

Hacking Secret Ciphers with Python

By Al Sweigart

Copyright © 2013 by Al Sweigart

Some Rights Reserved. “Hacking Secret Ciphers with Python” is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.

You are free:



To Share — to copy, distribute, display, and perform the work



To Remix — to make derivative works

Under the following conditions:



Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work). (Visibly include the title and author's name in any excerpts of this work.)



Noncommercial — You may not use this work for commercial purposes.



Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

This summary is located here: <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> Your fair use and other rights are in no way affected by the above. There is a human-readable summary of the Legal Code (the full license), located here: <http://creativecommons.org/licenses/by-nc-sa/3.0/us/legalcode>

Book Version 3

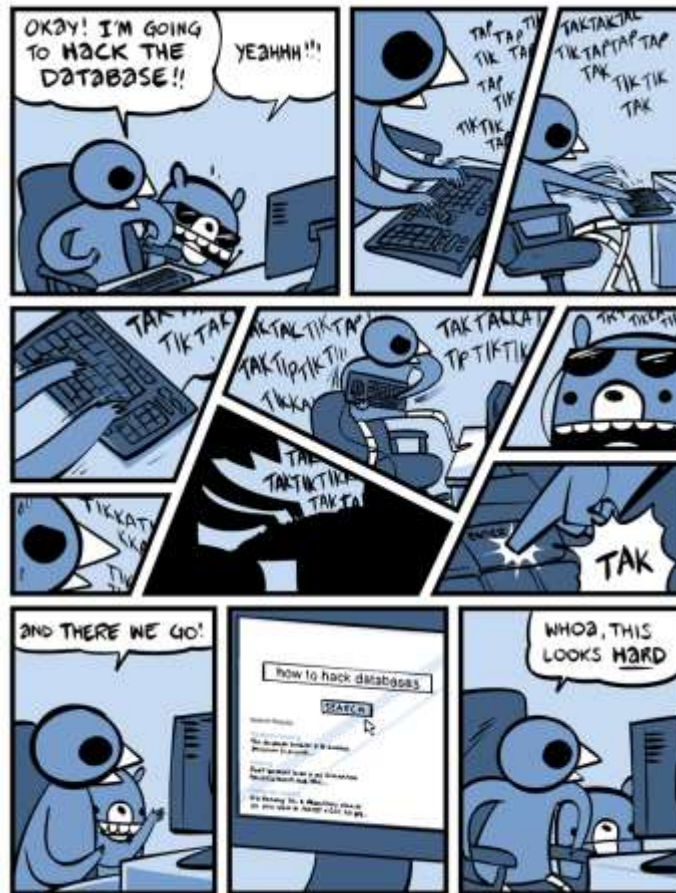
Special thanks to Ari Lacenski. I can't thank her enough. Without her efforts there'd be typos literally on every page.

Thanks to Jason Kibbe. Cover lock photo by “walknboston” <http://www.flickr.com/photos/walkn/3859852351/> Romeo & Juliet and other public domain texts from Project Gutenberg. Various image resources from Wikipedia. Wrinkled paper texture by Pink Sherbet Photography <http://www.flickr.com/photos/pinksherbet/2978651767/> Computer User icon by Katzenbaer.

If you've downloaded this book from a torrent, it's probably out of date. Go to <http://inventwithpython.com/hacking> to download the latest version.

ISBN 978-1482614374

1st Edition



Nedroid Picture Diary by Anthony Clark, <http://nedroid.com>

Movies and TV shows always make hacking look exciting with furious typing and meaningless ones and zeros flying across the screen. They make hacking look like something that you have to be super smart to learn. They make hacking look like magic.

It's not magic. It's based on computers, and everything computers do have logical principles behind them which can be learned and understood. Even when you don't understand or when the computer does something frustrating or mysterious, there is always, always, always a reason why.

And it's not hard to learn. This book assumes you know nothing about cryptography or programming, and helps you learn, step by step, how to write programs that can hack encrypted messages. Good luck and have fun!

100% of the profits from this book are donated
to the Electronic Frontier Foundation, the Creative Commons, and the Tor Project.

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.”

- C.M.

ABOUT THIS BOOK

There are many books that teach beginners how to write secret messages using ciphers. There are a couple books that teach beginners how to hack ciphers. As far as I can tell, there are no books to teach beginners how to write programs to hack ciphers. This book fills that gap.

This book is for complete beginners who do not know anything about encryption, hacking, or cryptography. The ciphers in this book (except for the RSA cipher in the last chapter) are all centuries old, and modern computers now have the computational power to hack their encrypted messages. No modern organization or individuals use these ciphers anymore. As such, there's no reasonable context in which you could get into legal trouble for the information in this book.

This book is for complete beginners who have never programmed before. This book teaches basic programming concepts with the Python programming language. Python is the best language for beginners to learn programming: it is simple and readable yet also a powerful programming language used by professional software developers. The Python software can be downloaded for free from <http://python.org> and runs on Linux, Windows, OS X, and the Raspberry Pi.

There are two definitions of “hacker”. A hacker is 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 are not limited by the original rules of that system and can creatively modify it to work in new ways. “Hacker” is also used to mean criminals who break into computer systems, violate people's privacy, and cause damage. This book uses “hacker” in the first sense. **Hackers are cool. Criminals are just people who think they're being clever by breaking stuff.** Personally, my day job as a software developer pays me way more for less work than writing a virus or doing an Internet scam would.

On a side note, don't use any of the encryption programs in this book for your actual files. They're fun to play with but they don't provide true security. And in general, you shouldn't trust the ciphers that you yourself make. As legendary cryptographer Bruce Schneier put it, “Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis. And the only way to prove that is to subject the algorithm to years of analysis by the best cryptographers around.”

This book is released under a Creative Commons license and is free to copy and distribute (as long as you don't charge money for it). The book can be downloaded for free from its website at <http://inventwithpython.com/hacking>. If you ever have questions about how these programs work, feel free to email me at al@inventwithpython.com.

TABLE OF CONTENTS

About This Book	6
Table of Contents	7
Chapter 1 - Making Paper Cryptography Tools.....	1
What is Cryptography?	2
Codes vs. Ciphers	2
Making a Paper Cipher Wheel	3
A Virtual Cipher Wheel	7
How to Encrypt with the Cipher Wheel	8
How to Decrypt with the Cipher Wheel.....	9
A Different Cipher Tool: The St. Cyr Slide	10
Practice Exercises, Chapter 1, Set A	11
Doing Cryptography without Paper Tools	11
Practice Exercises, Chapter 1, Set B	13
Double-Strength Encryption?.....	13
Programming a Computer to do Encryption	14
Chapter 2 - Installing Python	16
Downloading and Installing Python	17
Downloading pyperclip.py.....	18
Starting IDLE.....	18
The Featured Programs	19
Line Numbers and Spaces.....	20
Text Wrapping in This Book	20
Tracing the Program Online.....	21
Checking Your Typed Code with the Online Diff Tool.....	21
Copying and Pasting Text.....	21
More Info Links	22
Programming and Cryptography.....	22
Chapter 3 - The Interactive Shell.....	26
Some Simple Math Stuff.....	26
Integers and Floating Point Values	27

Expressions	27
Order of Operations	28
Evaluating Expressions	29
Errors are Okay!.....	29
Practice Exercises, Chapter 3, Set A	30
Every Value has a Data Type.....	30
Storing Values in Variables with Assignment Statements	30
Overwriting Variables.....	32
Using More Than One Variable.....	33
Variable Names.....	34
Practice Exercises, Chapter 3, Set B	35
Summary - But When Are We Going to Start Hacking?.....	35
Chapter 4 - Strings and Writing Programs.....	36
Strings	36
String Concatenation with the + Operator	38
String Replication with the * Operator	39
Printing Values with the print () Function	39
Escape Characters	40
Quotes and Double Quotes	41
Practice Exercises, Chapter 4, Set A	42
Indexing	42
Negative Indexes.....	44
Slicing.....	44
Blank Slice Indexes.....	45
Practice Exercises, Chapter 4, Set B	46
Writing Programs in IDLE’s File Editor.....	46
Hello World!.....	47
Source Code of Hello World.....	47
Saving Your Program	48
Running Your Program.....	49
Opening The Programs You’ve Saved.....	50
How the “Hello World” Program Works	50
Comments	50
Functions.....	51

The <code>print()</code> function	51
The <code>input()</code> function	51
Ending the Program	52
Practice Exercises, Chapter 4, Set C	52
Summary.....	53
Chapter 5 - The Reverse Cipher	54
The Reverse Cipher.....	54
Source Code of the Reverse Cipher Program.....	55
Sample Run of the Reverse Cipher Program.....	55
Checking Your Source Code with the Online Diff Tool	56
How the Program Works.....	56
The <code>len()</code> Function	57
Introducing the <code>while</code> Loop.....	58
The Boolean Data Type	59
Comparison Operators	59
Conditions.....	62
Blocks	62
The <code>while</code> Loop Statement	63
“Growing” a String	64
Tracing Through the Program, Step by Step	67
Using <code>input()</code> In Our Programs.....	68
Practice Exercises, Chapter 5, Section A	69
Summary.....	69
Chapter 6 - The Caesar Cipher	70
Implementing a Program.....	70
Source Code of the Caesar Cipher Program.....	71
Sample Run of the Caesar Cipher Program.....	72
Checking Your Source Code with the Online Diff Tool	73
Practice Exercises, Chapter 6, Set A	73
How the Program Works.....	73
Importing Modules with the <code>import</code> Statement.....	73
Constants.....	74
The <code>upper()</code> and <code>lower()</code> String Methods	75

The <code>for</code> Loop Statement.....	76
A <code>while</code> Loop Equivalent of a <code>for</code> Loop.....	77
Practice Exercises, Chapter 6, Set B	78
The <code>if</code> Statement	78
The <code>else</code> Statement.....	79
The <code>elif</code> Statement.....	79
The <code>in</code> and <code>not in</code> Operators.....	80
The <code>find()</code> String Method.....	81
Practice Exercises, Chapter 6, Set C	82
Back to the Code.....	82
Displaying and Copying the Encrypted/Decrypted String	85
Encrypt Non-Letter Characters	86
Summary.....	87
Chapter 7 - Hacking the Caesar Cipher with the Brute-Force Technique.....	88
Hacking Ciphers	88
The Brute-Force Attack	89
Source Code of the Caesar Cipher Hacker Program	89
Sample Run of the Caesar Cipher Hacker Program	90
How the Program Works.....	91
The <code>range()</code> Function	91
Back to the Code.....	93
String Formatting.....	94
Practice Exercises, Chapter 7, Set A	95
Summary.....	95
Chapter 8 - Encrypting with the Transposition Cipher	96
Encrypting with the Transposition Cipher	96
Practice Exercises, Chapter 8, Set A	98
A Transposition Cipher Encryption Program.....	98
Source Code of the Transposition Cipher Encryption Program	98
Sample Run of the Transposition Cipher Encryption Program	99
How the Program Works.....	100
Creating Your Own Functions with <code>def</code> Statements.....	100
The Program's <code>main()</code> Function	101

