. If the function can hack the ciphertext, it returns a string of the decrypted text. Otherwise, it returns the `None` value.

```
11.     hackedMessage = hackTransposition(myMessage)
12.
13.     if hackedMessage == None:
14.         print('Failed to hack encryption.')
15.     else:
16.         print('Copying hacked message to clipboard:')
17.         print(hackedMessage)
18.         pyperclip.copy(hackedMessage)
```

Line 11 calls the `hackTransposition()` function to return the hacked message if the attempt is successful or the `None` value if the attempt is unsuccessful, and it stores the returned value in `hackedMessage`.

Lines 13 and 14 tell the program what to do if the function is unable to hack the ciphertext. If `None` was stored in `hackedMessage`, the program lets the user know by printing that it was unable to break the encryption on the message.

The next four lines show what the program does if the function is able to hack the ciphertext. Line 17 prints the decrypted message, and line 18 copies it to the clipboard. However, for this code to work, we also need to define the `hackTransposition()` function, which we'll do next.

Getting the Hacked Message

The `hackTransposition()` function starts with a couple `print()` statements.

```
21. def hackTransposition(message):
22.     print('Hacking...')
23.
24.     # Python programs can be stopped at any time by pressing
25.     # Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux):
26.     print('(Press Ctrl-C (on Windows) or Ctrl-D (on macOS and Linux) to
          quit at any time.)')
```

Because the program can try many keys, the program displays a message telling the user that the hacking has started and that it might take a moment to finish the process. The `print()` call on line 26 tells the user to press CTRL-C (on Windows) or CTRL-D (on macOS and Linux) to exit the program at any point. You can actually press these keys to exit any running Python program.

The next couple lines tell the program which keys to loop through by specifying the range of possible keys for the transposition cipher:

```
28.     # Brute-force by looping through every possible key:
29.     for key in range(1, len(message)):
30.         print('Trying key #%s...' % (key))
```

The possible keys for the transposition cipher range between 1 and the length of the message. The `for` loop on line 29 runs the hacking part of the function with each of these keys. Line 30 uses string interpolation to print the key currently being tested using string interpolation to provide feedback to the user.

Using the `decryptMessage()` function in the *transpositionDecrypt.py* program that we've already written, line 32 gets the decrypted output from the current key being tested and stores it in the `decryptedText` variable:

```
32.         decryptedText = transpositionDecrypt.decryptMessage(key, message)
```

The decrypted output in `decryptedText` will be English only if the correct key was used. Otherwise, it will appear as garbage text.

Then the program passes the string in `decryptedText` to the `detectEnglish.isEnglish()` function we wrote in Chapter 11 and prints part of `decryptedText`, the `key` used, and instructions for the user:

```
34.         if detectEnglish.isEnglish(decryptedText):
35.             # Ask user if this is the correct decryption:
36.             print()
37.             print('Possible encryption hack:')
38.             print('Key %s: %s' % (key, decryptedText[:100]))
39.             print()
40.             print('Enter D if done, anything else to continue hacking:')
41.             response = input('> ')
```

But just because `detectEnglish.isEnglish()` returns `True` and moves the execution to line 35 doesn't mean the program has found the correct key. It could be a false positive, meaning the program detected some text as English that is actually garbage text. To make sure, line 38 gives a preview of the text so the user can confirm that the text is indeed English. It uses the slice `decryptedText[:100]` to print out the first 100 characters of `decryptedText`.

The program pauses when line 41 executes, waits for the user to enter either `D` or anything else, and then stores this input as a string in `response`.

The strip() String Method

When a program gives a user specific instructions but the user doesn't follow them exactly, an error results. When the *transpositionHacker.py* program prompts the user to enter `D` to confirm the hacked message, it means the program won't accept any input other than `D`. If a user enters an extra space or character along with `D`, the program won't accept it. Let's look at how to use the `strip()` string method to make the program accept other inputs as long as they're similar enough to `D`.

The `strip()` string method returns a version of the string with any whitespace characters at the beginning and end of the string stripped out. The *whitespace characters* are the space character, the tab character, and the newline character. Enter the following into the interactive shell to see how this works:

```
>>> '       Hello'.strip()
'Hello'
>>> 'Hello       '.strip()
'Hello'
>>> '      Hello World      '.strip()
'Hello World'
```

In this example, `strip()` removes the space characters at the beginning or the end of the first two strings. If a string like `'       Hello World       '` includes spaces at the beginning and end of the string, the method removes them from both sides but doesn't remove any spaces between other characters.

The `strip()` method can also have a string argument passed to it that tells the method to remove characters other than whitespace from the beginning and end of the string. To see an example, enter the following into the interactive shell:

```
>>> 'aaaaaHELLOaa'.strip('a')
'HELLO'
>>> 'ababaHELLObaba'.strip('ab')
'HELLO'
>>> 'abccabcbacbXYZabcXYZacccab'.strip('abc')
'XYZabcXYZ'
```

Notice that passing the string arguments 'a' and 'ab' removes these characters when they occur at the beginning or end of the string. However, strip() doesn't remove characters embedded in the middle of the string. As you can see in the third example, the string 'abc' remains in 'XYZabcXYZ'.

Applying the strip() String Method

Let's return to the source code in *transpositionHacker.py* to see how to apply strip() in the program. Line 43 sets a condition using the if statement to give the user some input flexibility:

```
43.             if response.strip().upper().startswith('D'):
44.                 return decryptedText
```

If the condition for the statement were simply response == 'D', the user would have to enter D exactly and nothing else to end the program. For example, if the user enters 'd', ' D', or 'Done', the condition would be False and the program would continue checking other keys instead of returning the hacked message.

To avoid this issue, the string in response removes whitespace from the start or end of the string with the call to strip(). Then the string that response.strip() evaluates to has the upper() method called on it. Whether the user enters 'd' or 'D', the string returned from upper() will always be capitalized as 'D'. Adding flexibility in the type of input the program can accept makes it easier to use.

To make the program accept user input that starts with 'D' but is a full word, we use startswith() to check only the first letter. For example, if the user inputs ' done' as response, the whitespace would be stripped and then the string 'done' would be passed to upper(). After upper() capitalizes the whole string to 'DONE', the string is passed to startswith(), which returns True because the string does start with the substring 'D'.

If the user indicates that the decrypted string is correct, the function hackTransposition() on line 44 returns the decrypted text.

Failing to Hack the Message

Line 46 is the first line after the for loop that began on line 29:

```
46.     return None
```

If the program execution reaches this point, it means the program never reached the return statement on line 44, which would happen if the correctly decrypted text was never found for any of the keys that were tried. In that case, line 46 returns the None value to indicate that the hacking failed.

Calling the main() Function

Lines 48 and 49 call the `main()` function if this program was run by itself rather than being imported by another program using its `hackTransposition()` function:

```
48. if __name__ == '__main__':
49.     main()
```

Remember that the `__name__` variable is set by Python. The `main()` function will not be called if *transpositionHacker.py* is imported as a module.

Summary

Like Chapter 6, this chapter was short because most of the code was already written in other programs. Our hacking program can use functions from other programs by importing them as modules.

You learned how to use triple quotes to include a string value that spans multiple lines in the source code. You also learned that the `strip()` string method is useful for removing whitespace or other characters from the beginning or end of a string.

Using the *detectEnglish.py* program saved us a lot of time we would have had to spend manually inspecting every decrypted output to see if it was English. It allowed us to use the brute-force technique to hack a cipher that has thousands of possible keys.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What does this expression evaluate to?

```
'   Hello world'.strip()
```

2. Which characters are whitespace characters?
3. Why does `'Hello world'.strip('o')` evaluate to a string that still has *O*s in it?
4. Why does `'xxxHelloxxx'.strip('X')` evaluate to a string that still has *X*s in it?

13

A MODULAR ARITHMETIC MODULE FOR THE AFFINE CIPHER

"People have been defending their own privacy for centuries with whispers, darkness, envelopes, closed doors, secret handshakes, and couriers. The technologies of the past did not allow for strong privacy, but electronic technologies do."
—Eric Hughes, "A Cypherpunk's Manifesto" (1993)

In this chapter, you'll learn about the multiplicative cipher and the affine cipher. The multiplicative cipher is similar to the Caesar cipher but encrypts using multiplication rather than addition. The affine cipher combines the multiplicative cipher and the Caesar cipher, resulting in a stronger and more reliable encryption.

But first, you'll learn about modular arithmetic and greatest common divisors—two mathematical concepts that are required to understand and implement the affine cipher. Using these concepts, we'll create a module to handle wraparound and find valid keys for the affine cipher. We'll use this module when we create a program for the affine cipher in Chapter 14.

TOPICS COVERED IN THIS CHAPTER

- Modular arithmetic
- The modulo operator (%)
- The greatest common divisor (GCD)
- Multiple assignment
- Euclid's algorithm for finding the GCD
- The multiplicative and affine ciphers
- Euclid's extended algorithm for finding modular inverses

Modular Arithmetic

Modular arithmetic, or *clock arithmetic*, refers to math in which numbers wrap around when they reach a particular value. We'll use modular arithmetic to handle wraparound in the affine cipher. Let's see how it works.

Imagine a clock with just an hour hand and the 12 replaced with a 0. (If programmers designed clocks, the first hour would begin at 0.) If the current time is 3 o'clock, what time will it be in 5 hours? This is easy enough to figure out: 3 + 5 = 8. It will be 8 o'clock in 5 hours. Think of the hour hand starting at 3 and then moving 5 hours clockwise, as shown in Figure 13-1.

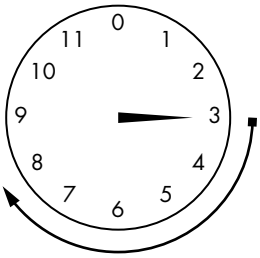


Figure 13-1: 3 o'clock + 5 hours = 8 o'clock

If the current time is 10 o'clock, what time will it be in 5 hours? Adding 5 + 10 = 15, but 15 o'clock doesn't make sense for clocks that show only 12 hours. To find out what time it will be, you subtract 15 – 12 = 3, so it will be 3 o'clock. (Normally, you would distinguish between 3 AM and 3 PM, but that doesn't matter in modular arithmetic.)

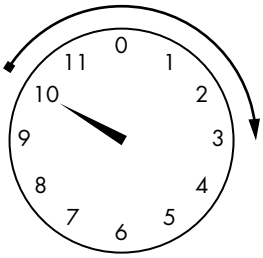


Figure 13-2: 10 o'clock + 5 hours = 3 o'clock

Double-check this math by moving the hour hand clockwise 5 hours, starting from 10. It does indeed land on 3, as shown in Figure 13-2.

If the current time is 10 o'clock, what time will it be in 200 hours? Adding 200 + 10 = 210, and 210 is certainly larger than 12. Because one full rotation brings the hour hand back to its original position, we can solve this problem by subtracting by 12 (which is one full rotation) until the result is a number less than 12. Subtracting 210 – 12 = 198. But 198 is still larger than 12, so we continue to subtract 12 until the difference is less than 12;

in this case the final answer will be 6. If the current time is 10 o'clock, the time 200 hours later will be 6 o'clock, as shown in Figure 13-3.

If you want to double-check the 10 o'clock + 200 hours math, you can repeatedly move the hour hand around the clock face. When you move the hour hand for the 200th hour, it should land on 6.

However, it's easier to have the computer do this modular arithmetic for us with the modulo operator.

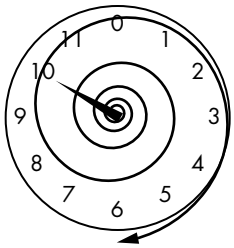


Figure 13-3: 10 o'clock + 200 hours = 6 o'clock

The Modulo Operator

You can use the *modulo operator*, abbreviated as *mod*, to write modular expressions. In Python, the mod operator is the percent sign (`%`). You can think of the mod operator as a kind of division remainder operator; for example, 21 ÷ 5 = 4 with a remainder of 1, and 21 % 5 = 1. Similarly, 15 % 12 is equal to 3, just as 15 o'clock would be 3 o'clock. Enter the following into the interactive shell to see the mod operator in action:

```
>>> 21 % 5
1
>>> (10 + 200) % 12
6
>>> 10 % 10
0
>>> 20 % 10
0
```

Just as 10 o'clock plus 200 hours will wrap around to 6 o'clock on a clock with 12 hours, `(10 + 200) % 12` will evaluate to `6`. Notice that numbers that divide evenly will mod to `0`, such as `10 % 10` or `20 % 10`.

Later, we'll use the mod operator to handle wraparound in the affine cipher. It's also used in the algorithm that we'll use to find the greatest common divisor of two numbers, which will enable us to find valid keys for the affine cipher.

Finding Factors to Calculate the Greatest Common Divisor

Factors are the numbers that are multiplied to produce a particular number. Consider 4 × 6 = 24. In this equation, 4 and 6 are factors of 24. Because a number's factors can also be used to divide that number without leaving a remainder, factors are also called *divisors*.

The number 24 also has some other factors:

$$8 \times 3 = 24$$

$$12 \times 2 = 24$$

$$24 \times 1 = 24$$

So the factors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.
Let's look at the factors of 30:

$$1 \times 30 = 30$$

$$2 \times 15 = 30$$

$$3 \times 10 = 30$$

$$5 \times 6 = 30$$

The factors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30. Note that any number will always have 1 and itself as its factors because 1 times a number is equal to that number. Notice too that the list of factors for 24 and 30 have 1, 2, 3, and 6 in common. The greatest of these common factors is 6, so 6 is the *greatest common factor*, more commonly known as the *greatest common divisor (GCD)*, of 24 and 30.

It's easiest to find a GCD of two numbers by visualizing their factors. We'll visualize factors and the GCD using *Cuisenaire rods*. A Cuisenaire rod is made up of squares equal to the number the rod represents, and the rods help us visualize math operations. Figure 13-4 uses Cuisenaire rods to visualize 3 + 2 = 5 and 5 × 3 = 15.

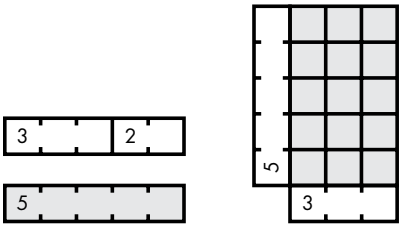


Figure 13-4: Using Cuisenaire rods to demonstrate addition and multiplication

A rod of 3 added to a rod of 2 is the same length as a rod of 5. You can even use rods to find answers to multiplication problems by making a rectangle with sides made from rods of the numbers you want to multiply. The number of squares in the rectangle is the answer to the multiplication problem.

If a rod 20 units long represents the number 20, a number is a factor of 20 if that number's rods can evenly fit inside the 20-square rod. Figure 13-5 shows that 4 and 10 are factors of 20 because they fit evenly into 20.

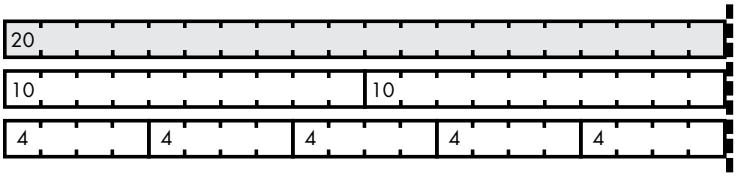


Figure 13-5: Cuisenaire rods demonstrating 4 and 10 are factors of 20

But 6 and 7 are not factors of 20, because the 6-square and 7-square rods won't evenly fit into the 20-square rod, as shown in Figure 13-6.

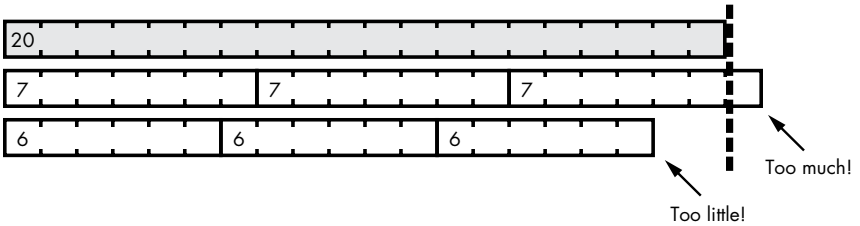


Figure 13-6: Cuisenaire rods demonstrating 6 and 7 are not factors of 20

The GCD of two rods, or two numbers represented by those rods, is the *longest* rod that can evenly fit into *both* rods, as shown in Figure 13-7.

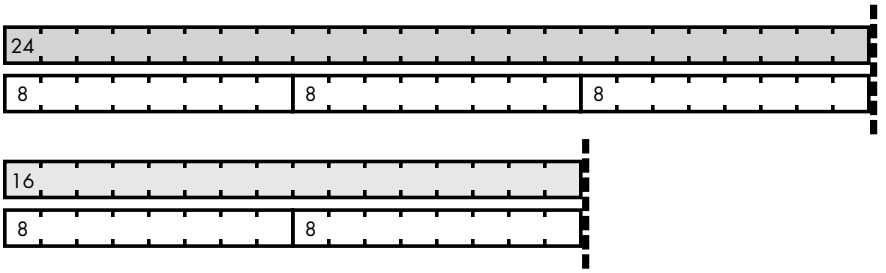


Figure 13-7: Cuisenaire rods demonstrating the GCD of 16 and 24

In this example, the 8-square rod is the longest rod that can fit evenly into 24 and 32. Therefore, 8 is their GCD.

Now that you know how factors and the GCD work, let's find the GCD of two numbers using a function we can write in Python.

Multiple Assignment

The `gcd()` function we'll write finds the GCD of two numbers. But before you learn how to code it, let's look at a trick in Python called *multiple assignment*. The multiple assignment trick lets you assign values to more than one variable at once in a single assignment statement. Enter the following into the interactive shell to see how this works:

```
>>> spam, eggs = 42, 'Hello'
>>> spam
42
>>> eggs
'Hello'
>>> a, b, c, d = ['Alice', 'Brienne', 'Carol', 'Danielle']
>>> a
'Alice'
>>> d
'Danielle'
```

You can separate the variable names on the left side of the = operator as well as the values on the right side of the = operator using commas. You can also assign each of the values in a list to its own variable as long as the number of items in the list is the same as the number of variables on the left side of the = operator. If you don't have the same number of variables as you have values, Python will raise an error that indicates the call needs more or has too many values.

One of the main uses of multiple assignment is to swap the values in two variables. Enter the following into the interactive shell to see an example:

```
>>> spam = 'hello'
>>> eggs = 'goodbye'
>>> spam, eggs = eggs, spam
>>> spam
'goodbye'
>>> eggs
'hello'
```

After assigning `'hello'` to `spam` and `'goodbye'` to `eggs`, we swap those values using multiple assignment. Let's look at how to use this swapping trick to implement Euclid's algorithm for finding the GCD.

Euclid's Algorithm for Finding the GCD

Finding the GCD seems simple enough: identify all the factors of the two numbers you'll use and then find the largest factor they have in common. But it isn't so easy to find the GCD of larger numbers.

Euclid, a mathematician who lived 2000 years ago, came up with a short algorithm for finding the GCD of two numbers using modular arithmetic. Here's a `gcd()` function that implements his algorithm in Python code, returning the GCD of integers `a` and `b`:

```
def gcd(a, b):
    while a != 0:
        a, b = b % a, a
    return b
```

The `gcd()` function takes two numbers `a` and `b`, and then uses a loop and multiple assignment to find the GCD. Figure 13-8 shows how the `gcd()` function finds the GCD of 24 and 32.

Exactly how Euclid's algorithm works is beyond the scope of this book, but you can rely on this function to return the GCD of the two integers you pass it. If you call this function from the interactive shell and pass it `24` and `32` for the parameters `a` and `b`, the function will return `8`:

```
>>> gcd(24, 32)
8
```

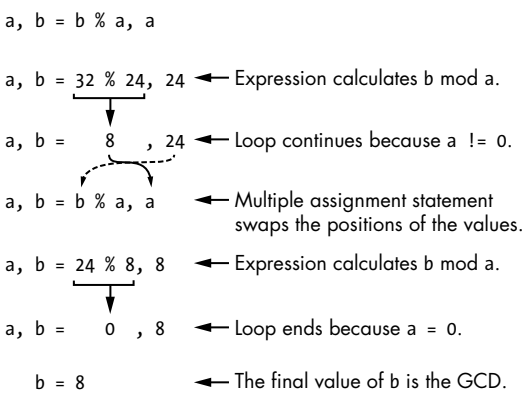


Figure 13-8: How the gcd() function works

The great benefit of this `gcd()` function, though, is that it can easily handle large numbers:

```
>>> gcd(409119243, 87780243)
6837
```

This `gcd()` function will come in handy when choosing valid keys for the multiplicative and affine ciphers, as you'll learn in the next section.

Understanding How the Multiplicative and Affine Ciphers Work

In the Caesar cipher, encrypting and decrypting symbols involved converting them to numbers, adding or subtracting the key, and then converting the new number back to a symbol.

When encrypting with the *multiplicative cipher*, you'll *multiply* the index by the key. For example, if you encrypted the letter E with the key 3, you would find E's index (4) and multiply it by the key (3) to get the index of the encrypted letter ($4 \times 3 = 12$), which would be M.

When the product exceeds the total number of letters, the multiplicative cipher has a wraparound issue similar to the Caesar cipher, but now we can use the mod operator to solve that issue. For example, the Caesar cipher's `SYMBOLS` variable contained the string `'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'`. The following is a table of the first and last few characters of `SYMBOLS` along with their indexes:

0	1	2	3	4	5	6	...	59	60	61	62	63	64	65
A	B	C	D	E	F	G	...	8	9	0		!	?	.

Let's calculate what these symbols encrypt to when the key is 17. To encrypt the symbol F with key 17, multiply its index 5 by 17 and mod the result by 66 to handle the wraparound of the 66-symbol set. The result of (5 × 17) mod 66 is 19, and 19 corresponds to the symbol T. So F encrypts to T in the multiplicative cipher with key 17. The following two strings show all the characters in plaintext and their corresponding ciphertext symbols. The symbol at a given index in the first string encrypts to the symbol at that same index in the second string:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'ARizCTk2EVm4GXo6IZq8KbsOMdu!Ofw.QhyBSj1DUl3FWn5HYp7Jar9Lct Nev?Pgx'
```

Compare this encryption output to the one you'd get when you encrypt using the Caesar cipher, which simply shifts the plaintext symbols over to create the ciphertext symbols:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'RSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.ABCDEFGHIJKLMNOPQ'
```

As you can see, the multiplicative cipher with key 17 results in ciphertext that is more randomized and harder to crack. However, you'll need to be careful when choosing keys for the multiplicative ciphers. I'll discuss why next.

Choosing Valid Multiplicative Keys

You can't just use any number for the multiplicative cipher's key. For example, if you chose the key 11, here's the mapping you would end up with:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
'ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4ALWhs4'
```

Notice that this key doesn't work because the symbols A, G, and M all encrypt to the same letter, A. When you encounter an A in the ciphertext, you wouldn't know which symbol it decrypts to. Using this key, you would run into the same problem when encrypting letters A, N, F, S, and others.

In the multiplicative cipher, the key and the size of the symbol set must be relatively prime to each other. Two numbers are *relatively prime* (or *coprime*) if their GCD is 1. In other words, they have no factors in common except 1. For example, the numbers `num1` and `num2` are relatively prime if `gcd(num1, num2) == 1`, where `num1` is the key and `num2` is the size of the symbol set. In the previous example, because 11 (the key) and 66 (the symbol set size) have a GCD that isn't 1, they are not relatively prime, which means that the key 11 cannot be used for the multiplicative cipher. Note that numbers don't actually have to be prime numbers to be relatively prime to each other.

Knowing how to use modular arithmetic and the `gcd()` function is important when using the multiplicative cipher. You can use the `gcd()` function to figure out whether a pair of numbers is relatively prime, which you need to know to choose valid keys for the multiplicative cipher.

The multiplicative cipher has only 20 different keys for a set of 66 symbols, even fewer than the Caesar cipher! However, you can combine the multiplicative cipher and the Caesar cipher to get the more powerful affine cipher, which I explain next.

Encrypting with the Affine Cipher

One downside to using the multiplicative cipher is that the letter *A* always maps to the letter *A*. The reason is that *A*'s number is 0, and 0 multiplied by anything will always be 0. You can fix this issue by adding a second key to perform a Caesar cipher encryption after the multiplicative cipher's multiplication and modding is done. This extra step changes the multiplicative cipher into the *affine cipher*.

The affine cipher has two keys: Key A and Key B. Key A is the integer you use to multiply the letter's number. After you multiply the plaintext by Key A, you add Key B to the product. Then you mod the sum by 66, as you did in the original Caesar cipher. This means the affine cipher has 66 times as many possible keys as the multiplicative cipher. It also ensures that the letter A doesn't always encrypt to itself.

The decryption process for the affine cipher mirrors the encryption process; both are shown in Figure 13-9.

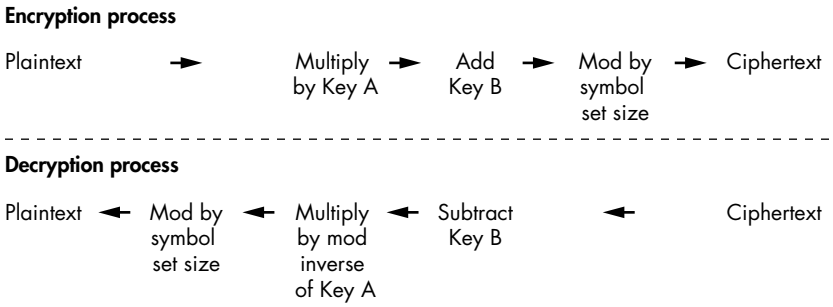


Figure 13-9: The affine cipher's encryption and decryption processes

We decrypt the affine cipher using the opposite operations used for encryption. Let's look at the decryption process and how to calculate the modular inverse in more detail.

Decrypting with the Affine Cipher

In the Caesar cipher, you used addition to encrypt and subtraction to decrypt. In the affine cipher, you use multiplication to encrypt. Naturally, you might think you can divide to decrypt with the affine cipher. But if you

try this, you'll see that it doesn't work. To decrypt with the affine cipher, you need to multiply by the key's modular inverse. This reverses the mod operation from the encryption process.

A *modular inverse* of two numbers is represented by the expression `(a * i) % m == 1`, where `i` is the modular inverse and `a` and `m` are the two numbers. For example, the modular inverse of 5 mod 7 would be some number `i` where `(5 * i) % 7` is equal to 1. You can brute-force this calculation like this:

1 isn't the modular inverse of 5 mod 7, because (5 * 1) % 7 = 5.

2 isn't the modular inverse of 5 mod 7, because (5 * 2) % 7 = 3.

3 is the modular inverse of 5 mod 7, because (5 * 3) % 7 = 1.

Although the encryption and decryption keys for the Caesar cipher part of the affine cipher are the same, the encryption key and decryption keys for the multiplicative cipher are two different numbers. The encryption key can be anything you choose as long as it's relatively prime to the size of the symbol set, which in this case is 66. If you choose the key 53 for encrypting with the affine cipher, the decryption key is the modular inverse of 53 mod 66:

1 isn't the modular inverse of 53 mod 66, because (53 * 1) % 66 = 53.

2 isn't the modular inverse of 53 mod 66, because (53 * 2) % 66 = 40.

3 isn't the modular inverse of 53 mod 66, because (53 * 3) % 66 = 27.

4 isn't the modular inverse of 53 mod 66, because (53 * 4) % 66 = 14.

5 is the modular inverse of 53 mod 66, because (53 * 5) % 66 = 1.

Because 5 is the modular inverse of 53 and 66, you know that the affine cipher decryption key is also 5. To decrypt a ciphertext letter, multiply that letter's number by 5 and then mod 66. The result is the number of the original plaintext's letter.

Using the 66-character symbol set, let's encrypt the word *Cat* using the key `53`. *C* is at index `2`, and `2 * 53` is `106`, which is larger than the symbol set size, so we mod `106` by `66`, and the result is `40`. The character at index `40` in the symbol set is `'o'`, so the symbol *C* encrypts to *o*.

We'll use the same steps for the next letter, *a*. The string `'a'` is at index `26` in the symbol set, and `26 * 53 % 66` is `58`, which is the index of `'7'`. So the symbol *a* encrypts to *7*. The string `'t'` is at index `45`, and `45 * 53 % 66` is `9`, which is the index of `'J'`. Therefore, the word *Cat* encrypts to *o7J*.

To decrypt, we multiply by the modular inverse of `53 % 66`, which is `5`. The symbol *o* is at index `40`, and `40 * 5 % 66` is `2`, which is the index of `'C'`. The symbol *7* is at index `58`, and `58 * 5 % 66` is `26`, which is the index of `'a'`. The symbol *J* is at index `9`, and `9 * 5 % 66` is `45`, which is the index of `'t'`. The ciphertext *o7J* decrypts to *Cat*, which is the original plaintext, just as expected.

Finding Modular Inverses

To calculate the modular inverse to determine the decryption key, you could take a brute-force approach and start testing the integer 1, and then 2, and then 3, and so on. But this is time-consuming for large keys such as 8,953,851.

Fortunately, you can use Euclid's extended algorithm to find the modular inverse of a number, which in Python looks like this:

```
def findModInverse(a, m):
    if gcd(a, m) != 1:
        return None  # No mod inverse if a & m aren't relatively prime.
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3  # Note that // is the integer division operator.
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3),
          v1, v2, v3
    return u1 % m
```

You don't have to understand how Euclid's extended algorithm works to use the `findModInverse()` function. As long as the two arguments you pass to the `findModInverse()` function are relatively prime, `findModInverse()` will return the modular inverse of the `a` parameter.

You can learn more about how Euclid's extended algorithm works at *https://www.nostarch.com/crackingcodes/*.

The Integer Division Operator

You may have noticed the `//` operator used in the `findModInverse()` function in the preceding section. This is the *integer division operator*. It divides two numbers and rounds down to the nearest integer. Enter the following into the interactive shell to see how the `//` operator works:

```
>>> 41 / 7
5.857142857142857
>>> 41 // 7
5
>>> 10 // 5
2
```

Whereas `41 / 7` evaluates to `5.857142857142857`, using `41 // 7` evaluates to `5`. For division expressions that do not divide evenly, the `//` operator is useful for getting the whole number part of the answer (sometimes called the *quotient*), while the `%` operator gets the remainder. An expression that uses the `//` integer division operator always evaluates to an int, not a float. As you can see when evaluating `10 // 5`, the result is `2` instead of `2.0`.

Source Code for the Cryptomath Module

We'll use `gcd()` and `findModInverse()` in more cipher programs later in this book, so let's put both functions into a module. Open a new file editor window, enter the following code, and save the file as *cryptomath.py*:

cryptomath.py

```
 1. # Cryptomath Module
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. def gcd(a, b):
 5.     # Return the GCD of a and b using Euclid's algorithm:
 6.     while a != 0:
 7.         a, b = b % a, a
 8.     return b
 9.
10.
11. def findModInverse(a, m):
12.     # Return the modular inverse of a % m, which is
13.     # the number x such that a*x % m = 1.
14.
15.     if gcd(a, m) != 1:
16.         return None  # No mod inverse if a & m aren't relatively prime.
17.
18.     # Calculate using the extended Euclidean algorithm:
19.     u1, u2, u3 = 1, 0, a
20.     v1, v2, v3 = 0, 1, m
21.     while v3 != 0:
22.         q = u3 // v3  # Note that // is the integer division operator.
23.         v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2),
              (u3 - q * v3), v1, v2, v3
24.     return u1 % m
```

This program contains the `gcd()` function described earlier in this chapter and the `findModInverse()` function that implements Euclid's extended algorithm.

After importing the *cryptomath.py* module, you can try out these functions from the interactive shell. Enter the following into the interactive shell:

```
>>> import cryptomath
>>> cryptomath.gcd(24, 32)
8
>>> cryptomath.gcd(37, 41)
1
>>> cryptomath.findModInverse(7, 26)
15
>>> cryptomath.findModInverse(8953851, 26)
17
```

As you can see, you can call the `gcd()` function and the `findModInverse()` function to find the GCD or modular inverse of two numbers.

Summary

This chapter covered some useful math concepts. The `%` operator finds the remainder after dividing one number by another. The `gcd()` function returns the largest number that can evenly divide two numbers. If the GCD of two numbers is 1, you know that those numbers are relatively prime to each other. The most useful algorithm to find the GCD of two numbers is Euclid's algorithm.

Unlike the Caesar cipher, the affine cipher uses multiplication and addition instead of just addition to encrypt letters. However, not all numbers work as keys for the affine cipher. The key number and the size of the symbol set must be relatively prime to each other.

To decrypt with the affine cipher, you multiply the ciphertext's index by the modular inverse of the key. The modular inverse of `a % m` is a number `i` such that `(a * i) % m == 1`. You can use Euclid's extended algorithm to calculate modular inverses. Chapter 23's public key cipher also uses modular inverses.

Using the math concepts you learned in this chapter, you'll write a program for the affine cipher in Chapter 14. Because the multiplicative cipher is the same thing as the affine cipher using a Key B of `0`, you won't have a separate multiplicative cipher program. And because the multiplicative cipher is just a less secure version of the affine cipher, you shouldn't use it anyway.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What do the following expressions evaluate to?

```
17 % 1000
5 % 5
```

2. What is the GCD of 10 and 15?
3. What does `spam` contain after executing `spam, eggs = 'hello', 'world'`?
4. The GCD of 17 and 31 is 1. Are 17 and 31 relatively prime?
5. Why aren't 6 and 8 relatively prime?
6. What is the formula for the modular inverse of A mod C?

14

PROGRAMMING THE AFFINE CIPHER

"I should be able to whisper something in your ear, even if your ear is 1000 miles away, and the government disagrees with that."
—Philip Zimmermann, creator of Pretty Good Privacy (PGP), the most widely used email encryption software in the world

In Chapter 13, you learned that the affine cipher is actually the multiplicative cipher combined with the Caesar cipher (Chapter 5), and the multiplicative cipher is similar to the Caesar cipher except it uses multiplication instead of addition to encrypt messages. In this chapter, you'll build and run programs to implement the affine cipher. Because the affine cipher uses two different ciphers as part of its encryption process, it needs two keys: one for the multiplicative cipher and another for the Caesar cipher. For the affine cipher program, we'll split a single integer into two keys.

TOPICS COVERED IN THIS CHAPTER

- The tuple data type
- How many different keys can the affine cipher have?
- Generating random keys

Source Code for the Affine Cipher Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and then save it as *affineCipher.py*. Make sure the *pyperclip.py* module and the *cryptomath.py* module you made in Chapter 13 are in the same folder as the *affineCipher.py* file.

affineCipher.py

```
 1. # Affine Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import sys, pyperclip, cryptomath, random
 5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
 6.
 7.
 8. def main():
 9.     myMessage = """A computer would deserve to be called intelligent
          if it could deceive a human into believing that it was human."
          -Alan Turing"""
10.     myKey = 2894
11.     myMode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
12.
13.     if myMode == 'encrypt':
14.         translated = encryptMessage(myKey, myMessage)
15.     elif myMode == 'decrypt':
16.         translated = decryptMessage(myKey, myMessage)
17.     print('Key: %s' % (myKey))
18.     print('%sed text:' % (myMode.title()))
19.     print(translated)
20.     pyperclip.copy(translated)
21.     print('Full %sed text copied to clipboard.' % (myMode))
22.
23.
24. def getKeyParts(key):
25.     keyA = key // len(SYMBOLS)
26.     keyB = key % len(SYMBOLS)
27.     return (keyA, keyB)
28.
29.
30. def checkKeys(keyA, keyB, mode):
31.     if keyA == 1 and mode == 'encrypt':
32.         sys.exit('Cipher is weak if key A is 1. Choose a different key.')
```

```
    if keyB == 0 and mode == 'encrypt':
        sys.exit('Cipher is weak if key B is 0. Choose a different key.')
    if keyA < 0 or keyB < 0 or keyB > len(SYMBOLS) - 1:
        sys.exit('Key A must be greater than 0 and Key B must be
          between 0 and %s.' % (len(SYMBOLS) - 1))
    if cryptomath.gcd(keyA, len(SYMBOLS)) != 1:
        sys.exit('Key A (%s) and the symbol set size (%s) are not
          relatively prime. Choose a different key.' % (keyA,
          len(SYMBOLS)))


def encryptMessage(key, message):
    keyA, keyB = getKeyParts(key)
    checkKeys(keyA, keyB, 'encrypt')
    ciphertext = ''
    for symbol in message:
        if symbol in SYMBOLS:
            # Encrypt the symbol:
            symbolIndex = SYMBOLS.find(symbol)
            ciphertext += SYMBOLS[(symbolIndex * keyA + keyB) %
              len(SYMBOLS)]
        else:
            ciphertext += symbol # Append the symbol without encrypting.
    return ciphertext


def decryptMessage(key, message):
    keyA, keyB = getKeyParts(key)
    checkKeys(keyA, keyB, 'decrypt')
    plaintext = ''
    modInverseOfKeyA = cryptomath.findModInverse(keyA, len(SYMBOLS))

    for symbol in message:
        if symbol in SYMBOLS:
            # Decrypt the symbol:
            symbolIndex = SYMBOLS.find(symbol)
            plaintext += SYMBOLS[(symbolIndex - keyB) * modInverseOfKeyA %
              len(SYMBOLS)]
        else:
            plaintext += symbol # Append the symbol without decrypting.
    return plaintext


def getRandomKey():
    while True:
        keyA = random.randint(2, len(SYMBOLS))
        keyB = random.randint(2, len(SYMBOLS))
        if cryptomath.gcd(keyA, len(SYMBOLS)) == 1:
            return keyA * len(SYMBOLS) + keyB


# If affineCipher.py is run (instead of imported as a module), call
# the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Affine Cipher Program

From the file editor, press F5 to run the *affineCipher.py* program; the output should look like this:

```
Key: 2894
Encrypted text:
"5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!AuaRLQADQALQG93!xQxaGaAfaQ1QX3o1R
QARL9Qda!AafARuQLX1LQALQI1iQX3o1RN"Q-5!1RQP36ARuFull encrypted text copied to
clipboard.
```

In the affine cipher program, the message, `"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human." -Alan Turing`, gets encrypted with the key 2894 into ciphertext. To decrypt this ciphertext, you can copy and paste it as the new value to be stored in `myMessage` on line 9 and change `myMode` on line 13 to the string `'decrypt'`.

Setting Up Modules, Constants, and the main() Function

Lines 1 and 2 of the program are comments describing what the program is. There's also an `import` statement for the modules used in this program:

```
1. # Affine Cipher
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import sys, pyperclip, cryptomath, random
```

The four modules imported in this program serve the following functions:

- The `sys` module is imported for the `exit()` function.
- The `pyperclip` module is imported for the `copy()` clipboard function.
- The `cryptomath` module that you created in Chapter 13 is imported for the `gcd()` and `findModInverse()` functions.
- The `random` module is imported for the `random.randint()` function to generate random keys.

The string stored in the `SYMBOLS` variable is the symbol set, which is the list of all characters that can be encrypted:

```
5. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
     67890 !?.'
```

Any characters in the message that don't appear in `SYMBOLS` remain unencrypted in the ciphertext. For example, in the sample run of *affineCipher.py*, the quotation marks and the hyphen (`-`) don't get encrypted in the ciphertext because they don't belong in the symbol set.

Line 8 calls the `main()` function, which is almost exactly the same as the one in the transposition cipher programs. Lines 9, 10, and 11 store the message, key, and mode in variables, respectively:

```
 8. def main():
 9.     myMessage = """A computer would deserve to be called intelligent
          if it could deceive a human into believing that it was human."
          -Alan Turing"""
10.     myKey = 2894
11.     myMode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
```

The value stored in `myMode` determines whether the program encrypts or decrypts the message:

```
13.     if myMode == 'encrypt':
14.         translated = encryptMessage(myKey, myMessage)
15.     elif myMode == 'decrypt':
16.         translated = decryptMessage(myKey, myMessage)
```

If `myMode` is set to `'encrypt'`, line 14 executes and the return value of `encryptMessage()` is stored in `translated`. But if `myMode` is set to `'decrypt'`, `decryptMessage()` is called on line 16 and the return value is stored in `translated`. I'll cover how the `encryptMessage()` and `decryptMessage()` functions work when we define them later in the chapter.

After the execution passes line 16, the `translated` variable has the encrypted or decrypted version of the message in `myMessage`.

Line 17 displays the key used for the cipher using the `%s` placeholder, and line 18 tells the user whether the output is encrypted or decrypted text:

```
17.     print('Key: %s' % (myKey))
18.     print('%sed text:' % (myMode.title()))
19.     print(translated)
20.     pyperclip.copy(translated)
21.     print('Full %sed text copied to clipboard.' % (myMode))
```

Line 19 prints the string in `translated`, which is the encrypted or decrypted version of the string in `myMessage`, and line 20 copies it to the clipboard. Line 21 notifies the user that it is on the clipboard.

Calculating and Validating the Keys

Unlike the Caesar cipher, which uses addition with only one key, the affine cipher uses multiplication and addition with two integer keys, which we'll call Key A and Key B. Because it's easier to remember just one number, we'll use a mathematical trick to convert two keys into one key. Let's look at how this works in *affineCipher.py*.

The `getKeyParts()` function on line 24 splits a single integer key into two integers for Key A and Key B:

```
24. def getKeyParts(key):
25.     keyA = key // len(SYMBOLS)
26.     keyB = key % len(SYMBOLS)
27.     return (keyA, keyB)
```

The key to split is passed to the `key` parameter. On line 25, Key A is calculated by using integer division to divide `key` by `len(SYMBOLS)`, the size of the symbol set. Integer division (`//`) returns the quotient without a remainder. The mod operator (`%`) on line 26 calculates the remainder, which we'll use for Key B.

For example, with `2894` as the `key` parameter and a `SYMBOLS` string of 66 characters, Key A would be `2894 // 66 = 43` and Key B would be `2894 % 66 = 56`.

To combine Key A and Key B back into a single key, multiply Key A by the size of the symbol set and add Key B to the product: `(43 * 66) + 56` evaluates to `2894`, which is the integer key we started with.