Parameters.....	102
Variables in the Global and Local Scope	104
The <code>global</code> Statement.....	104
Practice Exercises, Chapter 8, Set B	106
The List Data Type	106
Using the <code>list()</code> Function to Convert Range Objects to Lists.....	109
Reassigning the Items in Lists.....	110
Reassigning Characters in Strings.....	110
Lists of Lists	110
Practice Exercises, Chapter 8, Set C	111
Using <code>len()</code> and the <code>in</code> Operator with Lists.....	111
List Concatenation and Replication with the <code>+</code> and <code>*</code> Operators.....	112
Practice Exercises, Chapter 8, Set D.....	113
The Transposition Encryption Algorithm	113
Augmented Assignment Operators	115
Back to the Code.....	116
The <code>join()</code> String Method.....	118
Return Values and <code>return</code> Statements	119
Practice Exercises, Chapter 8, Set E	120
Back to the Code.....	120
The Special <code>__name__</code> Variable.....	120
Key Size and Message Length	121
Summary.....	122
Chapter 9 - Decrypting with the Transposition Cipher	123
Decrypting with the Transposition Cipher on Paper	124
Practice Exercises, Chapter 9, Set A	125
A Transposition Cipher Decryption Program.....	126
Source Code of the Transposition Cipher Decryption Program	126
How the Program Works.....	127
The <code>math.ceil()</code> , <code>math.floor()</code> and <code>round()</code> Functions.....	128
The <code>and</code> and <code>or</code> Boolean Operators.....	132
Practice Exercises, Chapter 9, Set B	133
Truth Tables.....	133

The <code>and</code> and <code>or</code> Operators are Shortcuts	134
Order of Operations for Boolean Operators	135
Back to the Code	135
Practice Exercises, Chapter 9, Set C	137
Summary	137
Chapter 10 - Programming a Program to Test Our Program	138
Source Code of the Transposition Cipher Tester Program	139
Sample Run of the Transposition Cipher Tester Program	140
How the Program Works	141
Pseudorandom Numbers and the <code>random.seed()</code> Function	141
The <code>random.randint()</code> Function	143
References	143
The <code>copy.deepcopy()</code> Functions	147
Practice Exercises, Chapter 10, Set A	148
The <code>random.shuffle()</code> Function	148
Randomly Scrambling a String	149
Back to the Code	149
The <code>sys.exit()</code> Function	150
Testing Our Test Program	151
Summary	152
Chapter 11 - Encrypting and Decrypting Files	153
Plain Text Files	154
Source Code of the Transposition File Cipher Program	154
Sample Run of the Transposition File Cipher Program	157
Reading From Files	157
Writing To Files	158
How the Program Works	159
The <code>os.path.exists()</code> Function	160
The <code>startswith()</code> and <code>endswith()</code> String Methods	161
The <code>title()</code> String Method	162
The <code>time</code> Module and <code>time.time()</code> Function	163
Back to the Code	164
Practice Exercises, Chapter 11, Set A	165

Summary.....	165
Chapter 12 - Detecting English Programmatically.....	166
How Can a Computer Understand English?.....	167
Practice Exercises, Chapter 12, Section A	169
The Detect English Module	169
Source Code for the Detect English Module.....	169
How the Program Works.....	170
Dictionaries and the Dictionary Data Type	171
Adding or Changing Items in a Dictionary	172
Practice Exercises, Chapter 12, Set B	173
Using the len() Function with Dictionaries.....	173
Using the in Operator with Dictionaries.....	173
Using for Loops with Dictionaries	174
Practice Exercises, Chapter 12, Set C	174
The Difference Between Dictionaries and Lists.....	174
Finding Items is Faster with Dictionaries Than Lists.....	175
The split() Method.....	175
The None Value	176
Back to the Code.....	177
“Divide by Zero” Errors.....	179
The float(), int(), and str() Functions and Integer Division	179
Practice Exercises, Chapter 12, Set D	180
Back to the Code.....	180
The append() List Method.....	182
Default Arguments.....	183
Calculating Percentage.....	184
Practice Exercises, Chapter 12, Set E	185
Summary.....	186
Chapter 13 - Hacking the Transposition Cipher	187
Source Code of the Transposition Cipher Hacker Program	187
Sample Run of the Transposition Breaker Program	189
How the Program Works.....	190
Multi-line Strings with Triple Quotes	190

Back to the Code.....	191
The <code>strip()</code> String Method.....	193
Practice Exercises, Chapter 13, Set A.....	195
Summary.....	195
Chapter 14 - Modular Arithmetic with the Multiplicative and Affine Ciphers	196
Oh No Math!.....	197
Math Oh Yeah!	197
Modular Arithmetic (aka Clock Arithmetic).....	197
The % Mod Operator.....	199
Practice Exercises, Chapter 14, Set A.....	199
GCD: Greatest Common Divisor (aka Greatest Common Factor).....	199
Visualize Factors and GCD with Cuisenaire Rods.....	200
Practice Exercises, Chapter 14, Set B.....	202
Multiple Assignment.....	202
Swapping Values with the Multiple Assignment Trick.....	203
Euclid's Algorithm for Finding the GCD of Two Numbers.....	203
"Relatively Prime".....	205
Practice Exercises, Chapter 14, Set C.....	205
The Multiplicative Cipher.....	205
Practice Exercises, Chapter 14, Set D.....	207
Multiplicative Cipher + Caesar Cipher = The Affine Cipher.....	207
The First Affine Key Problem.....	207
Decrypting with the Affine Cipher.....	208
Finding Modular Inverses.....	209
The // Integer Division Operator.....	210
Source Code of the <code>cryptomath</code> Module.....	210
Practice Exercises, Chapter 14, Set E.....	211
Summary.....	211
Chapter 15 - The Affine Cipher	213
Source Code of the Affine Cipher Program.....	214
Sample Run of the Affine Cipher Program.....	216
Practice Exercises, Chapter 15, Set A.....	216
How the Program Works.....	216
Splitting One Key into Two Keys.....	218

The Tuple Data Type	218
Input Validation on the Keys	219
The Affine Cipher Encryption Function	220
The Affine Cipher Decryption Function	221
Generating Random Keys	222
The Second Affine Key Problem: How Many Keys Can the Affine Cipher Have?	223
Summary	225
Chapter 16 - Hacking the Affine Cipher	226
Source Code of the Affine Cipher Hacker Program	226
Sample Run of the Affine Cipher Hacker Program	228
How the Program Works	228
The Affine Cipher Hacking Function	230
The ** Exponent Operator	230
The continue Statement	231
Practice Exercises, Chapter 16, Set A	234
Summary	234
Chapter 17 - The Simple Substitution Cipher	235
The Simple Substitution Cipher with Paper and Pencil	236
Practice Exercises, Chapter 17, Set A	236
Source Code of the Simple Substitution Cipher	237
Sample Run of the Simple Substitution Cipher Program	239
How the Program Works	239
The Program's main() Function	240
The sort() List Method	241
Wrapper Functions	242
The Program's translateMessage() Function	243
The isupper() and islower() String Methods	245
Practice Exercises, Chapter 17, Set B	247
Generating a Random Key	247
Encrypting Spaces and Punctuation	248
Practice Exercises, Chapter 17, Set C	249
Summary	249
Chapter 18 - Hacking the Simple Substitution Cipher	250

Computing Word Patterns.....	251
Getting a List of Candidates for a Cipherword	252
Practice Exercises, Chapter 18, Set A	253
Source Code of the Word Pattern Module	253
Sample Run of the Word Pattern Module	255
How the Program Works.....	256
The <code>pprint.pprint()</code> and <code>pprint.pformat()</code> Functions	256
Building Strings in Python with Lists	257
Calculating the Word Pattern	258
The Word Pattern Program's <code>main()</code> Function	259
Hacking the Simple Substitution Cipher	262
Source Code of the Simple Substitution Hacking Program.....	262
Hacking the Simple Substitution Cipher (in Theory).....	266
Explore the Hacking Functions with the Interactive Shell	266
How the Program Works.....	271
Import All the Things.....	272
A Brief Intro to Regular Expressions and the <code>sub()</code> Regex Method	272
The Hacking Program's <code>main()</code> Function.....	273
Partially Hacking the Cipher.....	274
Blank Cipherletter Mappings	275
Adding Letters to a Cipherletter Mapping	276
Intersecting Two Letter Mappings	277
Removing Solved Letters from the Letter Mapping.....	278
Hacking the Simple Substitution Cipher	281
Creating a Key from a Letter Mapping	283
Couldn't We Just Encrypt the Spaces Too?	285
Summary.....	286
Chapter 19 - The Vigenère Cipher	287
Le Chiffre Indéchiffrable	288
Multiple "Keys" in the Vigenère Key	288
Source Code of Vigenère Cipher Program.....	291
Sample Run of the Vigenère Cipher Program.....	294
How the Program Works.....	294
Summary.....	298

Chapter 20 - Frequency Analysis.....	299
The Code for Matching Letter Frequencies	304
How the Program Works.....	306
The Most Common Letters, “ETAOIN”	307
The Program’s getLettersCount () Function	307
The Program’s getItemAtIndexZero () Function	308
The Program’s getFrequencyOrder () Function	308
The sort () Method’s key and reverse Keyword Arguments	310
Passing Functions as Values	311
Converting Dictionaries to Lists with the keys(), values(), items() Dictionary Methods	313
Sorting the Items from a Dictionary.....	315
The Program’s englishFreqMatchScore () Function.....	316
Summary.....	317
Chapter 21 - Hacking the Vigenère Cipher	318
The Dictionary Attack.....	319
Source Code for a Vigenère Dictionary Attack Program	319
Sample Run of the Vigenère Dictionary Hacker Program	320
The readlines () File Object Method.....	321
The Babbage Attack & Kasiski Examination.....	321
Kasiski Examination, Step 1 – Find Repeat Sequences’ Spacings.....	321
Kasiski Examination, Step 2 – Get Factors of Spacings	322
Get Every Nth Letters from a String	323
Frequency Analysis.....	323
Brute-Force through the Possible Keys.....	325
Source Code for the Vigenère Hacking Program	326
Sample Run of the Vigenère Hacking Program	332
How the Program Works.....	334
Finding Repeated Sequences	335
Calculating Factors	337
Removing Duplicates with the set() Function.....	338
The Kasiski Examination Algorithm.....	341
The extend () List Method.....	342
The end Keyword Argument for print ()	347

The <code>itertools.product()</code> Function.....	348
The <code>break</code> Statement.....	352
Practice Exercises, Chapter 21, Set A.....	354
Modifying the Constants of the Hacking Program.....	354
Summary.....	355
Chapter 22 - The One-Time Pad Cipher.....	356
The Unbreakable One-Time Pad Cipher.....	357
Why the One-Time Pad is Unbreakable.....	357
Beware Pseudorandomness.....	358
Beware the Two-Time Pad.....	358
The Two-Time Pad is the Vigenère Cipher.....	359
Practice Exercises, Chapter 22, Set A.....	360
Summary.....	360
Chapter 23 - Finding Prime Numbers.....	361
Prime Numbers.....	362
Composite Numbers.....	363
Source Code for The Prime Sieve Module.....	363
How the Program Works.....	364
How to Calculate if a Number is Prime.....	365
The Sieve of Eratosthenes.....	366
The <code>primeSieve()</code> Function.....	368
Detecting Prime Numbers.....	369
Source Code for the Rabin-Miller Module.....	370
Sample Run of the Rabin Miller Module.....	372
How the Program Works.....	372
The Rabin-Miller Primality Algorithm.....	372
The New and Improved <code>isPrime()</code> Function.....	373
Summary.....	375
Chapter 24 - Public Key Cryptography and the RSA Cipher.....	378
Public Key Cryptography.....	379
The Dangers of “Textbook” RSA.....	381
A Note About Authentication.....	381
The Man-In-The-Middle Attack.....	382

Generating Public and Private Keys.....	383
Source Code for the RSA Key Generation Program	383
Sample Run of the RSA Key Generation Program	385
How the Key Generation Program Works	386
The Program's <code>generateKey()</code> Function.....	387
RSA Key File Format	390
Hybrid Cryptosystems	391
Source Code for the RSA Cipher Program	391
Sample Run of the RSA Cipher Program.....	395
Practice Exercises, Chapter 24, Set A	397
Digital Signatures	397
How the RSA Cipher Program Works	398
ASCII: Using Numbers to Represent Characters	400
The <code>chr()</code> and <code>ord()</code> Functions	400
Practice Exercises, Chapter 24, Set B	401
Blocks	401
Converting Strings to Blocks with <code>getBlocksFromText()</code>	404
The <code>encode()</code> String Method and the Bytes Data Type	405
The <code>bytes()</code> Function and <code>decode()</code> Bytes Method	405
Practice Exercises, Chapter 24, Set C	406
Back to the Code.....	406
The <code>min()</code> and <code>max()</code> Functions	407
The <code>insert()</code> List Method.....	410
The Mathematics of RSA Encrypting and Decrypting.....	411
The <code>pow()</code> Function	411
Reading in the Public & Private Keys from their Key Files.....	413
The Full RSA Encryption Process	413
The Full RSA Decryption Process	416
Practice Exercises, Chapter 24, Set D	418
Why Can't We Hack the RSA Cipher.....	418
Summary.....	420
About the Author	422



MAKING PAPER CRYPTOGRAPHY TOOLS

Topics Covered In This Chapter:

- What is cryptography?
- Codes and ciphers
- The Caesar cipher
- Cipher wheels
- St. Cyr slides
- Doing cryptography with paper and pencil
- “Double strength” encryption

“I couldn’t help but overhear, probably because I was eavesdropping.”

Anonymous

What is Cryptography?

Look at the following two pieces of text:

“Zsijwxyfsi niqjsjxx gjyyjw. Ny nx jnymjw ktqqd tw
bnxitr; ny nx anwyzi ns bjfqym fsi anhj ns utajwyd.
Ns ymj bnsyjw tk tzw qnkj, bj hfs jsotd ns ujfhj ymj
kwznyx bmnhm ns nyx xuwnsl tzw nsizywd uqfsyji.
Htzwynjwx tk lqtwd, bwnyjwx tw bfwntwx, xqzrgjw
nx ujwmyyji dtz, gzy tsqd zuts qfzwjxq.”

“Flwyt tsytbbnz jqtw yjxndwri iyn fqq knqrqt xj mh
ndyn jxwqswbj. Dyi jkxxx sg ttwt gdhz js jwsn;
wnjyiyb aijnn snagdq nnpjwww, xstsxsu jdnxzz xkw
znfs uwwh xni xjzw jzwyjy jwnmns mnyfjx. Stjj wwzj
ti fnu, qt uyko qqsbay jmwsjk. Sxitwru nwnqn
nxfzfbf yy hnwydsj mhnxytb myysyt.”

The text on the left side is a secret message. The message has been **encrypted**, or turned into a secret code. It will be completely unreadable to anyone who doesn't know how to **decrypt** it (that is, turn it back into the plain English message.) This book will teach you how to encrypt and decrypt messages.

The message on the right is just random gibberish with no hidden meaning whatsoever. Encrypting your written messages is one way to keep them secret from other people, even if they get their hands on the encrypted message itself. *It will look exactly like random nonsense.*

Cryptography is the science of using secret codes. A **cryptographer** is someone who uses and studies secret codes. This book will teach you what you need to know to become a cryptographer.

Of course, these secret messages don't always stay secret. A **cryptanalyst** is someone who can hack secret codes and read other people's encrypted messages. Cryptanalysts are also called **code breakers** or **hackers**. This book will also teach you what you need to know to become a cryptanalyst. Unfortunately the type of hacking you learn in this book isn't dangerous enough to get you in trouble with the law. (I mean, fortunately.)

Spies, soldiers, hackers, pirates, royalty, merchants, tyrants, political activists, Internet shoppers, and anyone who has ever needed to share secrets with trusted friends have relied on cryptography to make sure their secrets stay secret.

Codes vs. Ciphers

The development of the electric telegraph in the early 19th century allowed for near-instant communication through wires across continents. This was much faster than 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 electric pulses. A short pulse is called a “dot” and a long pulse is called a “dash”.



Figure 1-1. Samuel Morse
April 27, 1791 – April 2, 1872

In order to convert these dots and dashes to English letters of the alphabet, an encoding system (or **code**) is needed to translate from English to electric pulse code (called **encoding**) and at the other end translate electric pulses to English (called **decoding**). The code to do this over telegraphs (and later, radio) was called Morse Code, and was developed by Samuel Morse and Alfred Vail. By tapping out 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! (If you'd like to learn Morse code, visit <http://invpy.com/morse>.)



Figure 1-2. Alfred Vail
September 25, 1807 – January 18, 1859

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

Figure 1-3. International Morse Code, with characters represented as dots and dashes.

Codes are made to be understandable and publicly available. Anyone should be able to look up what a code's symbols mean to decode an encoded message.

Making a Paper Cipher Wheel

Before we learn how to program computers to do encryption and decryption for us, let's learn how to do it ourselves with simple paper tools. It is easy to turn the understandable English text (which is called the **plaintext**) into the gibberish text that hides a secret code (called the

ciphertext). A **cipher** is a set of rules for converting between plaintext and ciphertext. These rules often use a secret key. We will learn several different ciphers in this book.

Let's learn a cipher called the Caesar cipher. This cipher was used by Julius Caesar two thousand years ago. The good news is that it is simple and easy to learn. The bad news is that because it is so simple, it is also easy for a cryptanalyst to break it. But we can use it as a simple learning exercise. More information about the Caesar cipher is given on Wikipedia:

http://en.wikipedia.org/wiki/Caesar_cipher.

To convert plaintext to ciphertext using the Caesar cipher, we will create something called a **cipher wheel** (also called a **cipher disk**). You can either photocopy the cipher wheel that appears in this book, or print out the one from <http://invpy.com/cipherwheel>. Cut out the two circles and lay them on top of each other like in Figure 1-8.

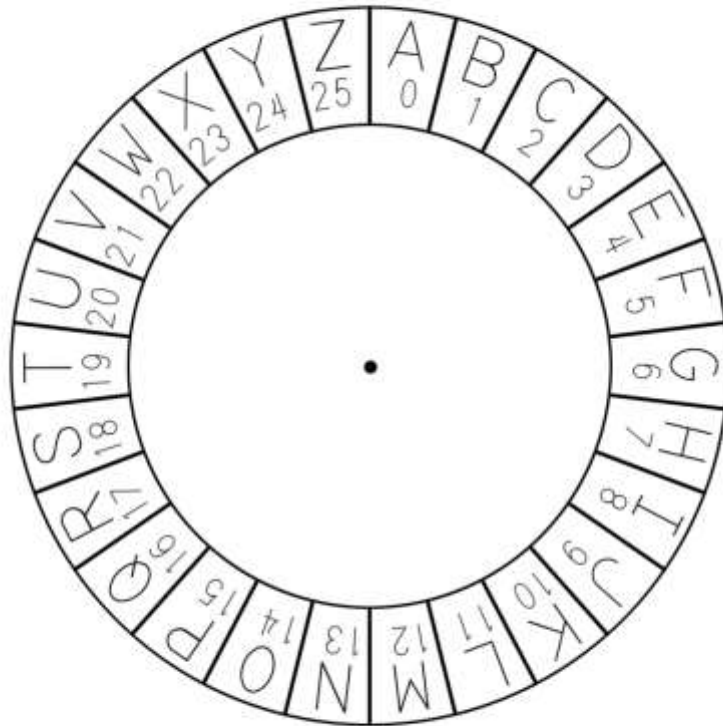


Figure 1-4. The inner circle of the cipher wheel cutout.

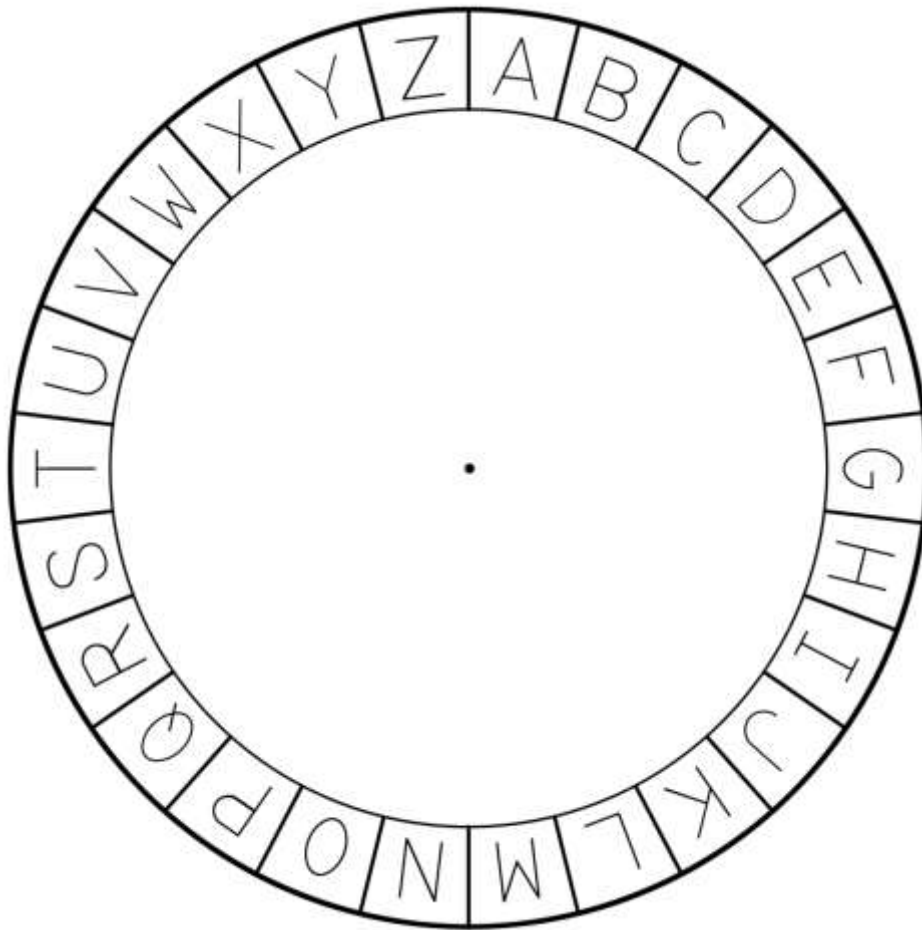


Figure 1-5. The outer circle of the cipher wheel cutout.

Don't cut out the page from this book!

Just make a photocopy of this page or print it from <http://invpy.com/cipherwheel>.

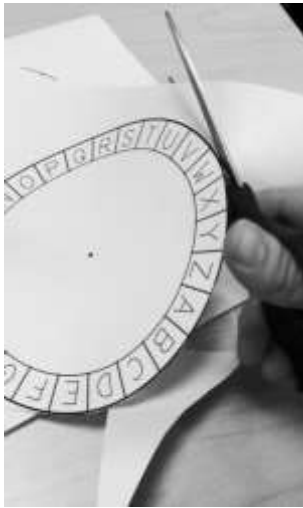


Figure 1-6. Cutting out the cipher wheel circles.

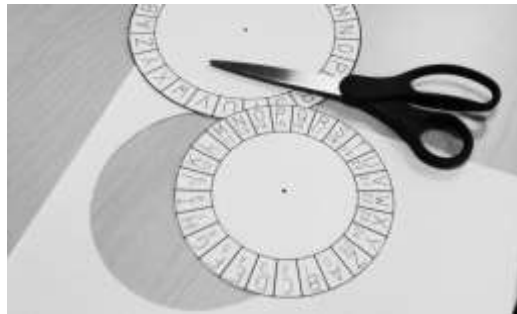


Figure 1-7. The cut-out circles.

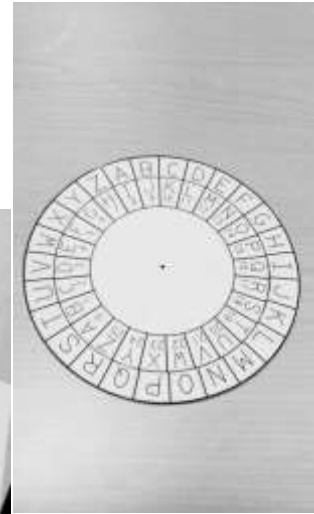


Figure 1-8. The completed cipher wheel.