NOTE *Keep in mind that according to Shannon's Maxim ("The enemy knows the system!") we must assume hackers know everything about the encryption algorithm, including the symbol set and the size of the symbol set. We assume that the only piece a hacker doesn't know is the key that was used. The security of our cipher program should depend only on the secrecy of the key, not the secrecy of the symbol set or the program's source code.*

The Tuple Data Type

Line 27 looks like it returns a list value, except parentheses are used instead of square brackets. This is a *tuple* value.

```
27.     return (keyA, keyB)
```

A tuple value is similar to a list value in that it can store other values, which can be accessed with indexes or slices. However, unlike list values, tuple values cannot be modified. There's no `append()` method for tuple values.

Because *affineCipher.py* doesn't need to modify the value returned by `getKeyParts()`, using a tuple is more appropriate than a list.

Checking for Weak Keys

Encrypting with the affine cipher involves a character's index in `SYMBOLS` being multiplied by Key A and added to Key B. But if `keyA` is `1`, the encrypted text is very weak because multiplying the index by `1` results in the same index. In fact, as defined by the multiplicative identity property, the product of any number and 1 is that number. Similarly, if `keyB` is `0`, the encrypted text is weak because adding `0` to the index doesn't change it. If `keyA` is `1` and `keyB` is `0` at the same time, the "encrypted" output would be identical to the original message. In other words, it wouldn't be encrypted at all!

We check for weak keys using the `checkKeys()` function on line 30. The `if` statements on lines 31 and 33 check whether `keyA` is `1` or `keyB` is `0`.

```
30. def checkKeys(keyA, keyB, mode):
31.     if keyA == 1 and mode == 'encrypt':
32.         sys.exit('Cipher is weak if key A is 1. Choose a different key.')
33.     if keyB == 0 and mode == 'encrypt':
34.         sys.exit('Cipher is weak if key B is 0. Choose a different key.')
```

If these conditions are met, the program exits with a message indicating what went wrong. Lines 32 and 34 each pass a string to the `sys.exit()` call. The `sys.exit()` function has an optional parameter that lets you print a string to the screen before terminating the program. You can use this function to display an error message on the screen before the program quits.

These checks prevent you from encrypting with weak keys, but if your `mode` is set to `'decrypt'`, the checks on lines 31 and 33 don't apply.

The condition on line 35 checks whether `keyA` is a negative number (that is, whether it's less than `0`) *or* whether `keyB` is greater than `0` *or* less than the size of the symbol set minus one:

```
35.     if keyA < 0 or keyB < 0 or keyB > len(SYMBOLS) - 1:
36.         sys.exit('Key A must be greater than 0 and Key B must be
              between 0 and %s.' % (len(SYMBOLS) - 1))
```

The reason the keys are in these ranges is described in the next section. If any of these conditions is `True`, the keys are invalid and the program exits.

Additionally, Key A must be relatively prime to the symbol set size. This means that the greatest common divisor (GCD) of `keyA` and `len(SYMBOLS)` must be equal to `1`. Line 37 checks for this using an `if` statement, and line 38 exits the program if the two values are not relatively prime:

```
37.     if cryptomath.gcd(keyA, len(SYMBOLS)) != 1:
38.         sys.exit('Key A (%s) and the symbol set size (%s) are not
              relatively prime. Choose a different key.' % (keyA,
              len(SYMBOLS)))
```

If all the conditions in the `checkKeys()` function return `False`, nothing is wrong with the key, and the program doesn't exit. Program execution returns to the line that originally called `checkKeys()`.

How Many Keys Can the Affine Cipher Have?

Let's try to calculate the number of possible keys the affine cipher has. The affine cipher's Key B is limited to the size of the symbol set, where `len(SYMBOLS)` is `66`. At first glance, it seems like Key A could be as large as you want it to be as long as it's relatively prime to the symbol set size. Therefore, you might think that the affine cipher has an infinite number of keys and cannot be brute-forced.

But this is not the case. Recall how large keys in the Caesar cipher ended up being the same as smaller keys due to the wraparound effect.

With a symbol set size of 66, the key 67 in the Caesar cipher would produce the same encrypted text as the key 1. The affine cipher also wraps around in this way.

Because the Key B part of the affine cipher is the same as the Caesar cipher, its range is limited from 1 to the size of the symbol set. To determine whether the affine cipher's Key A is also limited, we'll write a short program to encrypt a message using several different integers for Key A and see what the ciphertext looks like.

Open a new file editor window and enter the following source code. Save this file as *affineKeyTest.py* in the same folder as *affineCipher.py* and *cryptomath.py*. Then press F5 to run it.

affineKeyTest.py

```
1. # This program proves that the keyspace of the affine cipher is limited
2. # to less than len(SYMBOLS) ^ 2.
3.
4. import affineCipher, cryptomath
5.
6. message = 'Make things as simple as possible, but not simpler.'
7. for keyA in range(2, 80):
8.     key = keyA * len(affineCipher.SYMBOLS) + 1
9.
10.    if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) == 1:
11.        print(keyA, affineCipher.encryptMessage(key, message))
```

This program imports the `affineCipher` module for its `encryptMessage()` function and the `cryptomath` module for its `gcd()` function. We'll always encrypt the string stored in the `message` variable. The `for` loop remains in a range between `2` and `80`, because `0` and `1` are not allowed as valid Key A integers, as explained earlier.

On each iteration of the loop, line 8 calculates the key from the current `keyA` value and always uses `1` for Key B, which is why `1` is added at the end of line 8. Keep in mind that Key A must be relatively prime with the symbol set size to be valid. Key A is relatively prime with the symbol set size if the GCD of the key and the symbol set size is equal to `1`. So if the GCD of the key and the symbol set size is not equal to `1`, the `if` statement on line 10 will skip the call to `encryptMessage()` on line 11.

In short, this program prints the same message encrypted with several different integers for Key A. The output of this program looks like this:

```
5 O.xTvcin?dXv.XvXn8I3Tv.XvIDXXnE3T,vEhcv?DcvXn8I3TS
7 Tz4Nn1ipKbtnztntpDY NnztnYRttp7 N,n781nKR1ntpDY Nm9
13 ZJHOP7ivuVtPJtPtvhGUOPJtPG8ttvWUO,PWF7Pu87PtvhGUOg3
17 HvTx.oizERX.vX.Xz2mkx.vX.mVXXz?kx,.?6o.EVo.Xz2mkxGy
--snip--
67 Nblf!uijoht!bt!tjnqmf!bt!qpttjcmf,!cvu!opu!tjnqmfsA
71 O.xTvcin?dXv.XvXn8I3Tv.XvIDXXnE3T,vEhcv?DcvXn8I3TS
73 Tz4Nn1ipKbtnztntpDY NnztnYRttp7 N,n781nKR1ntpDY Nm9
79 ZJHOP7ivuVtPJtPtvhGUOPJtPG8ttvWUO,PWF7Pu87PtvhGUOg3
```

Look carefully at the output, and you'll notice that the ciphertext for Key A of 5 is the same as the ciphertext for Key A of 71! In fact, the ciphertext from keys 7 and 73 are the same, as are the ciphertext from keys 13 and 79!

Notice also that subtracting 5 from 71 results in 66, the size of our symbol set. This is why a Key A of 71 does the same thing as a Key A of 5: the encrypted output repeats itself, or wraps around, every 66 keys. As you can see, the affine cipher has the same wraparound effect for Key A as it does for Key B. In sum, Key A is also limited to the symbol set size.

When you multiply 66 possible Key A keys by 66 possible Key B keys, the result is 4356 possible combinations. Then when you subtract the integers that can't be used for Key A because they're not relatively prime with 66, the total number of possible key combinations for the affine cipher drops to 1320.

Writing the Encryption Function

To encrypt the message in *affineCipher.py*, we first need the `key` and the `message` to encrypt, which the `encryptMessage()` function takes as parameters:

```
41. def encryptMessage(key, message):
42.     keyA, keyB = getKeyParts(key)
43.     checkKeys(keyA, keyB, 'encrypt')
```

Then we need to get the integer values for Key A and Key B from the `getKeyParts()` function by passing it `key` on line 42. Next, we check whether these values are valid keys by passing them to the `checkKeys()` function. If the `checkKeys()` function doesn't cause the program to exit, the keys are valid and the rest of the code in the `encryptMessage()` function after line 43 can proceed.

On line 44, the `ciphertext` variable starts as a blank string but will eventually hold the encrypted string. The `for` loop that begins on line 45 iterates through each of the characters in `message` and then adds the encrypted character to `ciphertext`:

```
44.     ciphertext = ''
45.     for symbol in message:
```

By the time the `for` loop is done looping, the `ciphertext` variable will contain the complete string of the encrypted message.

On each iteration of the loop, the `symbol` variable is assigned a single character from `message`. If this character exists in `SYMBOLS`, which is our symbol set, the index in `SYMBOLS` is found and assigned to `symbolIndex` on line 48:

```
46.         if symbol in SYMBOLS:
47.             # Encrypt the symbol:
48.             symbolIndex = SYMBOLS.find(symbol)
49.             ciphertext += SYMBOLS[(symbolIndex * keyA + keyB) %
                  len(SYMBOLS)]
50.         else:
51.             ciphertext += symbol # Append the symbol without encrypting.
```

To encrypt the text, we need to calculate the index of the encrypted letter. Line 49 multiplies this `symbolIndex` by `keyA` and adds `keyB` to the product. Then it mods the result by the size of the symbol set, represented by the expression `len(SYMBOLS)`. Modding by `len(SYMBOLS)` handles the wraparound by ensuring the calculated index is always between `0` and up to, but not including, `len(SYMBOLS)`. The resulting number will be the index in `SYMBOLS` of the encrypted character, which is concatenated to the end of the string in `ciphertext`.

Everything in the previous paragraph is done on line 49, using a single line of code!

If `symbol` isn't in our symbol set, `symbol` is concatenated to the end of the `ciphertext` string on line 51. For example, the quotation marks and hyphen in the original message are not in the symbol set and therefore are concatenated to the string.

After the code has iterated through each character in the message string, the `ciphertext` variable should contain the full encrypted string. Line 52 returns the encrypted string from `encryptMessage()`:

```
52.     return ciphertext
```

Writing the Decryption Function

The `decryptMessage()` function that decrypts the text is almost the same as `encryptMessage()`. Lines 56 to 58 are equivalent to lines 42 to 44.

```
55. def decryptMessage(key, message):
56.     keyA, keyB = getKeyParts(key)
57.     checkKeys(keyA, keyB, 'decrypt')
58.     plaintext = ''
59.     modInverseOfKeyA = cryptomath.findModInverse(keyA, len(SYMBOLS))
```

However, instead of multiplying by Key A, the decryption process multiplies by the modular inverse of Key A. The mod inverse is calculated by calling `cryptomath.findModInverse()`, as explained in Chapter 13.

Lines 61 to 68 are almost identical to the `encryptMessage()` function's lines 45 to 52. The only difference is on line 65.

```
61.     for symbol in message:
62.         if symbol in SYMBOLS:
63.             # Decrypt the symbol:
64.             symbolIndex = SYMBOLS.find(symbol)
65.             plaintext += SYMBOLS[(symbolIndex - keyB) * modInverseOfKeyA %
                  len(SYMBOLS)]
66.         else:
67.             plaintext += symbol # Append the symbol without decrypting.
68.     return plaintext
```

In the `encryptMessage()` function, the symbol index was multiplied by Key A and then Key B was added to it. In the `decryptMessage()` function's

line 65, the symbol index first subtracts Key B from the symbol index and then multiplies it by the modular inverse. Then it mods this number by the size of the symbol set, `len(SYMBOLS)`.

This is how the decryption process in *affineCipher.py* undoes the encryption. Now let's look at how we can change *affineCipher.py* so that it randomly selects valid keys for the affine cipher.

Generating Random Keys

It can be difficult to come up with a valid key for the affine cipher, so you can instead use the `getRandomKey()` function to generate a random but valid key. To do this, simply change line 10 to store the return value of `getRandomKey()` in the `myKey` variable:

```
10.     myKey = getRandomKey()
        --snip--
17.     print('Key: %s' % (myKey))
```

Now the program randomly selects the key and prints it to the screen when line 17 executes. Let's look at how the `getRandomKey()` function works.

The code on line 72 enters a `while` loop where the condition is `True`. This *infinite loop* will loop forever until it is told to return or the user terminates the program. If your program gets stuck in an infinite loop, you can terminate the program by pressing CTRL-C (CTRL-D on Linux or macOS). The `getRandomKey()` function will eventually exit the infinite loop with a `return` statement.

```
71. def getRandomKey():
72.     while True:
73.         keyA = random.randint(2, len(SYMBOLS))
74.         keyB = random.randint(2, len(SYMBOLS))
```

Lines 73 and 74 determine random numbers between `2` and the size of the symbol set for `keyA` and for `keyB`. This code ensures that there's no chance that Key A or Key B will be equal to the invalid values `0` or `1`.

The `if` statement on line 75 checks to make sure that `keyA` is relatively prime with the size of the symbol set by calling the `gcd()` function in the `cryptomath` module.

```
75.         if cryptomath.gcd(keyA, len(SYMBOLS)) == 1:
76.             return keyA * len(SYMBOLS) + keyB
```

If `keyA` is relatively prime with the size of the symbol set, these two randomly selected keys are combined into a single key by multiplying `keyA` by the symbol set size and adding `keyB` to the product. (Note that this is the opposite of the `getKeyParts()` function, which splits a single integer key into two integers.) Line 76 returns this value from the `getRandomKey()` function.

If the condition on line 75 returns `False`, the code loops back to the start of the `while` loop on line 73 and picks random numbers for `keyA` and `keyB` again. The infinite loop ensures that the program continues looping until it finds random numbers that are valid keys.

Calling the main() Function

Lines 81 and 82 call the `main()` function if this program was run by itself rather than being imported by another program:

```
79. # If affineCipher.py is run (instead of imported as a module), call
80. # the main() function:
81. if __name__ == '__main__':
82.     main()
```

This ensures that the `main()` function runs when the program is run but not when the program is imported as a module.

Summary

Just as we did in Chapter 9, in this chapter we wrote a program (*affineKeyTest.py*) that can test our cipher program. Using this test program, you learned that the affine cipher has approximately 1320 possible keys, which is a number you can easily hack using brute-force. This means that we'll have to toss the affine cipher onto the heap of easily hackable weak ciphers.

So the affine cipher isn't much more secure than the previous ciphers we've looked at. The transposition cipher can have more possible keys, but the number of possible keys is limited to the size of the message. For a message with only 20 characters, the transposition cipher can have at most 18 keys, with keys ranging from 2 to 19. You can use the affine cipher to encrypt short messages with more security than the Caesar cipher provides, because its number of possible keys is based on the symbol set.

In Chapter 15, we'll write a brute-force program that can break affine cipher–encrypted messages!

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. The affine cipher is the combination of which two other ciphers?
2. What is a tuple? How is a tuple different from a list?
3. If Key A is 1, why does it make the affine cipher weak?
4. If Key B is 0, why does it make the affine cipher weak?

15

HACKING THE AFFINE CIPHER

"Cryptanalysis could not be invented until a civilization had reached a sufficiently sophisticated level of scholarship in several disciplines, including mathematics, statistics, and linguistics."
—*Simon Singh,* The Code Book

In Chapter 14, you learned that the affine cipher is limited to only a few thousand keys, which means we can easily perform a brute-force attack against it. In this chapter, you'll learn how to write a program that can break affine cipher–encrypted messages.

TOPICS COVERED IN THIS CHAPTER

- The exponent operator (**)
- The `continue` statement

Source Code for the Affine Cipher Hacker Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and then save it as *affineHacker.py*. Entering the string for the `myMessage` variable by hand might be tricky, so you can copy and paste it from the *affineHacker.py* file available at *https://www.nostarch.com/crackingcodes/* to save time. Make sure *dictionary.txt* as well as *pyperclip.py*, *affineCipher.py*, *detectEnglish.py*, and *cryptomath.py* are in the same directory as *affineHacker.py*.

affineHacker.py

```
# Affine Cipher Hacker
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import pyperclip, affineCipher, detectEnglish, cryptomath

SILENT_MODE = False

def main():
    # You might want to copy & paste this text from the source code at
    # https://www.nostarch.com/crackingcodes/.
    myMessage = """5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!A
      uaRLQADQALQG93!xQxaGaAfaQ1QX3o1RQARL9Qda!AafARuQLX1LQALQI1
      iQX3o1RN"Q-5!1RQP36ARu"""

    hackedMessage = hackAffine(myMessage)

    if hackedMessage != None:
        # The plaintext is displayed on the screen. For the convenience of
        # the user, we copy the text of the code to the clipboard:
        print('Copying hacked message to clipboard:')
        print(hackedMessage)
        pyperclip.copy(hackedMessage)
    else:
        print('Failed to hack encryption.')


def hackAffine(message):
    print('Hacking...')

    # Python programs can be stopped at any time by pressing Ctrl-C (on
    # Windows) or Ctrl-D (on macOS and Linux):
    print('(Press Ctrl-C or Ctrl-D to quit at any time.)')

    # Brute-force by looping through every possible key:
    for key in range(len(affineCipher.SYMBOLS) ** 2):
        keyA = affineCipher.getKeyParts(key)[0]
        if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) != 1:
            continue

```

```
        decryptedText = affineCipher.decryptMessage(key, message)
        if not SILENT_MODE:
            print('Tried Key %s... (%s)' % (key, decryptedText[:40]))

        if detectEnglish.isEnglish(decryptedText):
            # Check with the user if the decrypted key has been found:
            print()
            print('Possible encryption hack:')
            print('Key: %s' % (key))
            print('Decrypted message: ' + decryptedText[:200])
            print()
            print('Enter D for done, or just press Enter to continue
              hacking:')
            response = input('> ')

            if response.strip().upper().startswith('D'):
                return decryptedText
    return None


# If affineHacker.py is run (instead of imported as a module), call
# the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Affine Cipher Hacker Program

Press F5 from the file editor to run the *affineHacker.py* program; the output should look like this:

```
Hacking...
(Press Ctrl-C or Ctrl-D to quit at any time.)
Tried Key 95... (U&'<3dJ^Gjx'-3^MS'SjOjxuj'G3'%j'<mMMjS'g)
Tried Key 96... (T%&;2cI]Fiw&,2]LR&Ri/iwti&F2&$i&;lLLiR&f)
Tried Key 97... (S$%:1bH\Ehv%+1\KQ%Qh.hvsh%E1%#h%:kKKhQ%e)
--snip--
Tried Key 2190... (?^=!-+.32#O=5-3*"="#1#O4#=2-= #=!~**#"=')
Tried Key 2191... (' ^BNLOTSDQ^VNTKC^CDRDQUD^SN^AD^B@KKDC^H)
Tried Key 2192... ("A computer would deserve to be called i)
Possible encryption hack:
Key: 2192
Decrypted message: "A computer would deserve to be called intelligent if it
could deceive a human into believing that it was human." -Alan Turing
Enter D for done, or just press Enter to continue hacking:
> d
Copying hacked message to clipboard:
"A computer would deserve to be called intelligent if it could deceive a human
into believing that it was human." -Alan Turing
```

Let's take a closer look at how the affine cipher hacker program works.

Setting Up Modules, Constants, and the main() Function

The affine cipher hacker program is 60 lines long because we've already written much of the code it uses. Line 4 imports the modules we created in previous chapters:

```
1. # Affine Cipher Hacker
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, affineCipher, detectEnglish, cryptomath
5.
6. SILENT_MODE = False
```

When you run the affine cipher hacker program, you'll see that it produces lots of output as it works its way through all the possible decryptions. However, printing all this output slows down the program. If you want to speed up the program, set the `SILENT_MODE` variable on line 6 to `True` to stop it from printing all these messages.

Next, we set up the `main()` function:

```
 8. def main():
 9.     # You might want to copy & paste this text from the source code at
10.     # https://www.nostarch.com/crackingcodes/.
11.     myMessage = """5QG9ol3La6QI93!xQxaia6faQL9QdaQG1!!axQARLa!!A
          uaRLQADQALQG93!xQxaGaAfaQ1QX3o1RQARL9Qda!AafARuQLX1LQALQI1
          iQX3o1RN"Q-5!1RQP36ARu"""
12.
13.     hackedMessage = hackAffine(myMessage)
```

The ciphertext to be hacked is stored as a string in `myMessage` on line 11, and this string is passed to the `hackAffine()` function, which we'll look at in the next section. The return value from this call is either a string of the original message if the ciphertext was hacked or the `None` value if the hack failed.

The code on lines 15 to 22 checks whether `hackedMessage` was set to `None`:

```
15.     if hackedMessage != None:
16.         # The plaintext is displayed on the screen. For the convenience of
17.         # the user, we copy the text of the code to the clipboard:
18.         print('Copying hacked message to clipboard:')
19.         print(hackedMessage)
20.         pyperclip.copy(hackedMessage)
21.     else:
22.         print('Failed to hack encryption.')
```

If `hackedMessage` is not equal to `None`, the message is printed to the screen on line 19 and copied to the clipboard on line 20. Otherwise, the program simply prints feedback to the user that it was unable to hack the ciphertext. Let's take a closer look at how the `hackAffine()` function works.

The Affine Cipher Hacking Function

The `hackAffine()` function begins on line 25 and contains the code for decryption. It starts by printing some instructions for the user:

```
25. def hackAffine(message):
26.     print('Hacking...')
27.
28.     # Python programs can be stopped at any time by pressing Ctrl-C (on
29.     # Windows) or Ctrl-D (on macOS and Linux):
30.     print('(Press Ctrl-C or Ctrl-D to quit at any time.)')
```

The decryption process can take a while, so if the user wants to exit the program early, they can press CTRL-C (on Windows) or CTRL-D (on macOS and Linux).

Before we continue with the rest of the code, you need to learn about the exponent operator.

The Exponent Operator

A useful math operator you need to know to understand the affine cipher hacker program (besides the basic `+`, `-`, `*`, `/`, and `//` operators) is the *exponent operator* (`**`). The exponent operator raises a number to the power of another number. For example, two to the power of five would be `2 ** 5` in Python. This is equivalent to two multiplied by itself five times: `2 * 2 * 2 * 2 * 2`. Both expressions, `2 ** 5` and `2 * 2 * 2 * 2 * 2`, evaluate to the integer `32`.

Enter the following into the interactive shell to see how the ** operator works:

```
>>> 5 ** 2
25
>>> 2 ** 5
32
>>> 123 ** 10
792594609605189126649
```

The expression `5 ** 2` evaluates to `25` because `5` multiplied by itself is equivalent to `25`. Likewise, `2 ** 5` returns `32` because `2` multiplied by itself five times evaluates to `32`.

Let's return to the source code to see what the ** operator does in the program.

Calculating the Total Number of Possible Keys

Line 33 uses the ** operator to calculate the total number of possible keys:

```
32.     # Brute-force by looping through every possible key:
33.     for key in range(len(affineCipher.SYMBOLS) ** 2):
34.         keyA = affineCipher.getKeyParts(key)[0]
```

We know there are at most `len(affineCipher.SYMBOLS)` possible integers for Key A and `len(affineCipher.SYMBOLS)` possible integers for Key B. To get the entire range of possible keys, we multiply these values together. Because we're multiplying the same value by itself, we can use the ** operator in the expression `len(affineCipher.SYMBOLS) ** 2`.

Line 34 calls the `getKeyParts()` function that we used in *affineCipher.py* to split a single integer key into two integers. In this example, we're using the function to get the Key A part of the key we're testing. Recall that the return value of this function call is a tuple of two integers: one for Key A and one for Key B. Line 34 stores the tuple's first integer in `keyA` by placing the `[0]` after the `hackAffine()` function call.

For example, `affineCipher.getKeyParts(key)[0]` evaluates to the tuple and the index `(42, 22)[0]`, which then evaluates to `42`, the value at index `0` of the tuple. This gets just the Key A part of the return value and stores it in the variable `keyA`. The Key B part (the second value in the returned tuple) is ignored because we don't need Key B to calculate whether Key A is valid. Lines 35 and 36 check whether `keyA` is a valid Key A for the affine cipher, and if not, the program continues to the next key to try. To understand how the execution moves back to the start of the loop, you need to learn about the `continue` statement.

The continue Statement

The `continue` statement uses the `continue` keyword by itself and takes no parameters. We use a `continue` statement inside a `while` or `for` loop. When a `continue` statement executes, the program execution immediately jumps to the start of the loop for the next iteration. This also happens when the program execution reaches the end of the loop's block. But a `continue` statement makes the program execution jump back to the start of the loop before it reaches the end of the loop.

Enter the following into the interactive shell:

```
>>> for i in range(3):
...   print(i)
...   print('Hello!')
...
0
Hello!
1
Hello!
2
Hello!
```

The `for` loop loops through the `range` object, and the value in `i` becomes each integer from `0` up to, but not including, `3`. On each iteration, the `print('Hello!')` function call displays `Hello!` on the screen.

Now contrast that `for` loop with the next example, which is the same as the previous example except it has a `continue` statement before the `print('Hello!')` line.

```
>>> for i in range(3):
...   print(i)
...   continue
...   print('Hello!')
...
0
1
2
```

Notice that `Hello!` never gets printed, because the `continue` statement causes the program execution to jump back to the start of the `for` loop for the next iteration and the execution never reaches the `print('Hello!')` line.

A `continue` statement is often placed inside an `if` statement's block so that execution continues at the beginning of the loop under certain conditions. Let's return to our code to see how it uses the `continue` statement to skip execution depending on the key used.

Using continue to Skip Code

In the source code, line 35 uses the `gcd()` function in the `cryptomath` module to determine whether Key A is relatively prime to the symbol set size:

```
35.         if cryptomath.gcd(keyA, len(affineCipher.SYMBOLS)) != 1:
36.             continue
```

Recall that two numbers are relatively prime if their greatest common divisor (GCD) is 1. If Key A and the symbol set size are not relatively prime, the condition on line 35 is `True` and the `continue` statement on line 36 executes. This causes the program execution to jump back to the start of the loop for the next iteration. As a result, the program skips the call to `decryptMessage()` on line 38 if the key is invalid and continues to try other keys until it finds the right one.

When the program finds the right key, the message is decrypted by calling `decryptMessage()` with the key on line 38:

```
38.         decryptedText = affineCipher.decryptMessage(key, message)
39.         if not SILENT_MODE:
40.             print('Tried Key %s... (%s)' % (key, decryptedText[:40]))
```

If `SILENT_MODE` was set to `False`, the `Tried Key` message is printed on the screen, but if it was set to `True`, the `print()` call on line 40 is skipped.

Next, line 42 uses the `isEnglish()` function from the `detectEnglish` module to check whether the decrypted message is recognized as English:

```
42.         if detectEnglish.isEnglish(decryptedText):
43.             # Check with the user if the decrypted key has been found:
44.             print()
45.             print('Possible encryption hack:')
46.             print('Key: %s' % (key))
```

```
47.             print('Decrypted message: ' + decryptedText[:200])
48.             print()
```

If the wrong decryption key was used, the decrypted message would look like random characters and `isEnglish()` would return `False`. But if the decrypted message is recognized as readable English (by the `isEnglish()` function's standards), the program displays it to the user.

We display a snippet of the decrypted message that is recognized as English, because the `isEnglish()` function might mistakenly identify text as English even though it hasn't found the correct key. If the user decides that this is indeed the correct decryption, they can type `D` and then press ENTER.

```
49.             print('Enter D for done, or just press Enter to continue
                  hacking:')
50.             response = input('> ')
51.
52.             if response.strip().upper().startswith('D'):
53.                 return decryptedText
```

Otherwise, the user can just press ENTER to return a blank string from the `input()` call, and the `hackAffine()` function would continue trying more keys.

From the indentation at the beginning of line 54, you can see that this line executes after the `for` loop on line 33 has completed:

```
54.     return None
```

If the `for` loop finishes and reaches line 54, then it has gone through every possible decryption key without finding the correct one. At this point, the `hackAffine()` function returns the `None` value to signal that it was unsuccessful at hacking the ciphertext.

If the program had found the correct key, the execution would have previously returned from the function on line 53 and never reached line 54.

Calling the main() Function

If we run *affineHacker.py* as a program, the special `__name__` variable will be set to the string `'__main__'` instead of `'affineHacker'`. In this case, we call the `main()` function.

```
57. # If affineHacker.py is run (instead of imported as a module), call
58. # the main() function:
59. if __name__ == '__main__':
60.     main()
```

That concludes the affine cipher hacking program.

Summary

This chapter is fairly short because it doesn't introduce any new hacking techniques. As you've seen, as long as the number of possible keys is only a few thousand, it won't take long for computers to brute-force through every possible key and use the `isEnglish()` function to search for the right key.

You learned about the exponent operator (**), which raises a number to the power of another number. You also learned how to use the `continue` statement to send the program execution back to the beginning of the loop instead of waiting until the execution reaches the end of the block.

Conveniently, we already wrote much of the code used for the affine cipher hacker in *affineCipher.py*, *detectEnglish.py*, and *cryptomath.py*. The `main()` function trick helps us reuse the code in our programs.

In Chapter 16, you'll learn about the simple substitution cipher, which computers can't brute-force. The number of possible keys for this cipher is more than trillions of trillions! A single laptop couldn't possibly go through a fraction of those keys in our lifetime, which makes the cipher immune to a brute-force attack.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What does `2 ** 5` evaluate to?
2. What does `6 ** 2` evaluate to?
3. What does the following code print?

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

4. Does the `main()` function of *affineHacker.py* get called if another program runs `import affineHacker`?

16

PROGRAMMING THE SIMPLE SUBSTITUTION CIPHER

"The internet is the most liberating tool for humanity ever invented, and also the best for surveillance. It's not one or the other. It's both."
—John Perry Barlow, co-founder of the Electronic Frontier Foundation

In Chapter 15, you learned that the affine cipher has about a thousand possible keys but that computers can still brute-force through all of them easily. We need a cipher that has so many possible keys that no computer can brute-force through them all.

The *simple substitution cipher* is one such cipher that is effectively invulnerable to a brute-force attack because it has an enormous number of possible keys. Even if your computer could try a trillion keys every second, it would still take 12 million years for it to try every one! In this chapter, you'll write a program to implement the simple substitution cipher and learn some useful Python functions and string methods as well.

TOPICS COVERED IN THIS CHAPTER

- The `sort()` list method
- Getting rid of duplicate characters from a string
- Wrapper functions
- The `isupper()` and `islower()` string methods

How the Simple Substitution Cipher Works

To implement the simple substitution cipher, we choose a random letter to encrypt each letter of the alphabet, using each letter only once. The key for the simple substitution cipher is always a string of 26 letters of the alphabet in random order. There are 403,291,461,126,605,635,584,000,000 different possible key orderings for the simple substitution cipher. That's a lot of keys! More important, this number is so large that it's impossible to brute-force. (To see how this number was calculated, go to *https://www.nostarch.com/crackingcodes/*.)

Let's try using the simple substitution cipher with paper and pencil first. For this example, we'll encrypt the message "Attack at dawn." using the key VJZBGNFEPLITMXDWKQUCRYAHSO. First, write out the letters of the alphabet and the corresponding key underneath each letter, as in Figure 16-1.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O

Figure 16-1: Encryption letters for the example key

To encrypt a message, find the letter in the plaintext in the top row and substitute it with the letter in the bottom row. *A* encrypts to *V*, *T* encrypts to *C*, *C* encrypts to *Z*, and so on. So the message "Attack at dawn." encrypts to "Vccvzi vc bvax."

To decrypt the encrypted message, find the letter in the ciphertext in the bottom row and replace it with the corresponding letter in the top row. *V* decrypts to *A*, *C* decrypts to *T*, *Z* decrypts to *C*, and so on.

Unlike the Caesar cipher, in which the bottom row shifts but remains in alphabetical order, in the simple substitution cipher the bottom row is completely scrambled. This results in far more possible keys, which is a huge advantage of using the simple substitution cipher. The disadvantage is that the key is 26 characters long and more difficult to memorize. You may need to write down the key, but if you do, make sure no one else ever reads it!

Source Code for the Simple Substitution Cipher Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *simpleSubCipher.py*. Be sure to place the *pyperclip.py* file in the same directory as the *simpleSubCipher.py* file. Press F5 to run the program.

simpleSubCipher.py

```
# Simple Substitution Cipher
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import pyperclip, sys, random


LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def main():
    myMessage = 'If a man is offered a fact which goes against his
      instincts, he will scrutinize it closely, and unless the evidence
      is overwhelming, he will refuse to believe it. If, on the other
      hand, he is offered something which affords a reason for acting
      in accordance to his instincts, he will accept it even on the
      slightest evidence. The origin of myths is explained in this way.
      -Bertrand Russell'
    myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
    myMode = 'encrypt' # Set to 'encrypt' or 'decrypt'.

    if keyIsValid(myKey):
        sys.exit('There is an error in the key or symbol set.')
    if myMode == 'encrypt':
        translated = encryptMessage(myKey, myMessage)
    elif myMode == 'decrypt':
        translated = decryptMessage(myKey, myMessage)
    print('Using key %s' % (myKey))
    print('The %sed message is:' % (myMode))
    print(translated)
    pyperclip.copy(translated)
    print()
    print('This message has been copied to the clipboard.')


def keyIsValid(key):
    keyList = list(key)
    lettersList = list(LETTERS)
    keyList.sort()
    lettersList.sort()

    return keyList == lettersList


def encryptMessage(key, message):
    return translateMessage(key, message, 'encrypt')

```

```
41. def decryptMessage(key, message):
42.     return translateMessage(key, message, 'decrypt')
43.
44.
45. def translateMessage(key, message, mode):
46.     translated = ''
47.     charsA = LETTERS
48.     charsB = key
49.     if mode == 'decrypt':
50.         # For decrypting, we can use the same code as encrypting. We
51.         # just need to swap where the key and LETTERS strings are used.
52.         charsA, charsB = charsB, charsA
53.
54.     # Loop through each symbol in the message:
55.     for symbol in message:
56.         if symbol.upper() in charsA:
57.             # Encrypt/decrypt the symbol:
58.             symIndex = charsA.find(symbol.upper())
59.             if symbol.isupper():
60.                 translated += charsB[symIndex].upper()
61.             else:
62.                 translated += charsB[symIndex].lower()
63.         else:
64.             # Symbol is not in LETTERS; just add it:
65.             translated += symbol
66.
67.     return translated
68.
69.
70. def getRandomKey():
71.     key = list(LETTERS)
72.     random.shuffle(key)
73.     return ''.join(key)
74.
75.
76. if __name__ == '__main__':
77.     main()
```

Sample Run of the Simple Substitution Cipher Program

When you run the *simpleSubCipher.py* program, the encrypted output should look like this:

```
Using key LFWOAYUISVKMNXPBDCRJTQEGHZ
The encrypted message is:
Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr sxrjsxwjr, ia esmm
rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia esmm caytra
jp famsaqa sj. Sy, px jia pjiac ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor
l calrpx ypc lwjsxu sx lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax
px jia rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.
-Facjclxo Ctrramm

This message has been copied to the clipboard.
```

Notice that if the letter in the plaintext is lowercase, it's lowercase in the ciphertext. Likewise, if the letter is uppercase in the plaintext, it's uppercase in the ciphertext. The simple substitution cipher doesn't encrypt spaces or punctuation marks and simply returns those characters as is.

To decrypt this ciphertext, paste it as the value for the `myMessage` variable on line 10 and change `myMode` to the string `'decrypt'`. When you run the program again, the decryption output should look like this:

```
Using key LFWOAYUISVKMNXPBDCRJTQEGHZ
The decrypted message is:
If a man is offered a fact which goes against his instincts, he will
scrutinize it closely, and unless the evidence is overwhelming, he will refuse
to believe it. If, on the other hand, he is offered something which affords
a reason for acting in accordance to his instincts, he will accept it even
on the slightest evidence. The origin of myths is explained in this way.
-Bertrand Russell

This message has been copied to the clipboard.
```

Setting Up Modules, Constants, and the main() Function

Let's look at the first lines of simple substitution cipher program's source code.

```
1. # Simple Substitution Cipher
2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
3.
4. import pyperclip, sys, random
5.
6.
7. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Line 4 imports the `pyperclip`, `sys`, and `random` modules. The `LETTERS` constant variable is set to a string of all the uppercase letters, which is the symbol set for the simple substitution cipher program.

The `main()` function in *simpleSubCipher.py*, which is similar to the `main()` function of cipher programs in the previous chapters, is called when the program is first run. It contains the variables that store the `message`, `key`, and `mode` used for the program.

```
9. def main():
10.     myMessage = 'If a man is offered a fact which goes against his
          instincts, he will scrutinize it closely, and unless the evidence
          is overwhelming, he will refuse to believe it. If, on the other
          hand, he is offered something which affords a reason for acting
          in accordance to his instincts, he will accept it even on the
          slightest evidence. The origin of myths is explained in this way.
          -Bertrand Russell'
11.     myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'
12.     myMode = 'encrypt' # Set to 'encrypt' or 'decrypt'.
```

The keys for simple substitution ciphers are easy to get wrong because they're fairly long and need to have every letter in the alphabet. For example, it's easy to enter a key that is missing a letter or a key that has the same letter twice. The `keyIsValid()` function makes sure the key is usable by the encryption and decryption functions, and the function exits the program with an error message if the key is not valid:

```
14.     if keyIsValid(myKey):
15.         sys.exit('There is an error in the key or symbol set.')
```

If line 14 returns `False` from `keyIsValid()`, then `myKey` contains an invalid key and line 15 terminates the program.

Lines 16 through 19 check whether the `myMode` variable is set to `'encrypt'` or `'decrypt'` and calls either `encryptMessage()` or `decryptMessage()` accordingly:

```
16.     if myMode == 'encrypt':
17.         translated = encryptMessage(myKey, myMessage)
18.     elif myMode == 'decrypt':
19.         translated = decryptMessage(myKey, myMessage)
```

The return value of `encryptMessage()` and `decryptMessage()` is a string of the encrypted or decrypted message that is stored in the `translated` variable.

Line 20 prints the key that was used to the screen. The encrypted or decrypted message is printed to the screen and also copied to the clipboard.

```
20.     print('Using key %s' % (myKey))
21.     print('The %sed message is:' % (myMode))
22.     print(translated)
23.     pyperclip.copy(translated)
24.     print()
25.     print('This message has been copied to the clipboard.')
```

Line 25 is the last line of code in the `main()` function, so the program execution returns after line 25. When the `main()` call is done on the last line of the program, the program exits.

Next, we'll look at how the `keyIsValid()` function uses the `sort()` method to test whether the key is valid.

The sort() List Method

Lists have a `sort()` method that rearranges the list's items into numerical or alphabetical order. This ability to sort items in a list comes in handy when you have to check whether two lists contain the same items but don't list them in the same order.

In *simpleSubCipher.py*, a simple substitution key string value is valid only if it has each of the characters in the symbol set with no duplicate or missing letters. We can check whether a string value is a valid key by sorting it and checking whether it's equal to the sorted `LETTERS`. But because

we can sort only lists, not strings (recall that strings are immutable, meaning their values cannot be changed), we'll obtain list versions of the string values by passing them to `list()`. Then, after sorting these lists, we can compare the two to see whether or not they're equal. Although `LETTERS` is already in alphabetical order, we'll sort it because we'll expand it to contain other characters later on.

```
28. def keyIsValid(key):
29.     keyList = list(key)
30.     lettersList = list(LETTERS)
31.     keyList.sort()
32.     lettersList.sort()
```

The string in `key` is passed to `list()` on line 29. The list value returned is stored in a variable named `keyList`.

On line 30, the `LETTERS` constant variable (which contains the string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`) is passed to `list()`, which returns the list in the following format: `['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']`.

On lines 31 and 32, the lists in `keyList` and `lettersList` are then sorted in alphabetical order by calling the `sort()` list method on them. Note that similar to the `append()` list method, the `sort()` list method modifies the list in place and doesn't have a return value.

When sorted, the `keyList` and `lettersList` values *should* be the same, because `keyList` was simply the characters in `LETTERS` with the order scrambled. Line 34 checks whether the values `keyList` and `lettersList` are equal:

```
34.     return keyList == lettersList
```

If `keyList` and `lettersList` are equal, you can be sure that `keyList` and the `key` parameter don't have any duplicated characters, because `LETTERS` doesn't have duplicates in it. In that case, line 34 returns `True`. But if `keyList` and `lettersList` don't match, the key is invalid and line 34 returns `False`.

Wrapper Functions

The encryption code and the decryption code in the *simpleSubCipher.py* program are almost identical. When you have two very similar pieces of code, it's best to put them into a function and call it twice rather than enter the code twice. Not only does this save time, but more important, it avoids introducing bugs while copying and pasting code. It's also advantageous because if there's ever a bug in the code, you only have to fix the bug in one place instead of in multiple places.

Wrapper functions help you avoid having to enter duplicate code by wrapping the code of another function and returning the value the wrapped function returns. Often, the wrapper function makes a slight change to

the arguments or return value of the wrapped function. Otherwise, there would be no need for wrapping because you could just call the function directly.

Let's look at an example of using wrapper functions in our code to understand how they work. In this case, `encryptMessage()` and `decryptMessage()` on lines 37 and 41 are the wrapper functions:

```
37. def encryptMessage(key, message):
38.     return translateMessage(key, message, 'encrypt')
39.
40.
41. def decryptMessage(key, message):
42.     return translateMessage(key, message, 'decrypt')
```

Each of these wrapper functions calls `translateMessage()`, which is the wrapped function, and returns the value that `translateMessage()` returns. (We'll look at the `translateMessage()` function in the next section.) Because both wrapper functions use the same `translateMessage()` function, we need to modify only that one function instead of the `encryptMessage()` and `decryptMessage()` functions if we need to make any changes to the cipher.

With these wrapper functions, someone who imports the program *simpleSubCipher.py* can call the functions named `encryptMessage()` and `decryptMessage()` just as they can with all the other cipher programs in this book. The wrapper functions have clear names that tell others who use the functions what they do without having to look at the code. As a result, if we want to share our code, others can use it more easily.