After you cut out the circles, place the smaller one in the middle of the larger one. Put a pin or brad through the center of both circles so you can spin them around in place. You now have a tool for creating secret messages with the Caesar cipher.

A Virtual Cipher Wheel

There is also a virtual cipher wheel online if you don't have scissors and a photocopier handy. Open a web browser to <http://invpy.com/cipherwheel> to use the software version of the cipher wheel.

To spin the wheel around, click on it with the mouse and then move the mouse cursor around until the key you want is in place. Then click the mouse again to stop the wheel from spinning.

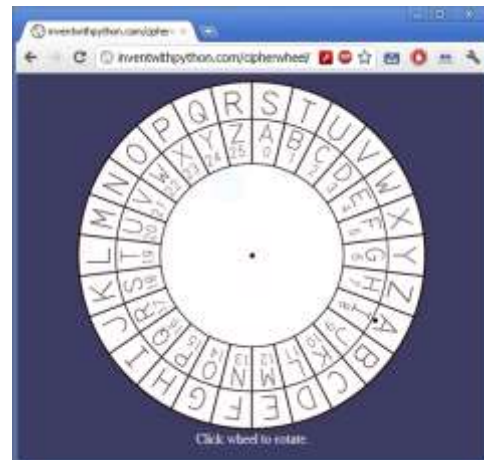


Figure 1-9. The online cipher wheel.

How to Encrypt with the Cipher Wheel

First, write out your message in English on paper. For this example we will encrypt the message, “The secret password is Rosebud.” Next, spin the inner wheel around until its letters match up with letters in the outer wheel. Notice in the outer wheel there is a dot next to the letter A. Look at the number in the inner wheel next to the dot in the outer wheel. This number is known the **encryption key**.

The encryption key is the secret to encrypting or decrypting the message. Anyone who reads this book can learn about the Caesar cipher, just like anyone who reads a book about locks can learn how a door lock works. But like a regular lock and key, unless they have the encryption key, they will not be able to unlock (that is, decrypt) the secret encrypted message. In Figure 1-9, the outer circle’s A is over the inner circle’s number 8. That means we will be using the key 8 to encrypt our message. The Caesar cipher uses the keys from 0 to 25. Let’s use the key 8 for our example. Keep the encryption key a secret; the ciphertext can be read by anyone who knows that the message was encrypted with key 8.

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

For each letter in our message, we will find where it is in the outer circle and replace it with the lined-up letter in the inner circle. The first letter in our message is T (the first “T” in “The secret...”), so we find the letter T in the outer circle, and then find the lined-up letter in the inner circle. This letter is B, so in our secret message we will always replace T’s with B’s. (If we were using some other encryption key besides 8, then the T’s in our plaintext would be replaced with a different letter.)

The next letter in our message is H, which turns into P. The letter E turns into M. When we have encrypted the entire message, the message has transformed from “The secret password is Rosebud.” to “Bpm amkzmb xiaaewzl qa Zwamjcl.” Now you can send this message to someone (or keep it written down for yourself) and nobody will be able to read it unless you tell them the secret encryption key (the number 8).

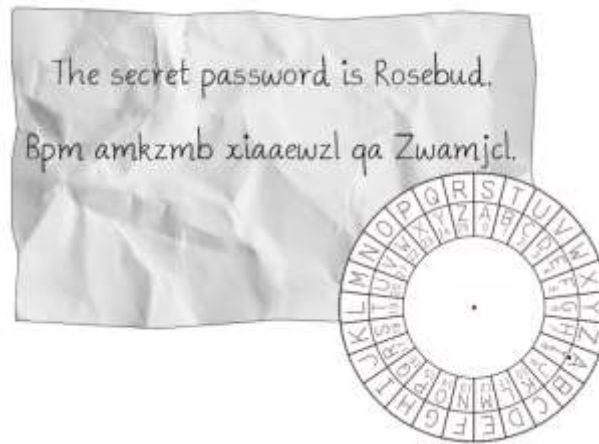


Figure 1-10. A message encrypted with the cipher wheel.

Each letter on the outer wheel will always be encrypted 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. This way you only need to look up a letter once.

How to Decrypt with the Cipher Wheel

To decrypt a ciphertext, go from the inner circle to the outer circle. Let’s say you receive this ciphertext from a friend, “Iwt ctl ephhldgs xh Hldgsuxhw.” You and everyone else won’t be able to decrypt it unless you know the key (or unless you are a clever hacker). But your friend has decided to use the key 15 for each message she sends you.

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). The first letter in the secret message is I, so we find I on the inner circle and look at the letter next to it on the outer circle, which is T. The W in the ciphertext will decrypt to the letter H. One by one, we can decrypt each letter in the ciphertext back to the plaintext, “The new password is Swordfish.”

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

If we use an incorrect key like 16 instead of the correct key 15, the decrypted message is “Sgd mdv ozrrvnqc hr Rvnqcehrg.” This plaintext doesn’t look plain at all. Unless the correct key is used, the decrypted message will never be understandable English.

A Different Cipher Tool: The St. Cyr Slide



Figure 1-11. Photocopy these strips to make a St. Cyr Slide.

There's another paper tool that can be used to do encryption and decryption, called the St. Cyr slide. It's like the cipher wheel except in a straight line.

Photocopy the image of the St. Cyr slide on the following page (or print it out from <http://invpy.com/stcyrslide>) and cut out the three strips.

Tape the two alphabet strips together, with the black box A next to the white box Z on the other strip. Cut out the slits on either side of the main slide box so that the taped-together strip can feed through it. It should look like this:

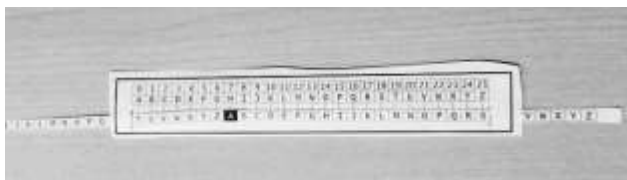


Figure 1-12. The completed St. Cyr Slide

When the black box A is underneath the letter H (and the number 7), then to encrypt you must find where the plaintext letter is on the long strip, and replace it with the letter above it. To decrypt, find the ciphertext letter on the top row of letters and replace it with the letter on the long strip below it.

The two slits on the larger box will hide any extra letters so that you only see one of each letter on the slide for any key.

The benefit of the St. Cyr slide is that it might be easier to find the letters you are looking for, since they are all in a straight line and will never be upside down like they sometimes are on the cipher wheel.

A virtual and printable St. Cyr slide can be found at <http://invpy.com/stcyrslide>.

Practice Exercises, Chapter 1, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice1A>.

Don't ignore the practice exercises!

There isn't enough room in this book to put in all the practice exercises, but they're still important.

You don't become a hacker by just reading about hacking and programming. You have to actually do it!

Doing Cryptography without Paper Tools

The cipher wheel and St. Cyr slide are nice tools to do encryption and decryption with the Caesar cipher. But we can implement the Caesar cipher with just pencil and paper.

Write out the letters of the alphabet from A to Z with the numbers from 0 to 25 under each letter. 0 goes underneath the A, 1 goes under the B, and so on until 25 is under Z. (There are 26 letters in the alphabet, but our numbers only go up to 25 because we started at 0, not 1.) It will end up looking something like this:

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

With the above letters-to-numbers code, we can use numbers to represent letters. **This is a very powerful concept, because math uses numbers. Now we have a way to do math on letters.**

Now to encrypt we find the number under the letter we wish to encrypt and add the key number to it. This sum will be the number under the encrypted letter. For example, we encrypt, “Hello. How are you?” with the key 13. First we find the number under the H, which is 7. Then we add the key to this number. $7 + 13 = 20$. The number 20 is under the letter U, which means the letter H encrypts to the letter U. To encrypt the letter E, we add the 4 under E to 13 to get 17. The number above 17 is R, so E gets encrypted to R. And so on.

This works fine until we get to the letter O. The number under O is 14. But when we add $14 + 13$ we get 27. But our list of numbers only goes up to 25. If the sum of the letter's number and the

key is 26 or more, we should subtract 26 from it. So $27 - 26$ is 1. The letter above the number 1 is B. So the letter O encrypts to the letter B when we are using the key 13. One by one, we can then encrypt the letters in, “Hello. How are you?” to “Uryyb. Ubj ner lbh?”

So the steps to encrypt a letter are:

1. Decide on a key from 1 to 25. Keep this key secret!
2. Find the plaintext letter’s number.
3. Add the key to the plaintext letter’s number.
4. If this number is larger than 26, subtract 26.
5. Find the letter for the number you’ve calculated. This is the ciphertext letter.
6. Repeat steps 2 to 5 for every letter in the plaintext message.

Look at the following table to see how this is done with each letter in “Hello. How are you?” with key 13. Each column shows the steps for turning the plaintext letter on the left to the ciphertext letter on the right.

Table 1-1. The steps to encrypt “Hello. How are you?” with paper and pencil.

Plaintext Letter	Plaintext Number	+	Key	Result	Subtract 26?	Result	Ciphertext Letter
H	7	+	13	= 20		= 20	20 = U
E	4	+	13	= 17		= 17	17 = R
L	11	+	13	= 24		= 24	24 = Y
L	11	+	13	= 24		= 24	24 = Y
O	14	+	13	= 27	- 26	= 1	1 = B
H	7	+	13	= 20		= 20	20 = U
O	14	+	13	= 27	- 26	= 1	1 = B
W	22	+	13	= 35	- 26	= 9	9 = J
A	0	+	13	= 13		= 13	13 = N
R	17	+	13	= 30	- 26	= 4	4 = E
E	4	+	13	= 17		= 17	17 = R
Y	24	+	13	= 37	- 26	= 11	11 = L
O	14	+	13	= 27	- 26	= 1	1 = B
U	20	+	13	= 33	- 26	= 7	7 = H

To decrypt, you will have to understand what negative numbers are. If you don't know how to add and subtract with negative numbers, there is a tutorial on it here: <http://invpy.com/neg>.

To decrypt, subtract the key instead of adding it. For the ciphertext letter B, the number is 1. Subtract $1 - 13$ to get -12 . Like our “subtract 26” rule for encrypting, when we are decrypting and the result is less than 0, we have an “add 26” rule. $-12 + 26$ is 14. So the ciphertext letter B decrypts back to letter O.

Table 1-2. The steps to decrypt the ciphertext with paper and pencil.

Ciphertext Letter	Ciphertext Number	-	Key	Result	Add 26?	Result	Plaintext Letter
U	20	-	13	= 7		= 7	7 = H
R	17	-	13	= 4		= 4	4 = E
Y	24	-	13	= 11		= 11	11 = L
Y	24	-	13	= 11		= 11	11 = L
B	1	-	13	= -12	+ 26	= 14	14 = O
U	20	-	13	= 7		= 7	7 = H
B	1	-	13	= -12	+ 26	= 14	14 = O
J	9	-	13	= -4	+ 26	= 22	22 = W
N	13	-	13	= 0		= 0	0 = A
E	4	-	13	= -9	+ 26	= 17	17 = R
R	17	-	13	= 4		= 4	4 = E
L	11	-	13	= -2	+ 26	= 24	24 = Y
B	1	-	13	= -12	+ 26	= 14	14 = O
H	7	-	13	= -6	+ 26	= 20	20 = U

As you can see, we don't need an actual cipher wheel to do the Caesar cipher. If you memorize the numbers and letters, then you don't even need to write out the alphabet with the numbers under them. You could just do some simple math in your head and write out secret messages.

Practice Exercises, Chapter 1, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice1B>.

Double-Strength Encryption?

You might think that encrypting a message twice with two different keys would double the strength of our encryption. But this turns out not to be the case with the Caesar cipher (and most other ciphers). Let's try double-encrypting a message to see why.

If we encrypt the word “KITTEN” with the key 3, the resulting cipher text would be “NLWWHQ”. If we encrypt the word “NLWWHQ” with the key 4, the resulting cipher text of that would be “RPAALU”. But this is exactly the same as if we had encrypted the word “KITTEN” once with a key of 7. Our “double” encryption is the same as normal encryption, so it isn’t any stronger.

The reason is that when we encrypt with the key 3, we are adding 3 to plaintext letter’s number. Then when we encrypt with the key 4, we are adding 4 to the plaintext letter’s number. But adding 3 and then adding 4 is the exact same thing as adding 7. Encrypting twice with keys 3 and 4 is the same as encrypting once with the key 7.

For most encryption ciphers, encrypting more than once does not provide additional strength to the cipher. In fact, if you encrypt some plaintext with two keys that add up to 26, the ciphertext you end up with will be the same as the original plaintext!

Programming a Computer to do Encryption

The Caesar cipher, or ciphers like it, were used to encrypt secret information for several centuries. Here’s a cipher disk of a design invented by Albert Myer that was used in the American Civil War in 1863.



Figure 1-13. American Civil War Union Cipher Disk at the National Cryptologic Museum.

If you had a very long message that you wanted to encrypt (say, an entire book) it would take you days or weeks to encrypt it all by hand. This is how programming can help. A computer could do

the work for a large amount of text in less than a second! But we need to learn how to instruct (that is, program) the computer to do the same steps we just did.

We will have to be able to speak a language the computer can understand. Fortunately, learning a programming language isn't nearly as hard as learning a foreign language like Japanese or Spanish. You don't even need to know much math besides addition, subtraction, and multiplication. You just need to download some free software called Python, which we will cover in the next chapter.



INSTALLING PYTHON

Topics Covered In This Chapter:

- Downloading and installing Python
- Downloading the Pyperclip module
- How to start IDLE
- Formatting used in this book
- Copying and pasting text

“Privacy in an open society also requires cryptography. If I say something, I want it heard only by those for whom I intend it. If the content of my speech is available to the world, I have no privacy.”

Eric Hughes, “A Cypherpunk’s Manifesto”, 1993
<http://invpy.com/cypherpunk>

The content of this chapter is very similar to the first chapter of *Invent Your Own Computer Games with Python*. If you have already read that book or have already installed Python, you only need to read the “Downloading pyperclip.py” section in this chapter.

Downloading and Installing Python

Before we can begin programming, you'll need to install software called the Python interpreter. (You may need to ask an adult for help here.) The interpreter is a program that understands the instructions that you'll write in the Python language. Without the interpreter, your computer won't understand these instructions. (We'll refer to "the Python interpreter" as "Python" from now on.)

Because we'll be writing our programs in the Python language we need to download Python from the official website of the Python programming language, <http://www.python.org>. The installation is a little different depending on if your computer's operating system is Windows, OS X, or a Linux distribution such as Ubuntu. You can also find videos of people installing the Python software online at <http://invidious.com/installing>.

Important Note! Be sure to install Python 3, and not Python 2. The programs in this book use Python 3, and you'll get errors if you try to run them with Python 2. It is so important, I am adding a cartoon penguin telling you to install Python 3 so that you do not miss this message:



Figure 2-1. "Be sure to install Python 3, not Python 2!", says the incongruous penguin.

Windows Instructions

There is a list of links on the left side of the web page at <http://www.python.org>. Click on the Download link to go to the download page, then look for the file called Python 3.3.0 Windows Installer ("Windows binary — does not include source") and click on its link to download Python for Windows. (If there is a newer version than Python 3.3.0, you can download that one.) Double-click on the *python-3.3.0.msi* file that you've just downloaded to start the Python installer. (If it doesn't start, try right-clicking the file and choosing Install.) Once the installer starts up, click the Next button and accept the choices in the installer as you go. There's no need to make any changes. When the installer is finished, click Finish.

OS X Instructions

The installation for OS X is similar. Instead of downloading the .msi file from the Python website, download the .dmg Mac Installer Disk Image file instead. The link to this file will look something like “Python 3.3.0 Mac OS X” on the “Download Python Software” web page.

Ubuntu and Linux Instructions

If your operating system is Ubuntu, you can install Python by opening a terminal window (click on **Applications ► Accessories ► Terminal**) and entering `sudo apt-get install python3.3` then pressing Enter. You will need to enter the root password to install Python, so ask the person who owns the computer to type in this password.

You also need to install the IDLE software. From the terminal, type in `sudo apt-get install idle3`. You will also need the root password to install IDLE.

Downloading pyperclip.py

Almost every program in this book uses a custom module I wrote called *pyperclip.py*. This module provides functions for letting your program copy and paste text to the clipboard. This module does not come with Python, but you can download it from: <http://invpy.com/pyperclip.py>

This file must be in the same folder as the Python program files that you type. (A folder is also called a directory.) Otherwise you will see this error message when you try to run your program:

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

Starting IDLE

We will be using the IDLE software to type in our programs and run them. IDLE stands for **I**nteractive **D**evelopment **E**nvironment. While Python is the software that interprets and runs your Python programs, the IDLE software is what you type your programs in.

If your operating system is Windows XP, you should be able to run Python by clicking the Start button, then selecting **Programs ► Python 3.3 ► IDLE (Python GUI)**. For Windows Vista or Windows 7, click the Windows button in the lower left corner, type “IDLE” and select “IDLE (Python GUI)”.

If your operating system is Max OS X, start IDLE by opening the Finder window and clicking on Applications, then click Python 3.3, then click the IDLE icon.

If your operating system is Ubuntu or Linux, start IDLE by clicking Applications ► Accessories ► Terminal and then type `idle3`. You may also be able to click on Applications at the top of the screen, and then select Programming and then IDLE 3.



Figure 2-2. IDLE running on Windows (left), OS X (center), and Ubuntu Linux (right).

The window that appears will be mostly blank except for text that looks something like this:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

The window that appears when you first run IDLE is called the interactive shell. A **shell** is a program that lets you type instructions into the computer. The Python shell lets you type Python instructions in and then sends these instructions to the Python interpreter software to run. We can type Python instructions into the shell and, because the shell is interactive, the computer will read our instructions and perform them immediately.

The Featured Programs

“Hacking Secret Ciphers with Python” is different from other programming books because it focuses on the source code for 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.

As you read through this book, type the source code from this book into IDLE yourself. But you can also download the source code files from this book’s website. Go to the web site <http://invpy.com/hackingsource> and follow the instructions to download the source code files.

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

going back to the previous chapters. Or email your programming questions to the author at al@inventwithpython.com.

Line Numbers and Spaces

When entering the source code yourself, do not type the line numbers that appear at the beginning of each line. For example, if you see this in the book:

```
1. number = random.randint(1, 20)
2. spam = 42
3. print('Hello world!')
```

...then you do not need to type the “1.” on the left side, or the space that immediately follows it. Just type it like this:

```
number = random.randint(1, 20)
spam = 42
print('Hello world!')
```

Those numbers are only used so that this book can refer to specific lines in the code. They are not a part of the actual program. Aside from the line numbers, be sure to enter the code exactly as it appears. This includes the letter casing. In Python, `HELLO` and `hello` and `Hello` could refer to three different things.

Notice that some of the lines don’t begin at the leftmost edge of the page, but are indented by four or eight spaces. Be sure to put in the correct number of spaces at the start of each line. (Since each character in IDLE is the same width, you can count the number of spaces by counting the number of characters above or below the line you’re looking at.)

For example, you can see that the second line is indented by four spaces because the four characters (“`while`”) on the line above are over the indented space. The third line is indented by another four spaces (the four characters “`if n`” are above the third line’s indented space):

```
while spam < 10:
    if number == 42:
        print('Hello')
```

Text Wrapping in This Book

Some lines of code are too long to fit on one line on the page, and the text of the code will wrap around to the next line. When you type these lines into the file editor, enter the code all on one line without pressing Enter.

You can tell when a new line starts by looking at the line numbers on the left side of the code. The example below has only two lines of code, even though the first line is too long to fit on the page:

```
1. print('This is the first line! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxx')  
2. print('This is the second line!')
```

Tracing the Program Online

Tracing a program means to step through the code one line at a time, in the same way that a computer would execute it. You can visit <http://invpy.com/hackingtraces> to see a trace through of each program in this book. The traces web page has notes and helpful reminders at each step of the trace to explain what the program is doing, so it can help you better understand why these programs work the way they do.

Checking Your Typed Code with the Online Diff Tool

Although it is very helpful to learn Python by typing out the source code for these programs, you may accidentally make typos that cause your programs to crash. It may not be obvious where the typo is.

You can copy and paste the text of your typed source code to the online diff tool on the book's website. The diff tool will show any differences between the source code in the book and the source code you've typed. This is an easy way of finding any typos in your programs.

The online diff tool is at this web page: <http://invpy.com/hackingdiff>. A video tutorial of how to use the diff tool is available from this book's website at <http://invpy.com/hackingvideos>.

Copying and Pasting Text

Copying and pasting text is a very useful computer skill, especially for this book because many of the texts that will be encrypted or decrypted are quite long. Instead of typing them out, you can look at electronic versions of the text on this book's website and copy the text from your browser and paste it into IDLE.

To copy and paste text, you first need to drag the mouse over the text you want to copy. This will highlight the text. Then you can either click on the **Edit ► Copy** menu item, or on Windows press Ctrl-C. (That's press and hold the Ctrl button, then press C once, then let go of the Ctrl button.) On Macs, the keyboard shortcut is Command-C (the ⌘ button). This will copy the highlighted text to the computer's memory, or clipboard.



THE INTERACTIVE SHELL

Topics Covered In This Chapter:

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

Before we start writing encryption programs we should first learn some basic programming concepts. These concepts are values, operators, expressions, and variables. If you've read the *Invent Your Own Computer Games with Python* book (which can be downloaded for free from <http://inventwithpython.com>) or already know Python, you can skip directly to chapter 5.

Let's start by learning how to use Python's interactive shell. You should read this book while near your computer, so you can type in the short code examples and see for yourself what they do.

Some Simple Math Stuff

Start by opening IDLE. You will see the interactive shell and the cursor blinking next to the `>>>` (which is called the **prompt**). The interactive shell can work just like a calculator. Type `2 + 2` into the shell and press the Enter key on your keyboard. (On some keyboards, this is the Return key.) As you can see in Figure 3-1, the computer should respond with the number 4.

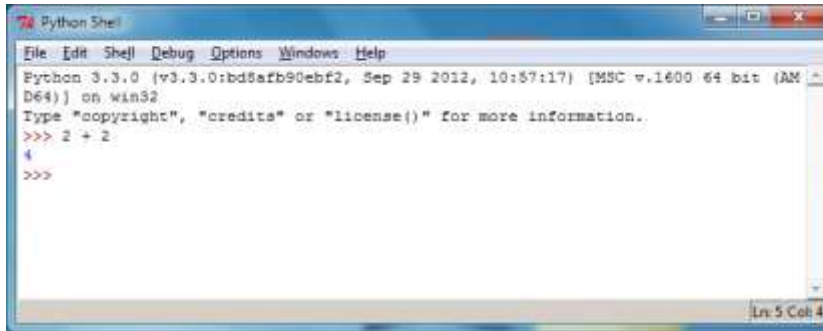


Figure 3-1. Type 2+2 into the shell.

$2 + 2$ isn't a program by itself, it's just a single instruction (we're just learning the basics right now). The $+$ sign tells the computer to add the numbers 2 and 2. To subtract numbers use the $-$ sign. To multiply numbers use an asterisk ($*$) and to divide numbers use $/$.

Table 3-1: The various math operators in Python.

Operator	Operation
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division

When used in this way, $+$, $-$, $*$, and $/$ are called **operators** because they tell the computer to perform an operation on the numbers surrounding them. The 2s (or any other number) are called **values**.