Other programs can encrypt a message in various ciphers by importing the cipher programs and calling their `encryptMessage()` functions, as shown here:

```
import affineCipher, simpleSubCipher, transpositionCipher
--snip--
ciphertext1 =        affineCipher.encryptMessage(encKey1, 'Hello!')
ciphertext2 = transpositionCipher.encryptMessage(encKey2, 'Hello!')
ciphertext3 =     simpleSubCipher.encryptMessage(encKey3, 'Hello!')
```

Naming consistency is helpful, because it makes it easier for someone familiar with one of the cipher programs to use the other cipher programs. For example, you can see that the first parameter is always the key and the second parameter is always the message, which is the convention used for most of the cipher programs in this book. Using the `translateMessage()` function instead of separate `encryptMessage()` and `decryptMessage()` functions would be inconsistent with the other programs.

Let's look at the `translateMessage()` function next.

The translateMessage() Function

The translateMessage() function is used for both encryption and decryption.

```
45. def translateMessage(key, message, mode):
46.     translated = ''
47.     charsA = LETTERS
48.     charsB = key
49.     if mode == 'decrypt':
50.         # For decrypting, we can use the same code as encrypting. We
51.         # just need to swap where the key and LETTERS strings are used.
52.         charsA, charsB = charsB, charsA
```

Notice that translateMessage() has the parameters key and message but also a third parameter named mode. When we call translateMessage(), the call in the encryptMessage() function passes 'encrypt' for the mode parameter, and the call in the decryptMessage() function passes 'decrypt'. This is how the translateMessage() function knows whether it should encrypt or decrypt the message passed to it.

The actual encryption process is simple: for each letter in the message parameter, the function looks up that letter's index in LETTERS and replaces the character with the letter at that same index in the key parameter. Decryption does the opposite: it looks up the index in key and replaces the character with the letter at the same index in LETTERS.

Instead of using LETTERS and key, the program uses the variables charsA and charsB, which allow it to replace the letter in charsA with the letter at the same index in charsB. Being able to change which values are assigned to charsA and charsB makes it easy for the program to switch between encrypting and decrypting. Line 47 sets the characters in charsA to the characters in LETTERS, and line 48 sets the characters in charsB to the characters in key.

The following figures show how the same code can be used to either encrypt or decrypt a letter. Figure 16-2 illustrates the encryption process. The top row in this figure shows the characters in charsA (set to LETTERS), the middle row shows the characters in charsB (set to key), and the bottom row shows the integer indexes corresponding to the characters.

charsA	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
charsB	V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figure 16-2: Using the index to encrypt plaintext

The code in translateMessage() always looks up the message character's index in charsA and replaces it with the corresponding character in charsB at that index. So to encrypt, we just leave charsA and charsB as they are. Using the variables charsA and charsB replaces the character in LETTERS with the character in key, because charsA is set to LETTERS and charsB is set to key.

To decrypt, the values in `charsA` and `charsB` are switched using `charsA, charsB = charsB, charsA` on line 52. Figure 16-3 shows the decryption process.

charsA	V	J	Z	B	G	N	F	E	P	L	I	T	M	X	D	W	K	Q	U	C	R	Y	A	H	S	O
charsB	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figure 16-3: Using the index to decrypt ciphertext

Keep in mind that the code in `translateMessage()` always replaces the character in `charsA` with the character at that same index in `charsB`. So when line 52 swaps the values, the code in `translateMessage()` does the decryption process instead of the encryption process.

The next lines of code show how the program finds the index to use for encryption and decryption.

```
54.     # Loop through each symbol in the message:
55.     for symbol in message:
56.         if symbol.upper() in charsA:
57.             # Encrypt/decrypt the symbol:
58.             symIndex = charsA.find(symbol.upper())
```

The `for` loop on line 55 sets the `symbol` variable to a character in the `message` string on each iteration through the loop. If the uppercase form of this symbol exists in `charsA` (recall that `key` and `LETTERS` have only uppercase characters in them), line 58 finds the index of the uppercase form of `symbol` in `charsA`. The `symIndex` variable stores this index.

We already know that the `find()` method would never return `-1` (a `-1` from the `find()` method means the argument could not be found in the string) because the `if` statement on line 56 guarantees that `symbol.upper()` exists in `charsA`. Otherwise, line 58 wouldn't have been executed.

Next, we'll use each encrypted or decrypted `symbol` to build the string that is returned by the `translateMessage()` function. But because `key` and `LETTERS` are both only in uppercase, we'll need to check whether the original `symbol` in `message` was lowercase and then adjust the decrypted or encrypted `symbol` to lowercase if it was. To do this, you need to learn two string methods: `isupper()` and `islower()`.

The isupper() and islower() String Methods

The `isupper()` and `islower()` methods check whether a string is in uppercase or lowercase.

More specifically, the `isupper()` string method returns `True` if both of these conditions are met:

- The string has at least one uppercase letter.
- The string does not have any lowercase letters in it.

The `islower()` string method returns `True` if both of these conditions are met:

- The string has at least one lowercase letter.
- The string does not have any uppercase letters in it.

Non-letter characters in the string don't affect whether these methods return `True` or `False`, although both methods evaluate to `False` if only non-letter characters exist in the string. Enter the following into the interactive shell to see how these methods work:

```
>>> 'HELLO'.isupper()
True
❶ >>> 'HELLO WORLD 123'.isupper()
True
❷ >>> 'hello'.islower()
True
>>> '123'.isupper()
False
>>> ''.islower()
False
```

The example at ❶ returns `True` because `'HELLO WORLD 123'` has at least one uppercase letter in it and no lowercase letters. The numbers in that string don't affect the evaluation. At ❷, `'hello'.islower()` returns `True` because the string `'hello'` has at least one lowercase letter in it and no uppercase letters.

Let's return to our code to see how it uses the `isupper()` and `islower()` string methods.

Preserving Cases with isupper()

The *simpleSubCipher.py* program uses the `isupper()` and `islower()` string methods to help ensure that the cases of the plaintext are reflected in the ciphertext.

```
59.             if symbol.isupper():
60.                 translated += charsB[symIndex].upper()
61.             else:
62.                 translated += charsB[symIndex].lower()
```

Line 59 tests whether `symbol` has an uppercase letter. If it does, line 60 concatenates the uppercase version of the character at `charsB[symIndex]` to `translated`. This results in the uppercase version of the `key` character corresponding to the uppercase input. If `symbol` instead has a lowercase letter, line 62 concatenates the lowercase version of the character at `charsB[symIndex]` to `translated`.

If `symbol` is not a character in the symbol set, such as `'5'` or `'?'`, line 59 would return `False`, and line 62 would execute instead of line 60. The reason is that the conditions for `isupper()` wouldn't be met because those strings don't have at least one uppercase letter. In this case, the `lower()`

method call on line 62 would have no effect on the string because it has no letters at all. The `lower()` method doesn't change non-letter characters like `'5'` and `'?'`. It simply returns the original non-letter characters.

Line 62 in the `else` block accounts for any lowercase characters *and* non-letter characters in our `symbol` string.

The indentation on line 63 indicates that the `else` statement is paired with the `if symbol.upper() in charsA:` statement on line 56, so line 63 executes if `symbol` is not in `LETTERS`.

```
63.         else:
64.             # Symbol is not in LETTERS; just add it:
65.             translated += symbol
```

If `symbol` is not in `LETTERS`, line 65 executes. This means we cannot encrypt or decrypt the character in `symbol`, so we simply concatenate it to the end of `translated` as is.

At the end of the `translateMessage()` function, line 67 returns the value in the `translated` variable, which contains the encrypted or decrypted message:

```
67.     return translated
```

Next, we'll look at how to use the `getRandomKey()` function to generate a valid key for the simple substitution cipher.

Generating a Random Key

Typing a string for a key that contains each letter of the alphabet can be difficult. To help us with this, the `getRandomKey()` function returns a valid key to use. Lines 71 to 73 randomly scramble the characters in the `LETTERS` constant.

```
70. def getRandomKey():
71.     key = list(LETTERS)
72.     random.shuffle(key)
73.     return ''.join(key)
```

NOTE *Read "Randomly Scrambling a String" on page 123 for an explanation of how to scramble a string using the `list()`, `random.shuffle()`, and `join()` methods.*

To use the `getRandomKey()` function, we need to change line 11 from `myKey = 'LFWOAYUISVKMNXPBDCRJTQEGHZ'` to this:

```
11.     myKey = getRandomKey()
```

Because line 20 in our simple substitution cipher program prints the key being used, you'll be able to see the key the `getRandomKey()` function returned.

Calling the main() Function

Lines 76 and 77 at the end of the program call `main()` if *simpleSubCipher.py* is being run as a program instead of being imported as a module by another program.

```
76. if __name__ == '__main__':
77.     main()
```

This concludes our study of the simple substitution cipher program.

Summary

In this chapter, you learned how to use the `sort()` list method to order items in a list and how to compare two ordered lists to check for duplicate or missing characters from a string. You also learned about the `isupper()` and `islower()` string methods, which check whether a string value is made up of uppercase or lowercase letters. You learned about wrapper functions, which are functions that call other functions, usually adding only slight changes or different arguments.

The simple substitution cipher has far too many possible keys to brute-force through. This makes it impervious to the techniques you used to hack previous cipher programs. You'll have to make smarter programs to break this code.

In Chapter 17, you'll learn how to hack the simple substitution cipher. Instead of brute-forcing through all the keys, you'll use a more intelligent and sophisticated algorithm.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Why can't a brute-force attack be used against a simple substitution cipher, even with a powerful supercomputer?
2. What does the `spam` variable contain after running this code?

```
spam = [4, 6, 2, 8]
spam.sort()
```

3. What is a wrapper function?
4. What does `'hello'.islower()` evaluate to?
5. What does `'HELLO 123'.isupper()` evaluate to?
6. What does `'123'.islower()` evaluate to?

17

HACKING THE SIMPLE SUBSTITUTION CIPHER

"Encryption is fundamentally a private act. The act of encryption, in fact, removes information from the public realm. Even laws against cryptography reach only so far as a nation's border and the arm of its violence."
—Eric Hughes, "A Cypherpunk's Manifesto" (1993)

In Chapter 16, you learned that the simple substitution cipher is impossible to crack using brute force because it has too many possible keys. To hack the simple substitution cipher, we need to create a more sophisticated program that uses dictionary values to map the potential decryption letters of a ciphertext. In this chapter, we'll write such a program to narrow down the list of potential decryption outputs to the right one.

TOPICS COVERED IN THIS CHAPTER

- Word patterns, candidates, potential decryption letters, and cipherletter mappings
- Regular expressions
- The `sub()` regex method

Using Word Patterns to Decrypt

In brute-force attacks, we try each possible key to check whether it can decrypt the ciphertext. If the key is correct, the decryption results in readable English. But by analyzing the ciphertext first, we can reduce the number of possible keys to try and maybe even find a full or partial key.

Let's assume the original plaintext consists mostly of words in an English dictionary file, like the one we used in Chapter 11. Although a ciphertext won't be made of real English words, it will still contain groups of letters broken up by spaces, just like words in regular sentences. We'll call these *cipherwords* in this book. In a substitution cipher, every letter of the alphabet has exactly one unique corresponding encryption letter. We'll call the letters in the ciphertext *cipherletters*. Because each plaintext letter can encrypt to only one cipherletter, and we're not encrypting spaces in this version of the cipher, the plaintext and ciphertext will share the same *word patterns*.

For example, if we had the plaintext MISSISSIPPI SPILL, the corresponding ciphertext might be RJBBJBBJXXJ BXJHH. The number of letters in the first word of the plaintext and the first cipherword are the same. The same is true for the second plaintext word and the second cipherword. The plaintext and ciphertext share the same pattern of letters and spaces. Also notice that letters that repeat in the plaintext repeat the same number of times and in the same places as the ciphertext.

We could therefore assume that a cipherword corresponds to a word in the English dictionary file and that their word patterns would match. Then, if we can find which word in the dictionary the cipherword decrypts to, we can figure out the decryption of each cipherletter in that word. And if we figure out enough cipherletter decryptions using this technique, we may be able to decrypt the entire message.

Finding Word Patterns

Let's examine the word pattern of the cipherword HGHHU. You can see that the cipherword has certain characteristics, which the original plaintext word must share. Both words must have the following in common.

1. They should be five letters long.
2. The first, third, and fourth letters should be the same.
3. They should have exactly three different letters; the first, second, and fifth letters should all be different.

Let's think of words in the English language that fit this pattern. *Puppy* is one such word, which is five letters long (P, U, P, P, Y) and uses three different letters (P, U, Y) arranged in that same pattern (P for the first, third, and fourth letter; U for the second letter; and Y for the fifth letter). *Mommy*, *bobby*, *lulls*, and *nanny* fit the pattern, too. These words, along with any other word in the English dictionary file that matches the criteria, are all possible decryptions of HGHHU.

To represent a word pattern in a way the program can understand, we'll make each pattern into a set of numbers separated by periods that indicates the pattern of letters.

Creating word patterns is easy: the first letter gets the number 0, and the first occurrence of each different letter thereafter gets the next number. For example, the word pattern for *cat* is 0.1.2, and the word pattern for *classification* is 0.1.2.3.3.4.5.4.0.2.6.4.7.8.

In simple substitution ciphers, no matter which key is used to encrypt, a plaintext word and its cipherword *always* have the same word pattern. The word pattern for the cipherword HGHHU is 0.1.0.0.2, which means the word pattern of the plaintext corresponding to HGHHU is also 0.1.0.0.2.

Finding Potential Decryption Letters

To decrypt HGHHU, we need to find all the words in an English dictionary file whose word pattern is also 0.1.0.0.2. In this book, we'll call the plaintext words that have the same word pattern as the cipherword the *candidates* for that cipherword. Here is a list of candidates for HGHHU:

- puppy
- mommy
- bobby
- lulls
- nanny

Using word patterns, we can guess which plaintext letters cipherletters might decrypt to, which we'll call the cipherletter's *potential decryption letters*. To crack a message encrypted with the simple substitution cipher, we need to find all the potential decryption letters of each word in the message and determine the actual decryption letters through the process of elimination. Table 17-1 lists the potential decryption letters for HGHHU.

Table 17-1: Potential Decryption Letters of the Cipherletters in HGHHU

Cipherletters	**H**	**G**	**H**	**H**	**U**
Potential decryption letters	P	U	P	P	Y
	M	O	M	M	Y
	B	O	B	B	Y
	L	U	L	L	S
	N	A	N	N	Y

The following is a *cipherletter mapping* created using Table 17-1:

1. H has the potential decryption letters P, M, B, L, and N.
2. G has the potential decryption letters U, O, and A.
3. U has the potential decryption letters Y and S.
4. All of the other cipherletters besides H, G, and U have no potential decryption letters in this example.

A cipherletter mapping shows all the letters of the alphabet and their potential decryption letters. As we start to gather encrypted messages, we'll find potential decryption letters for every letter in the alphabet, but because only the cipherletters H, G, and U were part of our example ciphertext, we don't have the potential decryption letters of other cipherletters.

Notice also that U has only two potential decryption letters (Y and S) because there are overlaps between the candidates, many of which end in the letter Y. *The more overlaps there are, the fewer potential decryption letters there will be, and the easier it will be to figure out what that cipherletter decrypts to.*

To represent Table 17-1 in Python code, we'll use a dictionary value to represent cipherletter mappings as follows (the key-value pairs for `'H'`, `'G'`, and `'U'` are in bold):

```
{'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': ['U', 'O', 'A'],
'H': ['P', 'M', 'B', 'L', 'N'], 'I': [], 'J': [], 'K': [], 'L': [], 'M': [],
'N': [], 'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': ['Y',
'S'], 'V': [], 'W': [], 'X': [], 'Y': [], 'Z': []}
```

This dictionary has 26 key-value pairs, one key for each letter of the alphabet and a list of potential decryption letters for each letter. It shows potential decryption letters for keys `'H'`, `'G'`, and `'U'`. The other keys have empty lists, `[]`, for values, because they have no potential decryption letters so far.

If we can reduce the number of potential decryption letters for a cipherletter to just one letter by cross-referencing cipherletter mappings of other encrypted words, we can find what that cipherletter decrypts to. Even if we can't solve all 26 cipherletters, we might be able to hack most of the cipherletter mappings to decrypt most of the ciphertext.

Now that we've covered some of the basic concepts and terminology we'll be using in this chapter, let's look at the steps involved in the hacking process.

Overview of the Hacking Process

Hacking the simple substitution cipher is pretty easy using word patterns. We can summarize the major steps of the hacking process as follows:

1. Find the word pattern for each cipherword in the ciphertext.
2. Find the English word candidates that each cipherword could decrypt to.
3. Create a dictionary showing potential decryption letters for each cipherletter to act as the cipherletter mapping for each cipherword.
4. Combine the cipherletter mappings into a single mapping, which we'll call an *intersected mapping*.
5. Remove any solved cipherletters from the combined mapping.
6. Decrypt the ciphertext with the solved cipherletters.

The more cipherwords in a ciphertext, the more likely it is for the mappings to overlap with one another and the fewer the potential decryption letters for each cipherletter. This means that in the simple substitution cipher, *the longer the ciphertext message, the easier it is to hack*.

Before diving into the source code, let's look at how we can make the first two steps of the hacking process easier. We'll use the dictionary file we used in Chapter 11 and a module called *wordPatterns.py* to get the word pattern for every word in the dictionary file and sort them in a list.

The Word Pattern Modules

To calculate word patterns for every word in the *dictionary.txt* dictionary file, download *makeWordPatterns.py* from *https://www.nostarch.com/crackingcodes/*. Make sure this program and *dictionary.txt* are both in the folder where you'll be saving this chapter's *simpleSubHacker.py* program.

The *makeWordPatterns.py* program has a `getWordPattern()` function that takes a string (such as `'puppy'`) and returns its word pattern (such as `'0.1.0.0.2'`). When you run *makeWordPatterns.py*, it should create the Python module *wordPatterns.py*. The module contains a single variable assignment statement, as shown here, and is more than 43,000 lines long:

```
allPatterns = {'0.0.1': ['EEL'],
 '0.0.1.2': ['EELS', 'OOZE'],
 '0.0.1.2.0': ['EERIE'],
 '0.0.1.2.3': ['AARON', 'LLOYD', 'OOZED'],
--snip--
```

The allPatterns variable contains a dictionary value with the word pattern strings as keys and a list of English words that match the pattern as its values. For example, to find all the English words with the pattern 0.1.2.1.3.4.5.4.6.7.8, enter the following into the interactive shell:

```
>>> import wordPatterns
>>> wordPatterns.allPatterns['0.1.2.1.3.4.5.4.6.7.8']
['BENEFICIARY', 'HOMOGENEITY', 'MOTORCYCLES']
```

In the allPatterns dictionary, the key '0.1.2.1.3.4.5.4.6.7.8' has the list value ['BENEFICIARY', 'HOMOGENEITY', 'MOTORCYCLES'], which contains three English words with this particular word pattern.

Now let's import the *wordPatterns.py* module to start building the simple substitution hacking program!