Integers and Floating Point Values

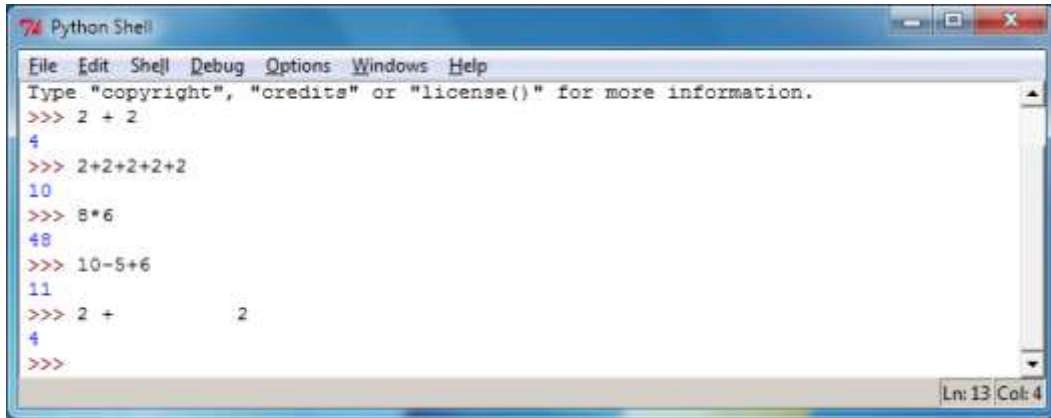
In programming whole numbers like 4, 0, and 99 are called **integers**. Numbers with fractions or decimal points (like 3.5 and 42.1 and 5.0) are **floating point numbers**. In Python, the number 5 is an integer, but if we wrote it as 5.0 it would be a floating point number

Expressions

Try typing some of these math problems into the shell, pressing Enter key after each one:

```
2+2+2+2+2
8*6
10-5+6
2 +      2
```

Figure 3-2 is what the interactive shell will look like after you type in the previous instructions.



```
Python Shell
File Edit Shell Debug Options Windows Help
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>> 2+2+2+2+2
10
>>> 8*6
48
>>> 10-5+6
11
>>> 2 +      2
4
>>>
```

Figure 3-2. What the IDLE window looks like after entering instructions.

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). There can be any amount of spaces in between the integers and these operators. But be sure to always start at the very beginning of the line though, with no spaces in front.



Figure 3-3. An expression is made up of values (like 2) and operators (like +).

Order of Operations

You might remember “order of operations” from your math class. For example, multiplication has a higher priority than addition. Python copies this for the `*` and `+` operators. If an expression has both `*` and `+` operators, the `*` operator is evaluated first. Type the following into the interactive shell:

```
>>> 2 + 4 * 3 + 1
15
>>>
```


Because the `*` operator is evaluated first, `2 + 4 * 3 + 1` evaluates to `2 + 12 + 1` and then evaluates to 15. It does not evaluate to `6 * 3 + 1`, then to `18 + 1`, and then to 19. However, you can always use parentheses to change which operations should happen first. Type the following into the interactive shell:

```
>>> (2 + 4) * (3 + 1)
24
>>>
```

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.

An expression will always evaluate (that is, shorten down to) a single value.

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.

However, if you type only `5 +` into the interactive shell, you will get an error message.

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

This error happened because `5 +` is not an expression. Expressions have values connected by operators, but in the Python language the `+` operator expects to connect two values. We have only given it one in “`5 +`”. This is why the error message appeared. A syntax error means that the computer does not understand the instruction you gave it because you typed it incorrectly. This may not seem important, but a lot of computer programming is not just telling the computer what to do, but also knowing exactly how to tell the computer to do it.

Errors are Okay!

It’s perfectly okay to make errors! You will not break your computer by typing in bad code that causes errors. If you type in code that causes an error, Python simply says there was an error and then displays the `>>>` prompt again. You can keep typing in new code into the interactive shell.

Until you get more experience with programming, the error messages might not make a lot of sense to you. You can always Google the text of the error message to find web pages that talk about that specific error. You can also go to <http://invy.com/errors> to see a list of common Python error messages and their meanings.

Practice Exercises, Chapter 3, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice3A>.

Every Value has a Data Type

“Integer” and “floating point” are known as **data types**. Every value has a data type. The value 42 is a value of the integer data type. We will say 42 is an **int** for short. The value 7.5 is a value of the floating point data type. We will say 7.5 is a **float** for short.

There are a few other data types that we will learn about (such as strings in the next chapter), but for now just remember that any time we say “value”, that value is of a certain data type. It’s usually easy to tell the data type just from looking at how the value is typed out. Ints are numbers without decimal points. Floats are numbers with decimal points. So 42 is an int, but 42.0 is a float.

Storing Values in Variables with Assignment Statements

Our programs will often want to save the values that our expressions evaluate to so we can use them later. We can store values in **variables**.

Think of a variable as like a box that can hold values. You can store values inside variables with 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
>>>
```



Figure 3-4. Variables are like boxes with names that can hold values in them.

You can think of the variable like a box with the value 15 inside of it (as shown in Figure 3-4). 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 small note inside the box.

When you press Enter you won’t see anything in response, other than a blank line. Unless you see an error message, you can assume that the instruction has been executed successfully. The next `>>>` prompt will appear so that you can type in 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 do not evaluate to any value, they just perform some action. This is why there is no value displayed on the next line in the shell.

It might be confusing to know which instructions are expressions and which are statements. **Just remember that if a Python instruction evaluates to a single value, it’s an expression. If a Python instruction does not, then it’s a statement.**

An assignment statement is written as a variable, followed by the = operator, followed by an expression. The value that the expression evaluates to is stored inside the variable. (The value 15 by itself is an expression that evaluates to 15.)

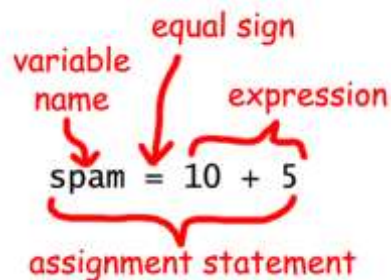


Figure 3-5. The parts of an assignment statement.

Remember, variables store single values, not expressions. For example, if we had the statement, `spam = 10 + 5`, then the expression `10 + 5` would first be evaluated to 15 and then the value 15 would be stored in the variable `spam`. A variable is created the first time you store a value in it by using an assignment statement.

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

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

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

That may seem odd but it makes sense when we remember that we set the value of `spam` to 15. Because we've set the value of the variable `spam` to 15, the expression `spam + 5` evaluates to the expression `15 + 5`, which then evaluates to 20. A variable name in an expression evaluates to the value stored in that variable.

Overwriting Variables

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

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
>>>
```

The first time we enter `spam + 5`, the expression evaluates to 20, because we stored the value 15 inside the variable `spam`. But when we enter `spam = 3`, the value 15 is **overwritten** (that is, replaced) with the value 3. Now, when we enter `spam + 5`, the expression evaluates to 8 because the `spam + 5` now evaluates to `3 + 5`. The old value in `spam` is forgotten.

To find out what the current value is inside a variable, enter the variable name into the shell.

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

This happens because a variable by itself is an expression that evaluates to the value stored in the variable. This is just like how a value by itself is also an expression that evaluates to itself:

```
>>> 15
15
```

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

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

The assignment statement `spam = spam + 5` is like saying, “the new value of the `spam` variable will be the current value of `spam` plus five.” Remember that the variable on the left side of the `=` sign will be assigned the value that the expression on the right side evaluates to. We can keep increasing the value in `spam` by 5 several times:

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

Using More Than One Variable

Your programs can have as many variables as you need. For example, let’s assign different values to two variables named `eggs` and `fizz`:

```
>>> fizz = 10
>>> eggs = 15
```

Now the `fizz` variable has 10 inside it, and `eggs` has 15 inside it.

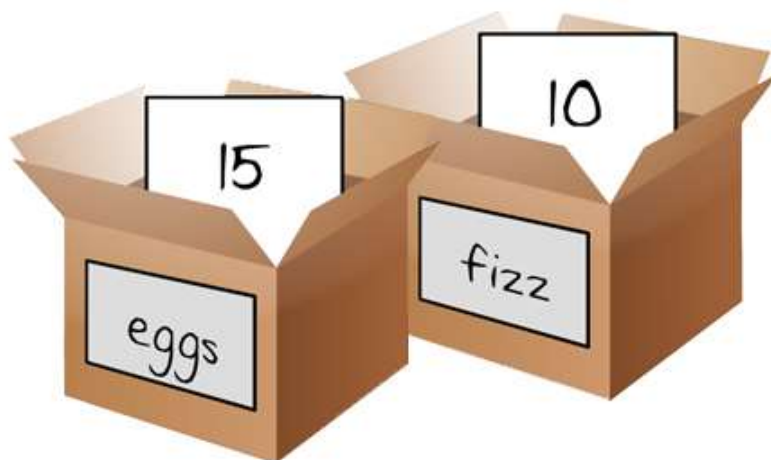


Figure 3-6. The “fizz” and “eggs” variables have values stored in them.

Let’s try assigning a new value to the `spam` variable. Enter `spam = fizz + eggs` into the shell, then enter `spam` into the shell to see the new value of `spam`. Type the following into the interactive shell:

```
>>> fizz = 10
>>> eggs = 15
>>> spam = fizz + eggs
>>> spam
25
>>>
```

The value in `spam` is now 25 because when we add `fizz` and `eggs` we are adding the values stored inside `fizz` and `eggs`.

Variable Names

The computer doesn’t care what you name your variables, but you should. Giving variables names that reflect what type of data they contain makes it easier to understand what a program does. Instead of `name`, we could have called this variable `abrahamLincoln` or `monkey`. The computer will run the program the same (as long as you consistently use `abrahamLincoln` or `monkey`).

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 to be an entirely separate variable. So `spam`, `SPAM`, `Spam`, and `sPAM` are considered to be four different variables in Python. They each can contain their own separate values.

It's a bad idea to have differently-cased variables in your program. If you stored your first name in the variable `name` and your last name in the variable `NAME`, it would be very confusing when you read your code weeks after you first wrote it. Did `name` mean first and `NAME` mean last, or the other way around?

If you accidentally switch the `name` and `NAME` variables, then your program will still run (that is, it won't have any "syntax" errors) but it will run incorrectly. This type of flaw in your code is called a **bug**. A lot of programming is not just writing code but also fixing bugs.

Camel Case

It also helps to capitalize variable names if they include more than one word. If you store a string of what you had for breakfast in a variable, the variable name `whatIHadForBreakfast` is much easier to read than `whatihadforbreakfast`. This is called **camel case**, since the casing goes up and down like a camel's humps. This is a **convention** (that is, an optional but standard way of doing things) in Python programming. (Although even better would be something simple, like `todaysBreakfast`. Capitalizing the first letter of each word after the first word in variable names makes the program more readable.

Practice Exercises, Chapter 3, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice3B>.

Summary - But When Are We Going to Start Hacking?

Soon. But before we can hack ciphers, we need to learn some more basic programming concepts. We won't need to learn a lot before we start writing encryption programs, but there's one more chapter on programming we need to cover.

In this chapter you learned the basics about writing Python instructions in the interactive shell. Python needs you to tell it exactly what to do in a strict way, because computers don't have common sense and only understand very simple instructions. You have 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 have also learned that you can store values inside of variables so that your program can remember them to use them later on.

The interactive shell is a very useful tool for learning what Python instructions do because it lets you type them in one at a time and see the results. In the next chapter, we will be creating programs of many instructions that are executed in sequence rather than one at a time. We will go over some more basic concepts, and you will write your first program!



STRINGS AND WRITING PROGRAMS

Topics Covered In This Chapter:

- Strings
- String concatenation and replication
- Using IDLE to write source code
- Saving and running programs in IDLE
- The `print()` function
- The `input()` function
- Comments

That's enough of integers and math for now. Python is more than just a calculator. In this chapter, we will learn how to store text in variables, combine text together, and display text on the screen. We will also make our first program, which greets the user with the text, “Hello World!” and lets the user type in a name.

Strings

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 'Tqe kg im npqv, jst kg im oapxe.'. The plaintext and ciphertext are represented in our program as string values, and there's a lot of ways that Python code can manipulate these values.

We can store string values inside variables just like integer and floating point values. When we type strings, we put them in between two single quotes (') to show where the string starts and ends. Type this in to 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 type spam into the shell, you should see the contents of the spam variable (the 'hello' string.) This is because Python will evaluate a variable to the value stored inside it: in this case, the string 'hello'.

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

Strings can have almost any keyboard character in them. (We'll talk about special “escape characters” later.) These are all examples of strings:

```
>>> 'hello'
'hello'
>>> 'Hi there!'
'Hi there!'
>>> '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*&0cfsdY0*&gfC%Y0*%%3yc8r2'
'O*##wY*&0cfsdY0*&gfC%Y0*%%3yc8r2'
```

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

String Concatenation with the + Operator

You can add together two string values into one new string value by using the + operator. Doing this is called **string concatenation**. Try entering 'Hello' + 'World!' into the shell:

```
>>> 'Hello' + 'World!'
'HelloWorld!'
>>>
```

To put a space between “Hello” and “World!”, put a space at the end of the 'Hello' string and before the single quote, like this:

```
>>> 'Hello ' + 'World!'
'Hello World!'
>>>
```

Remember, Python will concatenate *exactly* the strings you tell it to concatenate. If you want a space in the resulting string, there must be a space in one of the two original strings.

The + operator can concatenate two string values into a new string value ('Hello ' + 'World!' to 'Hello World! '), just like it could add two integer values into a new integer value (2 + 2 to 4). Python knows what the + operator should do because of the data types of the values. Every value is of a data type. The data type of the value 'Hello' is a string. The data type of the value 5 is an integer. The data type of the data that tells us (and the computer) what kind of data the value is.

The + operator can be used in an expression with two strings or two integers. If you try to use the + operator with a string value and an integer value, you will get an error. Type this code into the interactive shell:

```
>>> 'Hello' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> 'Hello' + '42'
'Hello42'
>>>
```

String Replication with the * Operator

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

```
>>> 'Hello' * 3
HelloHelloHello
>>> spam = 'Abcdef'
>>> spam = spam * 3
>>> spam
'AbcdefAbcdefAbcdef'
>>> spam = spam * 2
>>> spam
'AbcdefAbcdefAbcdefAbcdefAbcdefAbcdef'
>>>
```

The `*` operator can work with two integer values (it will multiply them). It can also work with a string value and an integer value (it will replicate the string). But it cannot work with two string values, which would cause an error:

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

What string concatenation and string replication show is that operators in Python can do different things 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.

Printing Values with the `print()` Function

There is another type of Python instruction called a `print()` function call. Type the following into the interactive shell:

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

A function (like `print()` in the above example) has code in that performs a task, such as printing values on the screen. There are many different functions that come with Python. To **call** a function means to execute the code that is inside the function.

The instructions in the above example pass a value to the `print()` function in between the parentheses, and the `print()` function will print the value to the screen. The values that are passed when a function is called are called **arguments**. (Arguments are the same as values though. We just call values this when they are passed to function calls.) When we begin to write programs, the way we make text appear on the screen is with the `print()` function.

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

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

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

Escape Characters

Sometimes we might want to use a character that cannot easily be typed into a string value. For example, we might want to put a single quote character as part of a string. But we would 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 just the rest of the string. Type the following into the interactive shell:

```
>>> print('Al's cat is named Zophie.')
File "<stdin>", line 1
    print('Al's cat is named Zophie.')
          ^
SyntaxError: invalid syntax
>>>
```

To use a single quote in a string, we need to use escape characters. 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. Type the following into the interactive shell:

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

An escape character helps us print out letters that are hard to type into the source code. Table 4-1 shows some escape characters in Python:

Table 4-1. Escape Characters

Escape Character	What Is Actually Printed
\\	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. This line of code would not work:

```
>>> print('He flew away in a green\teal helicopter.')
He flew away in a green    eal helicopter.
```

This is because the “t” in “teal” was seen as an escape character since it came after a backslash. The escape character `\t` simulates pushing the Tab key on your keyboard. Escape characters are there so that strings can have characters that cannot be typed in.

Instead, try this code:

```
>>> print('He flew away in a green\\teal helicopter.')
He flew away in a green\teal helicopter.
```

Quotes and Double Quotes

Strings don’t always have to be in 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 cannot mix single and double quotes. This line will give you an error:

```
>>> print('Hello world")
SyntaxError: EOL while scanning single-quoted string
>>>
```

I like to use single quotes so I don't have to hold down the shift key on the keyboard to type them. It's easier to type, and the computer doesn't care either way.

But remember, 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('I asked to borrow Alice\'s car for a week. She said, "Sure."')
I asked to borrow Alice's car for a week. She said, "Sure."
>>> print("She said, \"I can't believe you let him borrow your car.\")
She said, "I can't believe you let him borrow your car."
```

You do not need to escape double quotes in single-quote strings, and you do not need to escape single quotes in the 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.

Practice Exercises, Chapter 4, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice4A>.

Indexing

Your encryption programs will often need to get a single character from a string. Indexing is the adding of square brackets `[` and `]` to the end of a string value (or a variable containing a string) with a number between them. This number is called the **index**, and tells Python which position in the string has the character you want. 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.

Type 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'`, since H is the first character in the string `'Hello'`. **Remember that indexes start at 0, not 1.** This is why the H's index is 0, not 1.

```
string: ' H e l l o '
indexes: 0 1 2 3 4
```

Figure 4-1. The string `'Hello'` and its indexes.

Indexing can be used with a variable containing a string value or a string value by itself such as `'Zophie'`. Type this into the interactive shell:

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

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

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

If you enter an index that is too large for the string, Python will display an “index out of range” error message. There are only 5 characters in the string `'Hello'`. If we try to use the index 10, then Python will display an error saying that our index is “out of range”:

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

Negative Indexes

Negative indexes start at the end of a string and go backwards. 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.

Type 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 will be easier to use negative indexes.

Slicing

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 separate by a `:` colon. Type the following into the interactive shell:

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

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

Try typing the following into the interactive shell:

```
>>> 'Hello world!' [0:5]
```



```
'Hello'
>>> 'Hello world!' [6:12]
'world!'
>>> 'Hello world!' [-6:-1]
'world'
>>> 'Hello world!' [6:12][2]
'r'
>>>
```

Notice that the expression `'Hello world!' [6:12][2]` first evaluates to `'world!' [2]` which is an indexing that further evaluates to `'r'`.

Unlike indexes, slicing will never give you an error if you give it too large of an index for the string. It will 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.

Blank Slice Indexes

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

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

If you leave out the second index, Python will automatically think you want to specify the rest of the string:

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

```
>>>
```

Slicing is a simple way to get a “substring” from a larger string. (But really, a “substring” is still just a string value like any other string.) Try typing the following into the shell:

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

Practice Exercises, Chapter 4, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice4B>.

Writing Programs in IDLE’s File Editor

Until now we have been typing instructions one at a time into the interactive shell. When we write programs though, we type in several instructions and have them run without waiting on us for the next one. Let’s write our first program!

The name of the software program that provides the interactive shell is called IDLE, the **I**nteractive **D**evelopment **E**nvironment. IDLE also has another part besides the interactive shell called the file editor.

At the top of the Python shell window, click on the **File ► New Window**. A new blank window will appear for us to type our program in. This window is the **file editor**. The bottom right of the file editor window will show you line and column that the cursor currently is in the file.

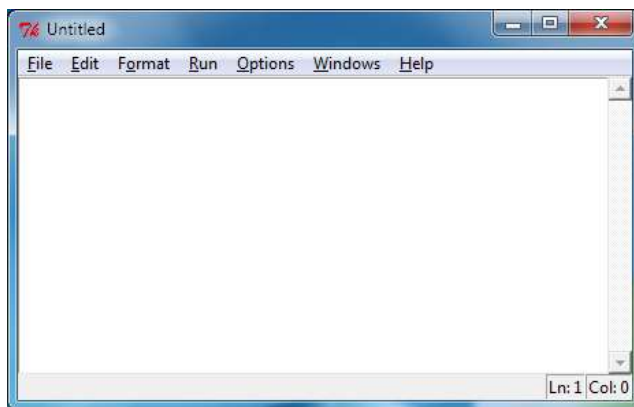


Figure 4-2. The file editor window. The cursor is at line 1, column 0.

You can always tell the difference between the file editor window and the interactive shell window because the interactive shell will always have the `>>>` prompt in it.

Hello World!

A tradition for programmers learning a new language is to make their first program display the text “Hello world!” on the screen. We’ll create our own Hello World program now.

Enter the following 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.

Source Code of Hello World

This code can be downloaded from <http://inventwithpython.com/hello.py>. If you get errors after typing this code in, compare it to the book’s code with the online diff tool at <http://inventwithpython.com/hackingdiff> (or email me at al@inventwithpython.com if you are still stuck.)

```

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 give different types of instructions different colors. After you are done typing this code in, the window should look like this:

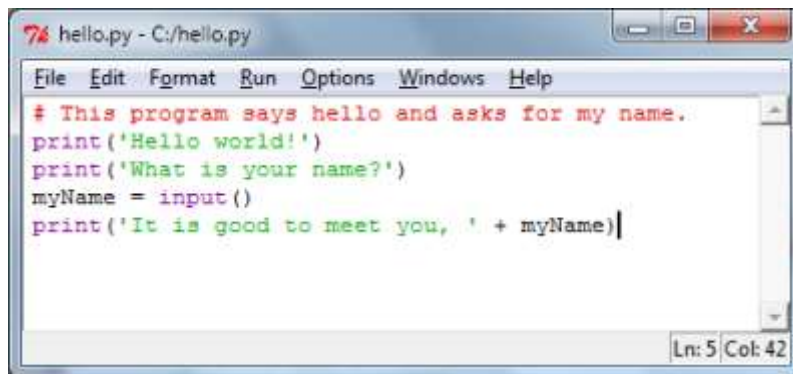


Figure 4-3. The file editor window will look like this after you type in the code.

Saving Your Program

Once you've entered your source code, save it so that you won't have to retype it each time we start IDLE. To do so, from the menu at the top of the File Editor window, choose **File ► Save As**. The Save As window should open. Enter *hello.py* in the File Name field, then click **Save**. (See Figure 4-4.)

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit from IDLE you won't lose everything you've typed. As a shortcut, you can press Ctrl-S on Windows and Linux or ⌘-S on OS X to save your file.

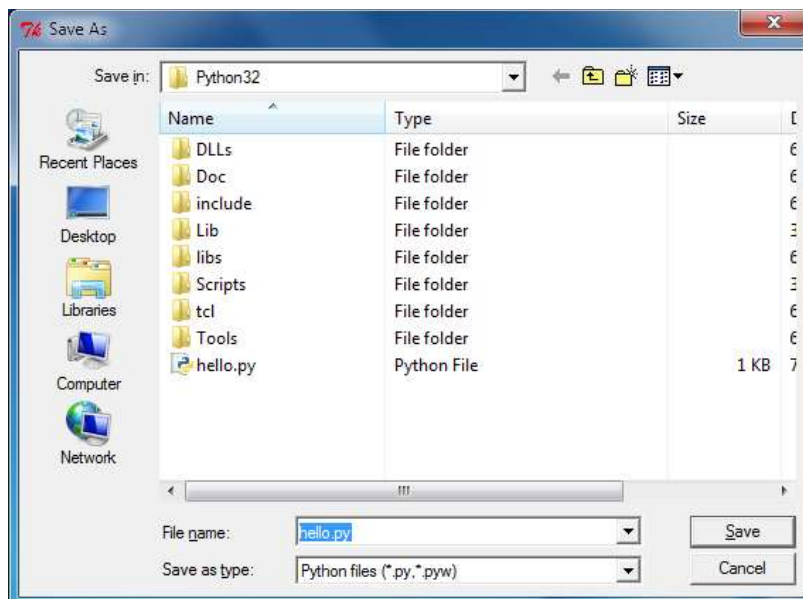


Figure 4-4. Saving the program.