NOTE *If you get a ModuleNotFoundError error message when importing wordPatterns into the interactive shell, enter the following into the interactive shell first:*

```
>>> import sys
>>> sys.path.append('name_of_folder')
```

Replace name_of_folder with the location where wordPatterns.py *is saved. This tells the interactive shell to look for modules in the folder you specify.*

Source Code for the Simple Substitution Hacking Program

Open a file editor window by selecting **File ▸ New File**. Enter the following code into the file editor and save it as *simpleSubHacker.py*. Be sure to place the *pyperclip.py*, *simpleSubCipher.py*, and *wordPatterns.py* files in the same directory as *simpleSubHacker.py*. Press F5 to run the program.

simpleSubHacker.py

```
# Simple Substitution Cipher Hacker
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

import os, re, copy, pyperclip, simpleSubCipher, wordPatterns,
    makeWordPatterns




LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
nonLettersOrSpacePattern = re.compile('[^A-Z\s]')

def main():
    message = 'Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr
      sxrjsxwjr, ia esmm rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa
      sr pqaceiamnsxu, ia esmm caytra jp famsaqa sj. Sy, px jia pjiac
      ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor l calrpx ypc lwjsxu sx
```

```
        lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax px jia
        rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.
        -Facjclxo Ctrramm'

    # Determine the possible valid ciphertext translations:
    print('Hacking...')
    letterMapping = hackSimpleSub(message)

    # Display the results to the user:
    print('Mapping:')
    print(letterMapping)
    print()
    print('Original ciphertext:')
    print(message)
    print()
    print('Copying hacked message to clipboard:')
    hackedMessage = decryptWithCipherletterMapping(message, letterMapping)
    pyperclip.copy(hackedMessage)
    print(hackedMessage)


def getBlankCipherletterMapping():
    # Returns a dictionary value that is a blank cipherletter mapping:
    return {'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': [],
      'H': [], 'I': [], 'J': [], 'K': [], 'L': [], 'M': [], 'N': [],
      'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': [],
      'V': [], 'W': [], 'X': [], 'Y': [], 'Z': []}


def addLettersToMapping(letterMapping, cipherword, candidate):
    # The letterMapping parameter takes a dictionary value that
    # stores a cipherletter mapping, which is copied by the function.
    # The cipherword parameter is a string value of the ciphertext word.
    # The candidate parameter is a possible English word that the
    # cipherword could decrypt to.

    # This function adds the letters in the candidate as potential
    # decryption letters for the cipherletters in the cipherletter
    # mapping.


    for i in range(len(cipherword)):
        if candidate[i] not in letterMapping[cipherword[i]]:
            letterMapping[cipherword[i]].append(candidate[i])



def intersectMappings(mapA, mapB):
    # To intersect two maps, create a blank map and then add only the
    # potential decryption letters if they exist in BOTH maps:
    intersectedMapping = getBlankCipherletterMapping()
    for letter in LETTERS:

```

```
        # An empty list means "any letter is possible". In this case just
        # copy the other map entirely:
        if mapA[letter] == []:
            intersectedMapping[letter] = copy.deepcopy(mapB[letter])
        elif mapB[letter] == []:
            intersectedMapping[letter] = copy.deepcopy(mapA[letter])
        else:
            # If a letter in mapA[letter] exists in mapB[letter],
            # add that letter to intersectedMapping[letter]:
            for mappedLetter in mapA[letter]:
                if mappedLetter in mapB[letter]:
                    intersectedMapping[letter].append(mappedLetter)

    return intersectedMapping


def removeSolvedLettersFromMapping(letterMapping):
    # Cipherletters in the mapping that map to only one letter are
    # "solved" and can be removed from the other letters.
    # For example, if 'A' maps to potential letters ['M', 'N'], and 'B'
    # maps to ['N'], then we know that 'B' must map to 'N', so we can
    # remove 'N' from the list of what 'A' could map to. So 'A' then maps
    # to ['M']. Note that now that 'A' maps to only one letter, we can
    # remove 'M' from the list of letters for every other letter.
    # (This is why there is a loop that keeps reducing the map.)

    loopAgain = True
    while loopAgain:
        # First assume that we will not loop again:
        loopAgain = False

        # solvedLetters will be a list of uppercase letters that have one
        # and only one possible mapping in letterMapping:
        solvedLetters = []
        for cipherletter in LETTERS:
            if len(letterMapping[cipherletter]) == 1:
                solvedLetters.append(letterMapping[cipherletter][0])

        # If a letter is solved, then it cannot possibly be a potential
        # decryption letter for a different ciphertext letter, so we
        # should remove it from those other lists:
        for cipherletter in LETTERS:
            for s in solvedLetters:
                if len(letterMapping[cipherletter]) != 1 and s in
                  letterMapping[cipherletter]:
                    letterMapping[cipherletter].remove(s)
                    if len(letterMapping[cipherletter]) == 1:
                        # A new letter is now solved, so loop again:
                        loopAgain = True
    return letterMapping


def hackSimpleSub(message):
    intersectedMap = getBlankCipherletterMapping()
```

```
    cipherwordList = nonLettersOrSpacePattern.sub('',
      message.upper()).split()
    for cipherword in cipherwordList:
        # Get a new cipherletter mapping for each ciphertext word:
        candidateMap = getBlankCipherletterMapping()

        wordPattern = makeWordPatterns.getWordPattern(cipherword)
        if wordPattern not in wordPatterns.allPatterns:
            continue # This word was not in our dictionary, so continue.

        # Add the letters of each candidate to the mapping:
        for candidate in wordPatterns.allPatterns[wordPattern]:
            addLettersToMapping(candidateMap, cipherword, candidate)

        # Intersect the new mapping with the existing intersected mapping:
        intersectedMap = intersectMappings(intersectedMap, candidateMap)

    # Remove any solved letters from the other lists:
    return removeSolvedLettersFromMapping(intersectedMap)


def decryptWithCipherletterMapping(ciphertext, letterMapping):
    # Return a string of the ciphertext decrypted with the letter mapping,
    # with any ambiguous decrypted letters replaced with an underscore.

    # First create a simple sub key from the letterMapping mapping:
    key = ['x'] * len(LETTERS)
    for cipherletter in LETTERS:
        if len(letterMapping[cipherletter]) == 1:
            # If there's only one letter, add it to the key:
            keyIndex = LETTERS.find(letterMapping[cipherletter][0])
            key[keyIndex] = cipherletter
        else:
            ciphertext = ciphertext.replace(cipherletter.lower(), '_')
            ciphertext = ciphertext.replace(cipherletter.upper(), '_')
    key = ''.join(key)

    # With the key we've created, decrypt the ciphertext:
    return simpleSubCipher.decryptMessage(key, ciphertext)


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

Sample Run of the Simple Substitution Hacking Program

When you run this program, it attempts to hack the ciphertext in the `message` variable. Its output should look like this:

```
Hacking...
Mapping:
{'A': ['E'], 'B': ['Y', 'P', 'B'], 'C': ['R'], 'D': [], 'E': ['W'], 'F':
['B', 'P'], 'G': ['B', 'Q', 'X', 'P', 'Y'], 'H': ['P', 'Y', 'K', 'X', 'B'],
```

```
'I': ['H'], 'J': ['T'], 'K': [], 'L': ['A'], 'M': ['L'], 'N': ['M'], 'O':
['D'], 'P': ['O'], 'Q': ['V'], 'R': ['S'], 'S': ['I'], 'T': ['U'], 'U': ['G'],
'V': [], 'W': ['C'], 'X': ['N'], 'Y': ['F'], 'Z': ['Z']}

Original ciphertext:
Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr sxrjsxwjr, ia esmm
rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa sr pqaceiamnsxu, ia esmm caytra
jp famsaqa sj. Sy, px jia pjiac ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor
l calrpx ypc lwjsxu sx lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax
px jia rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.
-Facjclxo Ctrramm

Copying hacked message to clipboard:
If a man is offered a fact which goes against his instincts, he will
scrutinize it closel_, and unless the evidence is overwhelming, he will refuse
to _elieve it. If, on the other hand, he is offered something which affords
a reason for acting in accordance to his instincts, he will acce_t it even
on the slightest evidence. The origin of m_ths is e__lained in this wa_.
-_ertrand Russell
```

Now let's explore the source code in detail.

Setting Up Modules and Constants

Let's look at the first few lines of the simple substitution hacking program. Line 4 imports seven different modules, more than any other program so far. The global variable `LETTERS` on line 10 stores the symbol set, which consists of the uppercase letters of the alphabet.

```
 1. # Simple Substitution Cipher Hacker
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import os, re, copy, pyperclip, simpleSubCipher, wordPatterns,
      makeWordPatterns
    --snip--
10. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

The `re` module is the regular expression module, which allows for sophisticated string manipulation using regular expressions. Let's look at how regular expressions work.

Finding Characters with Regular Expressions

Regular expressions are strings that define a specific pattern that matches certain strings. For example, the string `'[^A-Z\s]'` on line 11 is a regular expression that tells Python to find any character that is not an uppercase letter from A to Z or a whitespace character (such as a space, tab, or newline character).

```
11. nonLettersOrSpacePattern = re.compile('[^A-Z\s]')
```

The `re.compile()` function creates a regular expression pattern object (abbreviated as *regex object* or *pattern object*) that the `re` module can use. We'll use this object to remove any non-letter characters from the ciphertext in "The `hackSimpleSub()` Function" on page 241.

You can perform many sophisticated string manipulations with regular expressions. To learn more about regular expressions, go to *https://www.nostarch.com/crackingcodes/*.

Setting Up the main() Function

As with the previous hacking programs in this book, the `main()` function stores the ciphertext in the `message` variable, and line 18 passes this variable to the `hackSimpleSub()` function:

```
13. def main():
14.     message = 'Sy l nlx sr pyyacao l ylwj eiswi upar lulsxrj isr
          sxrjsxwjr, ia esmm rwctjsxsza sj wmpramh, lxo txmarr jia aqsoaxwa
          sr pqaceiamnsxu, ia esmm caytra jp famsaqa sj. Sy, px jia pjiac
          ilxo, ia sr pyyacao rpnajisxu eiswi lyypcor l calrpx ypc lwjsxu sx
          lwwpcolxwa jp isr sxrjsxwjr, ia esmm lwwabj sj aqax px jia
          rmsuijarj aqsoaxwa. Jia pcsusx py nhjir sr agbmlsxao sx jisr elh.
          -Facjclxo Ctrramm'
15.
16.     # Determine the possible valid ciphertext translations:
17.     print('Hacking...')
18.     letterMapping = hackSimpleSub(message)
```

Instead of returning the decrypted message or `None` if unable to decrypt it, `hackSimpleSub()` returns an intersected cipherletter mapping with the decrypted letters removed. (We'll look at how to create an intersected mapping in "Intersecting Two Mappings" on page 234.) This intersected cipherletter mapping then gets passed to `decryptWithCipherletterMapping()` to decrypt the ciphertext stored in `message` into a readable format, as you'll see in more detail in "Decrypting the Message" on page 243.

The cipherletter mapping stored in `letterMapping` is a dictionary value that has 26 uppercase single-letter strings as keys that represent the cipherletters. It also lists the uppercase letters of potential decryption letters for each cipherletter as the dictionary's values. When every cipherletter has just one potential decryption letter associated with it, we have a fully solved mapping and can decrypt any ciphertext using the same cipher and key.

Each cipherletter mapping generated depends on the ciphertext used. In some cases, we'll have only a partially solved mapping in which some cipherletters have no potential decryptions and other cipherletters have multiple potential decryptions. Shorter ciphertexts that don't contain every letter of the alphabet are more likely to result in incomplete mappings.

Displaying Hacking Results to the User

The program then calls the `print()` function to display `letterMapping`, the original message, and the decrypted message on the screen:

```
20.     # Display the results to the user:
21.     print('Mapping:')
22.     print(letterMapping)
23.     print()
24.     print('Original ciphertext:')
25.     print(message)
26.     print()
27.     print('Copying hacked message to clipboard:')
28.     hackedMessage = decryptWithCipherletterMapping(message, letterMapping)
29.     pyperclip.copy(hackedMessage)
30.     print(hackedMessage)
```

Line 28 stores the decrypted message in the variable `hackedMessage`, which is copied to the clipboard and printed to the screen so the user can compare it to the original message. We use `decryptWithCipherletterMapping()` to find the decrypted message, which is defined later in the program.

Next, let's look at all the functions that create the cipherletter mappings.

Creating a Cipherletter Mapping

The program needs a cipherletter mapping for each cipherword in the ciphertext. To create a complete mapping, we'll need several helper functions. One of those helper functions will set up a new cipherletter mapping so we can call it for every cipherword.

Another function will take a cipherword, its current letter mapping, and a candidate decryption word to find all the candidate decryption words. We'll call this function for each cipherword and each candidate. The function will then add all the potential decryption letters from the candidate word to the cipherword's letter mapping and return the letter mapping.

When we have letter mappings for several words from the ciphertext, we'll use a function to merge them together. Then, we'll use one final helper function to solve as many cipherletters' decryptions as we can by matching one decryption letter to each cipherletter. As noted, we won't always be able to solve all the cipherletters, but you'll find out how to deal with this issue in "Decrypting the Message" on page 243.

Creating a Blank Mapping

First, we'll need to create a blank cipherletter mapping.

```
33. def getBlankCipherletterMapping():
34.     # Returns a dictionary value that is a blank cipherletter mapping:
35.     return {'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': [],
          'H': [], 'I': [], 'J': [], 'K': [], 'L': [], 'M': [], 'N': [],
```

```
        'O': [], 'P': [], 'Q': [], 'R': [], 'S': [], 'T': [], 'U': [],
        'V': [], 'W': [], 'X': [], 'Y': [], 'Z': []}
```

When called, the `getBlankCipherletterMapping()` function returns a dictionary with the keys set to one-character strings of the 26 letters of the alphabet.

Adding Letters to a Mapping

To add letters to a mapping, we define the `addLettersToMapping()` function on line 38.

```
38. def addLettersToMapping(letterMapping, cipherword, candidate):
```

This function takes three parameters: a cipherletter mapping (`letterMapping`), a cipherword to map (`cipherword`), and a candidate decryption word the cipherword could decrypt to (`candidate`). The function maps every letter in `candidate` to the cipherletter at the corresponding index position in the `cipherword` and adds that letter to `letterMapping` if it isn't already there.

For example, if `'PUPPY'` is the `candidate` for the `cipherword 'HGHHU'`, the `addLettersToMapping()` function adds the value `'P'` to the key `'H'` in `letterMapping`. Then the function moves to the next letter and appends `'U'` to the list value paired with the key `'G'`, and so on.

If the letter is already in the list of potential decryption letters, then `addLettersToMapping()` doesn't add that letter to the list again. For example, in `'PUPPY'` it would skip adding `'P'` to the `'H'` key for the next two instances of `'P'` because it's already there. Finally, the function changes the value for key `'U'` so it has `'Y'` in its list of potential decryption letters.

The code in `addLettersToMapping()` assumes that `len(cipherword)` is the same as `len(candidate)` because we should only pass a `cipherword` and `candidate` pair with matching word patterns.

Then the program iterates over each index in the string in `cipherword` to check if a letter has already been added to the list of potential decryption letters:

```
50.     for i in range(len(cipherword)):
51.         if candidate[i] not in letterMapping[cipherword[i]]:
52.             letterMapping[cipherword[i]].append(candidate[i])
```

We'll use the variable `i` to iterate through each letter of `cipherword` and its corresponding potential decryption letter in `candidate` through indexing. We can do this because the potential decryption letter to be added is `candidate[i]` for the cipherletter `cipherword[i]`. For example, if the `cipherword` was `'HGHHU'` and the `candidate` was `'PUPPY'`, `i` would start at index `0`, and we would use `cipherword[0]` and `candidate[0]` to access the first letters in each string. Then the execution would move on to the `if` statement on line 51.

The `if` statement checks that the potential decryption letter, `candidate[i]`, is not already in the list of potential decryption letters for the cipherletter

and doesn't add it if it's already in the list. It does this by accessing the cipherletter in the mapping with `letterMapping[cipherword[i]]`, because `cipherword[i]` is the key in `letterMapping` that needs to be accessed. This check prevents duplicate letters in the list of potential decryption letters.

For example, the first `'P'` in `'PUPPY'` might be added to the `letterMapping` at the first iteration of the loop, but when `i` is equal to `2` in the third iteration, the `'P'` from `candidate[2]` wouldn't be added to the mapping because it was already added at the first iteration.

If the potential decryption letter isn't already in the mapping, line 52 adds the new letter, `candidate[i]`, to the list of potential decryption letters in the cipherletter mapping at `letterMapping[cipherword[i]]`.

Recall that because Python passes a copy of the reference to a dictionary passed for the parameter, instead of a copy of the dictionary itself, any changes made to `letterMapping` in this function will be done outside the `addLettersToMapping()` function as well. This is because both copies of the reference still refer to the same dictionary passed for the `letterMapping` parameter in the call to `addLettersToMapping()` on line 126.

After looping through all the indexes in `cipherword`, the function is done adding letters to the mapping in the `letterMapping` variable. Now let's look at how the program compares this mapping to that of other cipherwords to check for overlaps.

Intersecting Two Mappings

The `hackSimpleSub()` function uses the `intersectMappings()` function to take two cipherletter mappings passed as its `mapA` and `mapB` parameters and return a merged mapping of `mapA` and `mapB`. The `intersectMappings()` function instructs the program to combine `mapA` and `mapB`, create a blank map, and then add the potential decryption letters to the blank map only if they exist in *both* maps to prevent duplicates.

```
56. def intersectMappings(mapA, mapB):
57.     # To intersect two maps, create a blank map and then add only the
58.     # potential decryption letters if they exist in BOTH maps:
59.     intersectedMapping = getBlankCipherletterMapping()
```

First, line 59 creates a cipherletter mapping to store the merged mapping by calling `getBlankCipherletterMapping()` and storing the returned value in the `intersectedMapping` variable.

The `for` loop on line 60 loops through the uppercase letters in the `LETTERS` constant variable and uses the `letter` variable as the keys of the `mapA` and `mapB` dictionaries:

```
60.     for letter in LETTERS:
61.
62.         # An empty list means "any letter is possible". In this case just
63.         # copy the other map entirely:
64.         if mapA[letter] == []:
65.             intersectedMapping[letter] = copy.deepcopy(mapB[letter])
```

```
        elif mapB[letter] == []:
            intersectedMapping[letter] = copy.deepcopy(mapA[letter])
```

Line 64 checks whether the list of potential decryption letters for `mapA` is blank. A blank list means that this cipherletter could potentially decrypt to *any* letter. In this case, the intersected cipherletter mapping just copies the *other* mapping's list of potential decryption letters. For example, if the list of potential decryption letters in `mapA` is blank, then line 65 sets the intersected mapping's list to be a copy of the list in `mapB`, and vice versa on line 67. Note that if both mappings' lists are blank, the condition on line 64 is still `True`, and then line 65 simply copies the blank list in `mapB` to the intersected mapping.

The `else` block on line 68 handles the case in which neither `mapA` nor `mapB` is blank:

```
        else:
            # If a letter in mapA[letter] exists in mapB[letter],
            # add that letter to intersectedMapping[letter]:
            for mappedLetter in mapA[letter]:
                if mappedLetter in mapB[letter]:
                    intersectedMapping[letter].append(mappedLetter)

    return intersectedMapping
```

When the maps are not blank, line 71 loops through the uppercase letter strings in the list at `mapA[letter]`. Line 72 checks whether the uppercase letter in `mapA[letter]` also exists in the list of uppercase letter strings in `mapB[letter]`. If it does, then `intersectedMapping[letter]` on line 73 adds this common letter to the list of potential decryption letters.

After the `for` loop that started on line 60 has finished, the cipherletter mapping in `intersectedMapping` should only have the potential decryption letters that exist in the lists of potential decryption letters of both `mapA` and `mapB`. Line 75 returns this completely intersected cipherletter mapping. Next, let's look at an example output of an intersected mapping.

How the Letter-Mapping Helper Functions Work

Now that we've defined the letter-mapping helper functions, let's try using them in the interactive shell to better understand how they work together. Let's create an intersected cipherletter map for the ciphertext `'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC'`, which contains just three cipherwords. We'll do this by creating a mapping for each word and then combining the mappings.

Import *simpleSubHacker.py* into the interactive shell:

```
>>> import simpleSubHacker
```

Next, we call `getBlankCipherletterMapping()` to create a blank letter mapping and store this mapping in a variable named `letterMapping1`:

```
>>> letterMapping1 = simpleSubHacker.getBlankCipherletterMapping()
>>> letterMapping1
{'A': [], 'C': [], 'B': [], 'E': [], 'D': [], 'G': [], 'F': [], 'I': [],
'H': [], 'K': [], 'J': [], 'M': [], 'L': [], 'O': [], 'N': [], 'Q': [],
'P': [], 'S': [], 'R': [], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [],
'X': [], 'Z': []}
```

Let's start hacking the first cipherword, `'OLQIHXIRCKGNZ'`. First, we need to get the word pattern for this cipherword by calling the `makeWordPattern` module's `getWordPattern()` function, as shown here:

```
>>> import makeWordPatterns
>>> makeWordPatterns.getWordPattern('OLQIHXIRCKGNZ')
0.1.2.3.4.5.3.6.7.8.9.10.11
```

To figure out which English words in the dictionary have the word pattern 0.1.2.3.4.5.3.6.7.8.9.10.11 (that is, to figure out the candidates for the cipherword `'OLQIHXIRCKGNZ'`), we import the `wordPatterns` module and look up this pattern:

```
>>> import wordPatterns
>>> candidates = wordPatterns.allPatterns['0.1.2.3.4.5.3.6.7.8.9.10.11']
>>> candidates
['UNCOMFORTABLE', 'UNCOMFORTABLY']
```

Two English words match the word pattern for `'OLQIHXIRCKGNZ'`; therefore, the only two words the first cipherword could decrypt to are `'UNCOMFORTABLE'` and `'UNCOMFORTABLY'`. These words are our candidates, so we'll store them in the `candidates` variable (not to be confused with the `candidate` parameter in the `addLettersToMapping()` function) as a list.

Next, we need to map their letters to the `cipherword`'s letters using `addLettersToMapping()`. First, we'll map `'UNCOMFORTABLE'` by accessing the first member of the `candidates` list, like so:

```
>>> letterMapping1 = simpleSubHacker.addLettersToMapping(letterMapping1,
'OLQIHXIRCKGNZ', candidates[0])
>>> letterMapping1
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [], 'I':
['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'], 'N':
['L'], 'Q': ['C'], 'P': [], 'S': [], 'R': ['R'], 'U': [], 'T': [], 'W': [],
'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

From the `letterMapping1` value, you can see that the letters in `'OLQIHXIRCKGNZ'` map to the letters in `'UNCOMFORTABLE'`: `'O'` maps to `['U']`, `'L'` maps to `['N']`, `'Q'` maps to `['C']`, and so on.

But because the letters in 'OLQIHXIRCKGNZ' could also possibly decrypt to 'UNCOMFORTABLY', we also need to add it to the cipherletter mapping. Enter the following into the interactive shell:

```
>>> letterMapping1 = simpleSubHacker.addLettersToMapping(letterMapping1,
'OLQIHXIRCKGNZ', candidates[1])
>>> letterMapping1
{'A': [], 'C': ['T'], 'B': [], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'],
'N': ['L'], 'Q': ['C'], 'P': [], 'S': [], 'R': ['R'], 'U': [], 'T': [],
'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E', 'Y']}
```

Notice that not much has changed in `letterMapping1` except the cipherletter mapping in `letterMapping1` now has `'Z'` map to `'Y'` in addition to `'E'`. That's because `addLettersToMapping()` adds the letter to the list only if the letter is not already there.

Now we have a cipherletter mapping for the first of the three cipherwords. We need to get a new mapping for the second cipherword, `'PLQRZKBZB'`, and repeat the process:

```
>>> letterMapping2 = simpleSubHacker.getBlankCipherletterMapping()
>>> wordPat = makeWordPatterns.getWordPattern('PLQRZKBZB')
>>> candidates = wordPatterns.allPatterns[wordPat]
>>> candidates
['CONVERSES', 'INCREASES', 'PORTENDED', 'UNIVERSES']
>>> for candidate in candidates:
...   letterMapping2 = simpleSubHacker.addLettersToMapping(letterMapping2,
'PLQRZKBZB', candidate)
...
>>> letterMapping2
{'A': [], 'C': [], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': [], 'F': [], 'I':
[], 'H': [], 'K': ['R', 'A', 'N'], 'J': [], 'M': [], 'L': ['O', 'N'], 'O': [],
'N': [], 'Q': ['N', 'C', 'R', 'I'], 'P': ['C', 'I', 'P', 'U'], 'S': [], 'R':
['V', 'R', 'T'], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': [], 'Z':
['E']}
```

Instead of entering four calls to `addLettersToMapping()` for each of these four candidate words, we can write a `for` loop that goes through the list in `candidates` and calls `addLettersToMapping()` on each of them. This finishes the cipherletter mapping for the second cipherword.

Next, we need to get the intersection of the cipherletter mappings in `letterMapping1` and `letterMapping2` by passing them to `intersectMappings()`. Enter the following into the interactive shell:

```
>>> intersectedMapping = simpleSubHacker.intersectMappings(letterMapping1,
letterMapping2)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S', 'D'], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': [], 'L': ['N'], 'O': ['U'],
'N': ['L'], 'Q': ['C'], 'P': ['C', 'I', 'P', 'U'], 'S': [], 'R': ['R'],
'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

Now the list of potential decryption letters for any cipherletter in the intersected mapping should be only the potential decryption letters that are in *both* `letterMapping1` and `letterMapping2`.

For example, the list in `intersectedMapping` for the `'Z'` key is just `['E']` because `letterMapping1` had `['E', 'Y']` but `letterMapping2` had only `['E']`.

Next, we repeat all the preceding steps for the third cipherword, `'MPBKSSIPLC'`, as follows:

```
>>> letterMapping3 = simpleSubHacker.getBlankCipherletterMapping()
>>> wordPat = makeWordPatterns.getWordPattern('MPBKSSIPLC')
>>> candidates = wordPatterns.allPatterns[wordPat]
>>> for i in range(len(candidates)):
...   letterMapping3 = simpleSubHacker.addLettersToMapping(letterMapping3,
'MPBKSSIPLC', candidates[i])
...
>>> letterMapping3
{'A': [], 'C': ['Y', 'T'], 'B': ['M', 'S'], 'E': [], 'D': [], 'G': [],
'F': [], 'I': ['E', 'O'], 'H': [], 'K': ['I', 'A'], 'J': [], 'M': ['A', 'D'],
'L': ['L', 'N'], 'O': [], 'N': [], 'Q': [], 'P': ['D', 'I'], 'S': ['T', 'P'],
'R': [], 'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': [], 'Z': []}
```

Enter the following into the interactive shell to intersect `letterMapping3` with `intersectedMapping`, which is the intersected mapping of `letterMapping1` and `letterMapping2`:

```
>>> intersectedMapping = simpleSubHacker.intersectMappings(intersectedMapping,
letterMapping3)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['A', 'D'], 'L': ['N'],
'O': ['U'], 'N': ['L'], 'Q': ['C'], 'P': ['I'], 'S': ['T', 'P'], 'R': ['R'],
'U': [], 'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

In this example, we're able to find solutions for the keys that have only one value in their list. For example, `'K'` decrypts to `'A'`. But notice that key `'M'` could decrypt to `'A'` or `'D'`. Because we know that `'K'` decrypts to `'A'`, we can deduce that key `'M'` must decrypt to `'D'`, not `'A'`. After all, if the solved letter is used by one cipherletter, it can't be used by another cipherletter, because the simple substitution cipher encrypts a plaintext letter to exactly one cipherletter.

Let's look at how the `removeSolvedLettersFromMapping()` function finds these solved letters and removes them from the list of potential decryption letters. We'll need the `intersectedMapping` we just created, so don't close the IDLE window just yet.

Identifying Solved Letters in Mappings

The `removeSolvedLettersFromMapping()` function searches for any cipherletters in the `letterMapping` parameter that have only one potential decryption letter. These cipherletters are considered solved, which means that any other cipherletters with this solved letter in their list of potential decryption

letters can't possibly decrypt to this letter. This could cause a chain reaction, because when one potential decryption letter is removed from other lists of potential decryption letters holding just two letters, the result could be a new solved cipherletter. The program handles this situation by looping and removing the newly solved letter from the entire cipherletter mapping.

```
78. def removeSolvedLettersFromMapping(letterMapping):
        --snip--
88.     loopAgain = True
89.     while loopAgain:
90.         # First assume that we will not loop again:
91.         loopAgain = False
```

Because a reference to a dictionary is passed for the `letterMapping` parameter, that dictionary will contain the changes made in the function `removeSolvedLettersFromMapping()` even after the function returns. Line 88 creates `loopAgain`, a variable that holds a Boolean value, which determines whether the code needs to loop again when it finds another solved letter.

If the `loopAgain` variable is set to `True` on line 88, the program execution enters the `while` loop on line 89. At the beginning of the loop, line 91 sets `loopAgain` to `False`. The code assumes that this is the last iteration through the `while` loop on line 89. The `loopAgain` variable is only set to `True` if the program finds a new solved cipherletter during this iteration.

The next part of the code creates a list of cipherletters that have exactly one potential decryption letter. These are the solved letters that will be removed from the mapping.

```
93.         # solvedLetters will be a list of uppercase letters that have one
94.         # and only one possible mapping in letterMapping:
95.         solvedLetters = []
96.         for cipherletter in LETTERS:
97.             if len(letterMapping[cipherletter]) == 1:
98.                 solvedLetters.append(letterMapping[cipherletter][0])
```

The `for` loop on line 96 goes through all 26 possible cipherletters and looks at the cipherletter mapping's list of potential decryption letters for that cipherletter (that is, the list at `letterMapping[cipherletter]`).

Line 97 checks whether the length of this list is `1`. If it is, we know there's only one letter that the cipherletter could decrypt to and the cipherletter is solved. Line 98 adds the solved decryption letter to the `solvedLetters` list. The solved letter is always at `letterMapping[cipherletter][0]` because `letterMapping[cipherletter]` is a list of potential decryption letters that has only one string value in it at index `0` of the list.

After the previous `for` loop that started on line 96 has finished, the `solvedLetters` variable should contain a list of all the decryptions of a cipherletter. Line 98 stores these decrypted strings in `solvedLetters` as a list.

At this point, the program is done identifying all the solved letters. Then it checks whether they're listed as potential decryption letters for other cipherletters and removes them.

To do this, the `for` loop on line 103 loops through all 26 possible cipherletters and looks at the cipherletter mapping's list of potential decryption letters.

```
103.         for cipherletter in LETTERS:
104.             for s in solvedLetters:
105.                 if len(letterMapping[cipherletter]) != 1 and s in
                       letterMapping[cipherletter]:
106.                   letterMapping[cipherletter].remove(s)
107.                   if len(letterMapping[cipherletter]) == 1:
108.                       # A new letter is now solved, so loop again:
109.                       loopAgain = True
110.     return letterMapping
```

For each cipherletter examined, line 104 loops through the letters in `solvedLetters` to check whether any of them exists in the list of potential decryption letters for `letterMapping[cipherletter]`.

Line 105 checks whether a list of potential decryption letters isn't solved by checking whether `len(letterMapping[cipherletter]) != 1` *and* by checking whether the solved letter exists in the list of potential decryption letters. If both criteria are met, this condition returns `True`, and line 106 removes the solved letter in `s` from the list of potential decryption letters.

If this removal leaves only one letter in the list of potential decryption letters, line 109 sets the `loopAgain` variable to `True` so the code can remove this newly solved letter from the cipherletter mapping on the next iteration of the loop.

After the `while` loop on line 89 has gone through a full iteration without `loopAgain` being set to `True`, the program moves beyond the loop and line 110 returns the cipherletter mapping stored in `letterMapping`.

The variable `letterMapping` should now contain a partially or potentially fully solved cipherletter mapping.

Testing the removeSolvedLetterFromMapping() Function

Let's see `removeSolvedLetterFromMapping()` in action by testing it in the interactive shell. Return to the interactive shell window you had open when you created `intersectedMapping`. (If you closed the window, don't worry; you can just reenter the instructions in "How the Letter-Mapping Helper Functions Work" on page 235 and then follow along with this example.)

To remove the solved letters from `intersectedMapping`, enter the following into the interactive shell:

```
>>> letterMapping = simpleSubHacker.removeSolvedLettersFromMapping(
intersectedMapping)
>>> intersectedMapping
{'A': [], 'C': ['T'], 'B': ['S'], 'E': [], 'D': [], 'G': ['B'], 'F': [],
'I': ['O'], 'H': ['M'], 'K': ['A'], 'J': [], 'M': ['D'], 'L': ['N'], 'O':
['U'], 'N': ['L'], 'Q': ['C'], 'P': ['I'], 'S': ['P'], 'R': ['R'], 'U': [],
'T': [], 'W': [], 'V': [], 'Y': [], 'X': ['F'], 'Z': ['E']}
```

When you remove the solved letters from intersectedMapping, notice that 'M' now has just one potential decryption letter, 'D', which is what we predicted would be the case. Now each cipherletter has just one potential decryption letter, so we can use the cipherletter mapping to start decrypting. We'll need to return to this interactive shell example one more time, so keep its window open.

The hackSimpleSub() Function

Now that you've seen how the functions getBlankCipherletterMapping(), addLettersToMapping(), intersectMappings(), and removeSolvedLettersFromMapping() manipulate the cipherletter mappings you pass them, let's use them in our *simpleSubHacker.py* program to decrypt a message.

Line 113 defines the hackSimpleSub() function, which takes a ciphertext message and uses the letter-mapping helper functions to return a partially or fully solved cipherletter mapping:

```
113. def hackSimpleSub(message):
114.     intersectedMap = getBlankCipherletterMapping()
115.     cipherwordList = nonLettersOrSpacePattern.sub('',
           message.upper()).split()
```

On line 114, we create a new cipherletter mapping that we store in the intersectedMap variable. This variable will eventually hold the intersected mappings of each of the cipherwords.

On line 115, we remove any non-letter characters from message. The regex object in nonLettersOrSpacePattern matches any string that isn't a letter or whitespace character. The sub() method is called on a regular expression and takes two arguments. The function searches the string in the second argument for matches, and it replaces those matches with the string in the first argument. Then it returns a string with all these replacements. In this example, the sub() method tells the program to go through the uppercased message string and replace all the non-letter characters with a blank string (''). This makes sub() return a string with all punctuation and number characters removed, and this string is stored in the cipherwordList variable.

After line 115 executes, the cipherwordList variable should contain a list of uppercase strings of the individual cipherwords previously in message.

The for loop on line 116 assigns each string in the message list to the cipherword variable. Inside this loop, the code creates a blank map, gets the cipherword's candidates, adds the candidates' letters to a cipherletter mapping, and then intersects this mapping with intersectedMap.

```
116.     for cipherword in cipherwordList:
117.         # Get a new cipherletter mapping for each ciphertext word:
118.         candidateMap = getBlankCipherletterMapping()
120.         wordPattern = makeWordPatterns.getWordPattern(cipherword)
121.         if wordPattern not in wordPatterns.allPatterns:
122.             continue # This word was not in our dictionary, so continue.
```

```
        # Add the letters of each candidate to the mapping:
        for candidate in wordPatterns.allPatterns[wordPattern]:
            addLettersToMapping(candidateMap, cipherword, candidate)
        # Intersect the new mapping with the existing intersected mapping:
        intersectedMap = intersectMappings(intersectedMap, candidateMap)
```

Line 118 gets a new, blank cipherletter mapping from the function `getBlankCipherletterMapping()` and stores it in the `candidateMap` variable.

To find the candidates for the current cipherword, line 120 calls `getWordPattern()` in the `makeWordPatterns` module. In some cases, the cipherword may be a name or a very uncommon word that doesn't exist in the dictionary, in which case, its word pattern likely won't exist in `wordPatterns` either. If the word pattern of the cipherword doesn't exist in the keys of the `wordPatterns.allPatterns` dictionary, the original plaintext word doesn't exist in the dictionary file. In that case, the cipherword doesn't get a mapping, and the `continue` statement on line 122 returns to the next cipherword in the list on line 116.

If the execution reaches line 125, we know the word pattern exists in `wordPatterns.allPatterns`. The values in the `allPatterns` dictionary are lists of strings of the English words with the pattern in `wordPattern`. Because the values are in the form of a list, we use a `for` loop to iterate over them. The variable `candidate` is set to each of these English word strings on each iteration of the loop.

The `for` loop on line 125 calls `addLettersToMapping()` on line 126 to update the cipherletter mapping in `candidateMap` using the letters in each of the candidates. The `addLettersToMapping()` function modifies the list directly, so `candidateMap` is modified by the time the function call returns.

After all the letters in the candidates are added to the cipherletter mapping in `candidateMap`, line 129 intersects `candidateMap` with `intersectedMap` and returns the new value of `intersectedMap`.

At this point, the program execution returns to the beginning of the `for` loop on line 116 to create a new mapping for the next cipherword in the `cipherwordList` list, and the mapping for the next cipherword is also intersected with `intersectedMap`. The loop continues mapping cipherwords until it reaches the last word in `cipherWordList`.

When we have the final intersected cipherletter mapping that contains the mappings of all the cipherwords in the ciphertext, we pass it to `removeSolvedLettersFromMapping()` on line 132 to remove any solved letters.

```
    # Remove any solved letters from the other lists:
    return removeSolvedLettersFromMapping(intersectedMap)
```

The cipherletter mapping returned from `removeSolvedLettersFromMapping()` is then returned for the `hackSimpleSub()` function. Now we have part of the cipher's solution, so we can start decrypting the message.

The replace() String Method

The `replace()` string method returns a new string with replaced characters. The first argument is the substring to look for, and the second argument is the string to replace those substrings with. Enter the following into the interactive shell to see an example:

```
>>> 'mississippi'.replace('s', 'X')
'miXXiXXippi'
>>> 'dog'.replace('d', 'bl')
'blog'
>>> 'jogger'.replace('ger', 's')
'jogs'
```

We'll use the `replace()` string method in `decryptMessage()` in the *simpleSubHacker.py* program.

Decrypting the Message

To decrypt our message, we'll use the function `simpleSubstitutionCipher.decryptMessage()` that we already programmed in *simpleSubstitutionCipher.py*. But `simpleSubstitutionCipher.decryptMessage()` decrypts using keys only, not letter mappings, so we can't use the function directly. To address this issue, we'll create a `decryptWithCipherletterMapping()` function that takes a letter mapping, converts the mapping into a key, and then passes the key and message to `simpleSubstitutionCipher.decryptMessage()`. The function `decryptWithCipherletterMapping()` will return a decrypted string. Recall that the simple substitution keys are strings of 26 characters and the character at index `0` in the key string is the encrypted character for A, the character at index `1` is the encrypted character for B, and so on.

To convert a mapping into a decryption output we can read easily, we'll need to first create a placeholder key, which will look like this: `['x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x', 'x']`. The lowercase `'x'` can be used in the placeholder key because the actual key uses only uppercase letters. (You can use any character that isn't an uppercase letter as a placeholder.) Because not all the letters will have a decryption, we need to be able to distinguish between parts of the key list that have been filled with the decryption letters and those where the decryption hasn't been solved. The `'x'` indicates letters that haven't been solved.

Let's see how this all comes together in the source code:

```
135. def decryptWithCipherletterMapping(ciphertext, letterMapping):
136.     # Return a string of the ciphertext decrypted with the letter mapping,
137.     # with any ambiguous decrypted letters replaced with an underscore.
138.
139.     # First create a simple sub key from the letterMapping mapping:
140.     key = ['x'] * len(LETTERS)
141.     for cipherletter in LETTERS:
142.         if len(letterMapping[cipherletter]) == 1:
```

```
143.             # If there's only one letter, add it to the key:
144.             keyIndex = LETTERS.find(letterMapping[cipherletter][0])
145.             key[keyIndex] = cipherletter
```

Line 140 creates the placeholder list by replicating the single-item list `['x']` 26 times. Because `LETTERS` is a string of the letters of the alphabet, `len(LETTERS)` evaluates to `26`. When used on a list and integer, the multiplication operator (`*`) performs list replication.

The `for` loop on line 141 checks each of the letters in `LETTERS` for the `cipherletter` variable, and if the cipherletter is solved (that is, `letterMapping[cipherletter]` has only one letter in it), it replaces the `'x'` placeholder with that letter.

The `letterMapping[cipherletter][0]` on line 144 is the decryption letter, and `keyIndex` is the index of the decryption letter in `LETTERS`, which is returned from the `find()` call. Line 145 sets this index in the key list to the decryption letter.

However, if the cipherletter doesn't have a solution, the function inserts an underscore for that cipherletter to indicate which characters remain unsolved. Line 147 replaces the lowercase letters in `cipherletter` with an underscore, and line 148 replaces the uppercase letters with an underscore:

```
146.         else:
147.             ciphertext = ciphertext.replace(cipherletter.lower(), '_')
148.             ciphertext = ciphertext.replace(cipherletter.upper(), '_')
```

After replacing all the parts in the list in `key` with the solved letters, the function combines the list of strings into a single string using the `join()` method to create a simple substitution key. This string is passed to the `decryptMessage()` function in the *simpleSubCipher.py* program.

```
149.     key = ''.join(key)
150.
151.     # With the key we've created, decrypt the ciphertext:
152.     return simpleSubCipher.decryptMessage(key, ciphertext)
```

Finally, line 152 returns the decrypted message string from the `decryptMessage()` function. We now have all the functions we need to find an intersected letter mapping, hack a key, and decrypt a message. Let's look at a quick example of how these functions work in the interactive shell.

Decrypting in the Interactive Shell

Let's return to the example we used in "How the Letter-Mapping Helper Functions Work" on page 235. We'll use the `intersectedMapping` variable we created in our earlier shell examples to decrypt the ciphertext message `'OLQIHXIRCKGNZ PLQRZKBZB MPBKSSIPLC'`.

Enter the following into the interactive shell:

```
>>> simpleSubHacker.decryptWithCipherletterMapping('OLQIHXIRCKGNZ PLQRZKBZB
MPBKSSIPLC', intersectedMapping)
UNCOMFORTABLE INCREASES DISAPPOINT
```

The ciphertext decrypts to the message "Uncomfortable increases disappoint". As you can see, the `decryptWithCipherletterMapping()` function worked perfectly and returned the fully decrypted string. But this example doesn't show what happens when we don't have all the letters that appear in the ciphertext solved. To see what happens when we're missing a cipherletter's decryption, let's remove the solution for the cipherletters `'M'` and `'S'` from `intersectedMapping` by using the following instructions:

```
>>> intersectedMapping['M'] = []
>>> intersectedMapping['S'] = []
```

Then try to decrypt the ciphertext with `intersectedMapping` again:

```
>>> simpleSubHacker.decryptWithCipherletterMapping('OLQIHXIRCKGNZ PLQRZKBZB
MPBKSSIPLC', intersectedMapping)
UNCOMFORTABLE INCREASES _ISA__OINT
```

This time, part of the ciphertext wasn't decrypted. The cipherletters without a decryption letter were replaced with underscores.

This is a rather short ciphertext to hack. Normally, encrypted messages would be much longer. (This example was specifically chosen to be hackable. Messages as short as this example usually cannot be hacked using the word pattern method.) To hack longer encryptions, you'll need to create a cipherletter mapping for each cipherword in the longer messages and then intersect them all together. The `hackSimpleSub()` function calls the other functions in our program to do exactly this.

Calling the main() Function

Lines 155 and 156 call the `main()` function to run *simpleSubHacker.py* if it's being run directly instead of being imported as a module by another Python program:

```
155. if __name__ == '__main__':
156.     main()
```

That completes our discussion of all the functions the *simpleSubHacker.py* program uses.

NOTE *Our hacking approach works only if the spaces are not encrypted. You can expand the symbol set so the cipher program encrypts spaces, numbers, and punctuation characters as well as letters, making your encrypted messages even harder (but not impossible) to hack. Hacking such messages would involve updating the frequencies of not just letters, but all the symbols in the symbol set. This makes hacking more complicated, which is the reason this book encrypted letters only.*

Summary

Whew! The *simpleSubHacker.py* program is fairly complicated. You learned how to use cipherletter mapping to model the possible decryption letters for each ciphertext letter. You also learned how to narrow down the number of possible keys by adding potential letters to the mapping, intersecting them, and removing solved letters from other lists of potential decryption letters. Instead of brute-forcing 403,291,461,126,605,635,584,000,000 possible keys, you can use some sophisticated Python code to figure out most (if not all) of the original simple substitution key.

The main advantage of the simple substitution cipher is its large number of possible keys. The disadvantage is that it's relatively easy to compare cipherwords to words in a dictionary file to determine which cipherletters decrypt to which letters. In Chapter 18, we'll explore a more powerful polyalphabetic substitution cipher called the Vigenère cipher, which was considered impossible to break for several hundred years.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. What is the word pattern for the word *hello*?
2. Do *mammoth* and *goggles* have the same word pattern?
3. Which word could be the possible plaintext word for the cipherword PYYACAO? *Alleged*, *efficiently*, or *poodle*?

18

PROGRAMMING THE VIGENÈRE CIPHER

"I believed then, and continue to believe now, that the benefits to our security and freedom of widely available cryptography far, far outweigh the inevitable damage that comes from its use by criminals and terrorists."
—Matt Blaze, AT&T Labs, September 2001

The Italian cryptographer Giovan Battista Bellaso was the first person to describe the Vigenère cipher in 1553, but it was eventually named after the French diplomat Blaise de Vigenère, one of many people who reinvented the cipher in subsequent years. It was known as "le chiffre indéchiffrable," which means "the indecipherable cipher," and remained unbroken until British polymath Charles Babbage broke it in the 19th century.

Because the Vigenère cipher has too many possible keys to brute-force, even with our English detection module, it's one of the strongest ciphers discussed so far in this book. It's even invincible to the word pattern attack you learned in Chapter 17.

TOPICS COVERED IN THIS CHAPTER

- Subkeys
- Building strings using the list-append-join process

Using Multiple Letter Keys in the Vigenère Cipher

Unlike the Caesar cipher, the Vigenère cipher has multiple keys. Because it uses more than one set of substitutions, the Vigenère cipher is a *polyalphabetic substitution cipher*. Unlike with the simple substitution cipher, frequency analysis alone will not defeat the Vigenère cipher. Instead of using a numeric key between 0 and 25 as we did in the Caesar cipher, we use a letter key for the Vigenère.

The Vigenère key is a series of letters, such as a single English word, that is split into multiple single-letter subkeys that encrypt letters in the plaintext. For example, if we use a Vigenère key of PIZZA, the first subkey is P, the second subkey is I, the third and fourth subkeys are both Z, and the fifth subkey is A. The first subkey encrypts the first letter of the plaintext, the second subkey encrypts the second letter, and so on. When we get to the sixth letter of the plaintext, we return to the *first* subkey.

Using the Vigenère cipher is the same as using multiple Caesar ciphers, as shown in Figure 18-1. Instead of encrypting the whole plaintext with one Caesar cipher, we apply a different Caesar cipher to each letter of the plaintext.

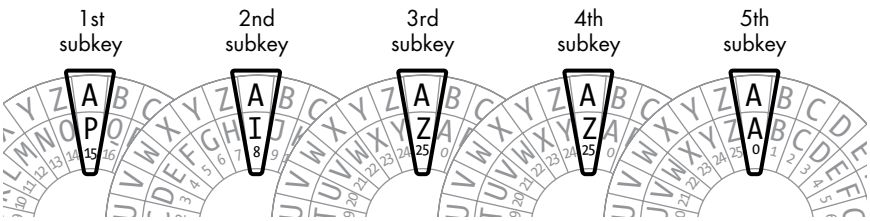


Figure 18-1: Multiple Caesar ciphers combine to make the Vigenère cipher

Each subkey is converted into an integer and serves as a Caesar cipher key. For example, the letter A corresponds to the Caesar cipher key 0. The letter B corresponds to key 1, and so on up to Z for key 25, as shown in Figure 18-2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Figure 18-2: Caesar cipher keys and their corresponding letters

Let's look at an example. The following is the message COMMON SENSE IS NOT SO COMMON shown alongside the Vigenère key PIZZA. The plaintext is shown with the corresponding subkey that encrypts each letter underneath it.

```
COMMONSENSEISNOTSOCOMMON
PIZZAPIZZAPIZZAPIZZAPIZZ
```

To encrypt the first C in the plaintext with the subkey P, encrypt it with the Caesar cipher using the subkey's corresponding numeric key 15, which results in the cipherletter R, and repeat the process for each letter of the plaintext by cycling through the subkeys. Table 18-1 shows this process. The integer for the plaintext letter and subkey (given in parentheses) are added together to produce the integer for the ciphertext letter.

Table 18-1: Encrypting Letters with Vigenère Subkeys

Plaintext letter	Subkey	Ciphertext letter	Plaintext letter	Subkey	Ciphertext letter
C (2)	P (15)	R (17)	S (18)	Z (25)	R (17)
O (14)	I (8)	W (22)	N (13)	Z (25)	M (12)
M (12)	Z (25)	L (11)	O (14)	A (0)	O (14)
M (12)	Z (25)	L (11)	T (19)	P (15)	I (8)
O (14)	A (0)	O (14)	S (18)	I (8)	A (0)
N (13)	P (15)	C (2)	O (14)	Z (25)	N (13)
S (18)	I (8)	A (0)	C (2)	Z (25)	B (1)
E (4)	Z (25)	D (3)	O (14)	A (0)	O (14)
N (13)	Z (25)	M (12)	M (12)	P (15)	B (1)
S (18)	A (0)	S (18)	M (12)	I (8)	U (20)
E (4)	P (15)	T (19)	O (14)	Z (25)	N (13)
I (8)	I (8)	Q (16)	N (13)	Z (25)	M (12)

Using the Vigenère cipher with the key PIZZA (which is made up of the subkeys 15, 8, 25, 25, 0) encrypts the plaintext COMMON SENSE IS NOT SO COMMON into the ciphertext RWLLOC ADMST QR MOI AN BOBUNM.

Longer Vigenère Keys Are More Secure

The more letters in the Vigenère key, the stronger the encrypted message will be against a brute-force attack. PIZZA is a poor choice for a Vigenère key because it has only five letters. A key with five letters has 11,881,376 possible combinations (because 26 letters to the power of 5 is $26^5 = 26 \times 26 \times 26 \times 26 \times 26 = 11{,}881{,}376$). Eleven million keys are far too many for a human to brute-force, but a computer can try them all in just a few hours.

It would first try to decrypt the message using the key AAAAA and check whether the resulting decryption was in English. Then it could try AAAAB, then AAAAC, and so on until it got to PIZZA.

The good news is that for every additional letter the key has, the number of possible keys multiplies by 26. Once there are quadrillions of possible keys, the cipher would take a computer many years to break. Table 18-2 shows how many possible keys there are for each key length.

Table 18-2: Number of Possible Keys Based on Vigenère Key Length

Key length	Equation	Possible keys
1	26	= 26
2	26 × 26	= 676
3	676 × 26	= 17,576
4	17,576 × 26	= 456,976
5	456,976 × 26	= 11,881,376
6	11,881,376 × 26	= 308,915,776
7	308,915,776 × 26	= 8,031,810,176
8	8,031,810,176 × 26	= 208,827,064,576
9	208,827,064,576 × 26	= 5,429,503,678,976
10	5,429,503,678,976 × 26	= 141,167,095,653,376
11	141,167,095,653,376 × 26	= 3,670,344,486,987,776
12	3,670,344,486,987,776 × 26	= 95,428,956,661,682,176
13	95,428,956,661,682,176 × 26	= 2,481,152,873,203,736,576
14	2,481,152,873,203,736,576 × 26	= 64,509,974,703,297,150,976

With keys that are twelve or more letters long, it becomes impossible for a mere laptop to crack them in a reasonable amount of time.

Choosing a Key That Prevents Dictionary Attacks

A Vigenère key doesn't have to be a real word like PIZZA. It can be any combination of letters of any length, such as the twelve-letter key DURIWKNMFICK. In fact, not using a word that can be found in the dictionary is best. Even though the word RADIOLOGISTS is also a twelve-letter key that is easier to remember than DURIWKNMFICK, a cryptanalyst might anticipate that the cryptographer is using an English word as a key.

Attempting a brute-force attack using every English word in the dictionary is known as a *dictionary attack*. There are 95,428,956,661,682,176 possible twelve-letter keys, but there are only about 1800 twelve-letter words in our dictionary file. If we use a twelve-letter word from the dictionary as a key, it would be easier to brute-force than a random three-letter key (which has 17,576 possible keys).

Of course, the cryptographer has an advantage in that the cryptanalyst doesn't know the length of the Vigenère key. But the cryptanalyst could try all one-letter keys, then all two-letter keys, and so on, which would still allow them to find a dictionary word key very quickly.

Source Code for the Vigenère Cipher Program

Open a new file editor window by selecting **File ▸ New File**. Enter the following code into the file editor, save it as *vigenereCipher.py*, and make sure *pyperclip.py* is in the same directory. Press F5 to run the program.

vigenereCipher.py

```
 1. # Vigenere Cipher (Polyalphabetic Substitution Cipher)
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 7.
 8. def main():
 9.     # This text can be downloaded from https://www.nostarch.com/
          crackingcodes/:
10.     myMessage = """Alan Mathison Turing was a British mathematician,
          logician, cryptanalyst, and computer scientist."""
11.     myKey = 'ASIMOV'
12.     myMode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
13.
14.     if myMode == 'encrypt':
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
24.
25.
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
32.
33.
34. def translateMessage(key, message, mode):
35.     translated = [] # Stores the encrypted/decrypted message string.
36.
37.     keyIndex = 0
38.     key = key.upper()
39.
```

```
    for symbol in message: # Loop through each symbol in message.
        num = LETTERS.find(symbol.upper())
        if num != -1: # -1 means symbol.upper() was not found in LETTERS.
            if mode == 'encrypt':
                num += LETTERS.find(key[keyIndex]) # Add if encrypting.
            elif mode == 'decrypt':
                num -= LETTERS.find(key[keyIndex]) # Subtract if
                  decrypting.

            num %= len(LETTERS) # Handle any wraparound.

            # Add the encrypted/decrypted symbol to the end of translated:
            if symbol.isupper():
                translated.append(LETTERS[num])
            elif symbol.islower():
                translated.append(LETTERS[num].lower())

            keyIndex += 1 # Move to the next letter in the key.
            if keyIndex == len(key):
                keyIndex = 0
        else:
            # Append the symbol without encrypting/decrypting:
            translated.append(symbol)

    return ''.join(translated)


# If vigenereCipher.py is run (instead of imported as a module), call
# the main() function:
if __name__ == '__main__':
    main()
```

Sample Run of the Vigenère Cipher Program

When you run the program, its output will look like this:

```
Encrypted message:
Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi, lgouqdaf, kdmktsvmztsl, izr
xoexghzr kkusitaaf.
The message has been copied to the clipboard.
```

The program prints the encrypted message and copies the encrypted text to the clipboard.

Setting Up Modules, Constants, and the main() Function

The beginning of the program has the usual comments describing the program, an `import` statement for the `pyperclip` module, and a variable called `LETTERS` that holds a string of every uppercase letter. The `main()` function for the Vigenère cipher is like the other `main()` functions in this book: it starts by defining the variables for `message`, `key`, and `mode`.

```
 1. # Vigenere Cipher (Polyalphabetic Substitution Cipher)
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 7.
 8. def main():
 9.     # This text can be downloaded from https://www.nostarch.com/
          crackingcodes/:
10.     myMessage = """Alan Mathison Turing was a British mathematician,
          logician, cryptanalyst, and computer scientist."""
11.     myKey = 'ASIMOV'
12.     myMode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
13.
14.     if myMode == 'encrypt':
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
```

The user sets these variables on lines 10, 11, and 12 before running the program. The encrypted or decrypted message (depending on what `myMode` is set to) is stored in a variable named `translated` so it can be printed to the screen (line 20) and copied to the clipboard (line 21).

Building Strings with the List-Append-Join Process

Almost all the programs in this book have built a string with code in some form. That is, the program creates a variable that starts as a blank string and then adds characters using string concatenation. This is what the previous cipher programs have done with the `translated` variable. Open the interactive shell and enter the following code:

```
>>> building = ''
>>> for c in 'Hello world!':
>>>     building += c
>>> print(building)
```

This code loops through each character in the string `'Hello world!'` and concatenates it to the end of the string stored in `building`. At the end of the loop, `building` holds the complete string.

Although string concatenation seems like a straightforward technique, it's very inefficient in Python. It's much faster to start with a blank list and then use the `append()` list method. When you're done building the list of

strings, you can convert the list to a single string value using the `join()` method. The following code does the same thing as the previous example, but faster. Enter the code into the interactive shell:

```
>>> building = []
>>> for c in 'Hello world!':
>>>     building.append(c)
>>> building = ''.join(building)
>>> print(building)
```

Using this approach to build up strings instead of modifying a string will result in much faster programs. You can see the difference by timing the two approaches using `time.time()`. Open a new file editor window and enter the following code:

stringTest.py

```
import time

startTime = time.time()
for trial in range(10000):
  building = ''
  for i in range(10000):
      building += 'x'
print('String concatenation: ', (time.time() - startTime))

startTime = time.time()
for trial in range(10000):
  building = []
  for i in range(10000):
      building.append('x')
  building = ''.join(building)
print('List appending:       ', (time.time() - startTime))
```

Save this program as *stringTest.py* and run it. The output will look something like this:

```
String concatenation:  40.317070960998535
List appending:        10.488219022750854
```

The program *stringTest.py* sets a variable `startTime` as the current time, runs code to append 10,000 characters to a string using concatenation, and then prints the time it took to finish the concatenation. Then the program resets `startTime` to the current time, runs code to use the list-appending method to build a string of the same length, and then prints the total time it took to finish. On my computer, using string concatenation to build 10,000 strings that are 10,000 characters each took about 40 seconds, but using the list-append-join process to do the same task took only 10 seconds. If your program builds a lot of strings, using lists can make your program much faster.

We'll use the list-append-join process to build strings for the remaining programs in this book.

Encrypting and Decrypting the Message

Because the encryption and decryption code is mostly the same, we'll create two wrapper functions called `encryptMessage()` and `decryptMessage()` for the function `translateMessage()`, which will hold the actual code to encrypt and decrypt.

```
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
```

The `translateMessage()` function builds the encrypted (or decrypted) string one character at a time. The list in `translated` stores these characters so they can be joined when the string building is done.

```
34. def translateMessage(key, message, mode):
35.     translated = [] # Stores the encrypted/decrypted message string.
36.
37.     keyIndex = 0
38.     key = key.upper()
```

Keep in mind that the Vigenère cipher is just the Caesar cipher except a different key is used depending on the position of the letter in the message. The `keyIndex` variable, which keeps track of which subkey to use, starts at `0` because the letter used to encrypt or decrypt the first character of the message is `key[0]`.

The program assumes that the key is in all uppercase letters. To make sure the key is valid, line 38 calls `upper()` on `key`.

The rest of the code in `translateMessage()` is similar to the Caesar cipher code:

```
40.     for symbol in message: # Loop through each symbol in message.
41.         num = LETTERS.find(symbol.upper())
42.         if num != -1: # -1 means symbol.upper() was not found in LETTERS.
43.             if mode == 'encrypt':
44.                 num += LETTERS.find(key[keyIndex]) # Add if encrypting.
45.             elif mode == 'decrypt':
46.                 num -= LETTERS.find(key[keyIndex]) # Subtract if
                      decrypting.
```

The `for` loop on line 40 sets the characters in `message` to the variable `symbol` on each iteration of the loop. Line 41 finds the index of the uppercase version of `symbol` in `LETTERS`, which is how we translate a letter into a number.

If `num` isn't set to `-1` on line 41, the uppercase version of `symbol` was found in `LETTERS` (meaning that `symbol` is a letter). The `keyIndex` variable keeps track of which subkey to use, and the subkey is always what `key[keyIndex]` evaluates to.

Of course, this is just a single letter string. We need to find this letter's index in LETTERS to convert the subkey to an integer. This integer is then added (if encrypting) to the symbol's number on line 44 or subtracted (if decrypting) to the symbol's number on line 46.

In the Caesar cipher code, we checked whether the new value of num was less than 0 (in which case, we added len(LETTERS) to it) or whether the new value of num was len(LETTERS) or greater (in which case, we subtracted len(LETTERS) from it). These checks handle the wraparound cases.

However, there is a simpler way to handle both of these cases. If we mod the integer stored in num by len(LETTERS), we can accomplish the same calculation in a single line of code:

```
48.             num %= len(LETTERS) # Handle any wraparound.
```

For example, if num was -8, we'd want to add 26 (that is, len(LETTERS)) to it to get 18, and that can be expressed as -8 % 26, which evaluates to 18. Or if num was 31, we'd want to subtract 26 to get 5, and 31 % 26 evaluates to 5. The modular arithmetic on line 48 handles both wraparound cases.

The encrypted (or decrypted) character exists at LETTERS[num]. However, we want the encrypted (or decrypted) character's case to match the original case of symbol.

```
50.             # Add the encrypted/decrypted symbol to the end of translated:
51.             if symbol.isupper():
52.                 translated.append(LETTERS[num])
53.             elif symbol.islower():
54.                 translated.append(LETTERS[num].lower())
```

So if symbol is an uppercase letter, the condition on line 51 is True, and line 52 appends the character at LETTERS[num] to translated because all the characters in LETTERS are already in uppercase.

However, if symbol is a lowercase letter, the condition on line 53 is True instead, and line 54 appends the lowercase form of LETTERS[num] to translated. This is how we make the encrypted (or decrypted) message match the original message casing.

Now that we've translated the symbol, we want to ensure that on the next iteration of the for loop, we use the next subkey. Line 56 increments keyIndex by 1, so the next iteration uses the index of the next subkey:

```
56.             keyIndex += 1 # Move to the next letter in the key.
57.             if keyIndex == len(key):
58.                 keyIndex = 0
```

However, if we were on the last subkey in the key, keyIndex would be equal to the length of key. Line 57 checks for this condition and resets keyIndex back to 0 on line 58 if that's the case so that key[keyIndex] points back to the first subkey.

The indentation indicates that the `else` statement on line 59 is paired with the `if` statement on line 42:

```
59.         else:
60.             # Append the symbol without encrypting/decrypting:
61.             translated.append(symbol)
```

The code on line 61 executes if the symbol was not found in the `LETTERS` string. This happens if `symbol` is a number or punctuation mark, such as `'5'` or `'?'`. In this case, line 61 appends the unmodified symbol to `translated`.

Now that we're done building the string in `translated`, we call the `join()` method on the blank string:

```
63.     return ''.join(translated)
```

This line makes the function return the whole encrypted or decrypted message when the function is called.

Calling the main() Function

Lines 68 and 69 finish the program's code:

```
68. if __name__ == '__main__':
69.     main()
```

These lines call the `main()` function if the program was run by itself rather than being imported by another program that wants to use its `encryptMessage()` and `decryptMessage()` functions.

Summary

You're close to the end of this book, but notice that the Vigenère cipher isn't that much more complicated than the Caesar cipher, which was one of the first cipher programs you learned. With just a few changes to the Caesar cipher, we created a cipher that has exponentially more possible keys than can be brute-forced.

The Vigenère cipher isn't vulnerable to the dictionary word pattern attack that the simple substitution hacker program uses. For hundreds of years, the "indecipherable" Vigenère cipher kept messages secret, but this cipher, too, eventually became vulnerable. In Chapters 19 and 20, you'll learn frequency analysis techniques that will enable you to hack the Vigenère cipher.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Which cipher is the Vigenère cipher similar to, except that the Vigenère cipher uses multiple keys instead of just one key?
2. How many possible keys are there for a Vigenère key with a key length of 10?
   a. Hundreds
   b. Thousands
   c. Millions
   d. More than a trillion
3. What kind of cipher is the Vigenère cipher?

LEARN PYTHON BY HACKING SECRET CIPHERS

COVERS PYTHON 3

Learn how to program in Python while making and breaking ciphers—algorithms used to create and send secret messages!

After a crash course in Python programming basics, you'll learn to make, test, and hack programs that encrypt text with classical ciphers like the transposition cipher and Vigenère cipher. You'll begin with simple programs for the reverse and Caesar ciphers and then work your way up to public key cryptography, the type of encryption used to secure today's online transactions, including digital signatures, email, and Bitcoin.

Each program includes the full code and a line-by-line explanation of how things work. By the end of the book, you'll have learned how to code in Python and you'll have the clever programs to prove it!

You'll also learn how to:

- Combine loops, variables, and flow control statements into real working programs
- Use dictionary files to instantly detect whether decrypted messages are valid English or gibberish
- Create test programs to make sure that your code encrypts and decrypts correctly
- Code (and hack!) a working example of the affine cipher, which uses modular arithmetic to encrypt a message
- Break ciphers with techniques such as brute-force and frequency analysis

There's no better way to learn to code than to play with real programs. *Cracking Codes with Python* makes the learning fun!

ABOUT THE AUTHOR

Al Sweigart is a professional software developer who teaches programming to kids and adults. He is the author of *Automate the Boring Stuff with Python*, *Invent Your Own Computer Games with Python*, and *Scratch Programming Playground*, also from No Starch Press. His programming tutorials can be found at *inventwithpython.com*.

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

$29.95 *($39.95 CDN)*

ISBN: 978-1-59327-822-9

9 781593 278229 52995

SHELVE IN: PROGRAMMING LANGUAGES/PYTHON