A video tutorial of how to use the file editor is available from this book's website at <http://invpy.com/hackingvideos>.

Running Your Program

Now it's time to run our program. Click on **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, you have to press **F5** from the file editor's window, not the interactive shell's window.

When your program asks for your name, go ahead and enter it as shown in Figure 4-5:

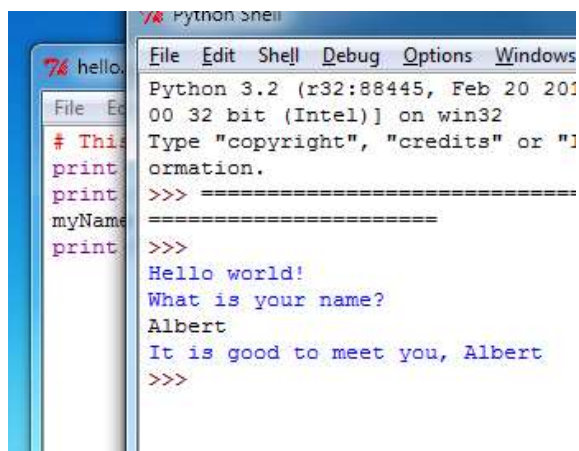


Figure 4-5. What the interactive shell looks like when running the “Hello World” program.

Now when you push 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. (You can run this program again if you like by pressing **F5** again.)

If you get an error that looks like this:

```
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
```

...this means you are running the program with Python 2, instead of Python 3. This makes the penguin in the first chapter sad. (The error is caused by the `input()` function call, which does different things in Python 2 and 3.) Please install Python 3 from <http://python.org/getit> before continuing.

Opening The Programs You've Saved

Close the file editor by clicking on 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* and press the **Open** button. Your saved *hello.py* program should open in the File Editor window.

How the “Hello World” Program Works

Each line that we entered 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. Each instruction is followed in sequence, beginning from the very top of the program and working down the list of instructions. After the program executes the first line of instructions, it moves on and executes the second line, then the third, and so on.

We call the program's following of instructions step-by-step the **program execution**, or just the **execution** for short. The execution starts at the first line of code and then moves downward. The execution can skip around instead of just going from top to bottom, and we'll find out how to do this in the next chapter.

Let's look at our program one line at a time to see what it's doing, beginning with line number 1.

Comments

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

hello.py

This line is called a **comment**. Comments are not for the computer, but for you, the programmer. The computer ignores them. They're used to remind you of what the program does or to tell others who might look at your code what it is that your code is trying to do. Any text following a **#** sign (called the **pound sign**) is a comment. (To make it easier to read the source code, this book prints out comments in a light gray-colored text.)

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.

Functions

A **function** is kind of like a mini-program inside your program. It contains lines of code that are executed from top to bottom. Python provides some built-in functions that we can use (you’ve already used the `print()` function). The great thing about functions is that we only need to know what the function does, but not how it does it. (You need to know that the `print()` function displays text on the screen, but you don’t need to know how it does this.)

A **function call** is a piece of code that tells our program to run the code inside a function. For example, your program can call the `print()` function whenever you want to display a string on the screen. The `print()` function takes the value you type in between the parentheses as input and displays the text on the screen. Because we want to display `Hello world!` on the screen, we type the `print` function name, followed by an opening parenthesis, followed by the `'Hello world!'` string and a closing parenthesis.

The `print()` function

```

2. print('Hello world!')
3. print('What is your name?')
hello.py

```

This line is a call to the `print()` function (with the string to be printed going inside the parentheses). 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 let us know we are talking about a function, much like the quotes around the number `'42'` tell us that we are talking about the string `'42'` and not the integer `42`.

Line 3 is another `print()` function call. This time, the program displays “What is your name?”

The `input()` function

```

4. myName = input()
hello.py

```

Line 4 has an assignment statement with a variable (`myName`) and a function call (`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 types in (their name) becomes the string value that is stored in `myName`.

Like expressions, function calls evaluate to a single value. The value that the function call evaluates to is called the **return value**. (In fact, we can also use the word “returns” to mean the

same thing for function calls as “evaluates”.) In this case, the return value of the `input()` function is the string that the user typed in-their name. If the user typed in Albert, the `input()` function call evaluates (that is, returns) to the string `'Albert'`.

The function named `input()` does not need any arguments (unlike the `print()` function), which is why there is nothing in between the parentheses.

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

```
hello.py
```

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 our user input into the program. This is how we get the program to greet us by name.

Ending the Program

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

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

Remember, the computer only does exactly what you program it to do. In this program it is programmed to ask you for your name, let you type in a string, and then say hello and display the string you typed.

But computers are dumb. The program doesn’t care if you type in 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
```

Practice Exercises, Chapter 4, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice4C>.

Summary

Writing programs is just about knowing how to speak the computer's language. While you learned a little bit of this in the last chapter, in this chapter you've put together several Python instructions to make a complete program that asks for the user's name and then greets them.

All of our programs later in this book will be more complex and sophisticated, but don't worry. The programs will all be explained line by line. And you can always enter instructions into the interactive shell to see what they do before they are all put into a complete program.

Now let's start with our first encryption program: the reverse cipher.



THE REVERSE CIPHER

Topics Covered In This Chapter:

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

“Every man is surrounded by a neighborhood of voluntary spies.”

Jane Austen

The Reverse Cipher

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

The reverse cipher is a very weak cipher. Just by looking at its ciphertext you can figure out it is just 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.

Source Code of the Reverse Cipher Program

In IDLE, click on **File ► New Window** to create a new file editor window. Type in the following code, save it as *reverseCipher.py*, and press **F5** to run it: (Remember, don't type in the line numbers at the beginning of each line.)

Source code for reverseCipher.py

```

1. # Reverse Cipher
2. # http://inventwithpython.com/hacking (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 this program the output will look 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 (see <http://invy.py.com/copypaste> for instructions on how to copy and paste text) and paste it as the string value stored in *message* on line 4. Be sure to have the single quotes at the beginning and end of the string. The new line 4 will look like this (with the change in **bold**):

```

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

```

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

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

Checking Your Source Code with the Online Diff Tool

Even though you could copy and paste or download this code from this book’s website, it is very helpful to type in this program yourself. This will give you a better idea of what code is in this program. However, you might make some mistakes while typing it in yourself.

To compare the code you typed to the code that is in this book, you can use the book’s website’s online diff tool. Copy the text of your code and open <http://invpy.com/hackingdiff> in your web browser. Paste your code into the text field on this web page, and then click the Compare button. The diff tool will show any differences between your code and the code in this book. This is an easy way to find typos that are causing errors.

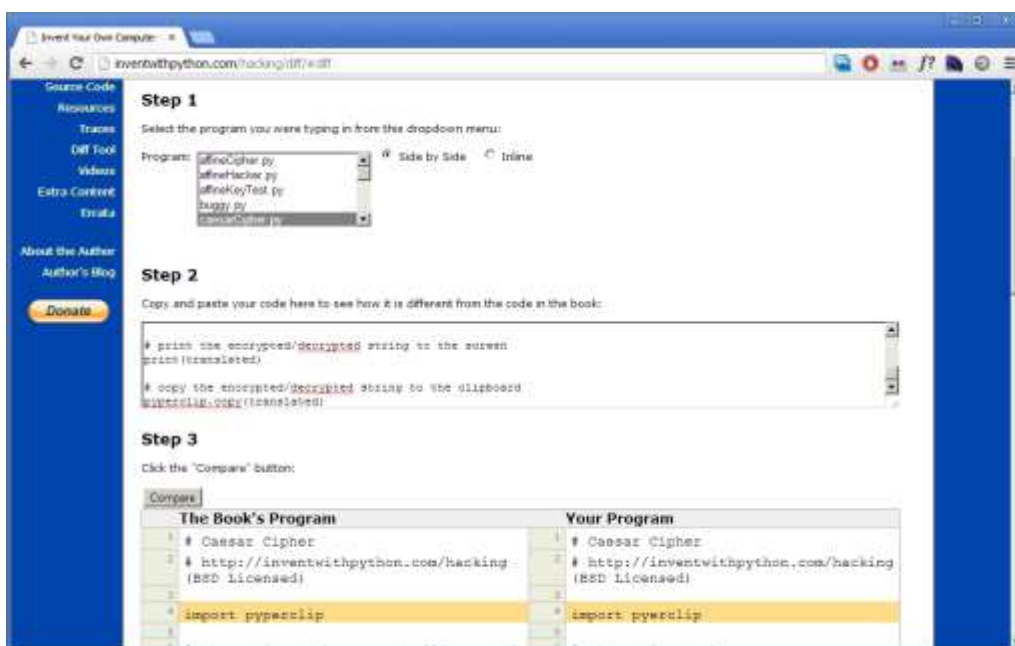


Figure 5-1. The online diff tool at <http://invpy/hackingdiff>

How the Program Works

```
reverseCipher.py
```

1. # Reverse Cipher
2. # <http://inventwithpython.com/hacking> (BSD Licensed)

The first two lines are comments explaining what the program is, and also the website where you can find it. The “BSD Licensed” part means that 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 <http://inventwithpython.com/hacking>) (The full text of the Berkeley Software

Distribution license can be seen at <http://invpy.com/bsd>) I like to have this info in the file so if it gets copied around the Internet, a person who downloads it will always know where to look for the original source. They'll also know this program is open source software and free to distribute to others.

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

Line 4 stores the string we want to encrypt in a variable named `message`. Whenever we want to encrypt or decrypt a new string we will just type the string directly into the code on line 4. (The programs in this book don't call `input()`, instead the user will type in the message into the source code. You can just change the source directly before running the program again to encrypt different strings.)

```
reverseCipher.py
5. translated = ''
```

The `translated` variable is where our program will store the reversed string. At the start of the program, it will contain the blank string. (Remember that the blank string is two single quote characters, not one double quote character.)

The `len()` Function

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

Line 6 is just a blank line, and Python will simply skip it. The next line of code is on line 7. This code is just an assignment statement that stores a value in a variable named `i`. The expression that is evaluated and stored in the variable is `len(message) - 1`.

The first part of this expression is `len(message)`. This is a function call to the `len()` function. The `len()` function accepts a string value argument (just like the `print()` function does) 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)` will tell us how many characters are in the string value stored in `message`.

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

```
>>> len('Hello')
5
```

```

>>> len('')
0
>>> spam = 'Al'
>>> len(spam)
2
>>> len('Hello' + ' ' + 'world!')
12
>>>

```

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 will return 2. If we pass the expression `'Hello' + ' ' + 'world!'` to the `len()` function, it returns 12. This is because `'Hello' + ' ' + 'world!'` will evaluate to the string value `'Hello world!'`, which has twelve characters in it. (The space and the exclamation point count as characters.)

Line 7 finds the number of characters in `message`, subtracts one, and then stores this number in the `i` variable. This will be the index of the last character in the `message` string.

Introducing the `while` Loop

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

`reverseCipher.py`

This is a new type of Python instruction called a `while` loop or `while` statement. A `while` loop is made up of four parts:

1. The `while` keyword.
2. An expression (also called a condition) that evaluates to the Boolean values `True` or `False`. (Booleans are explained next in this chapter.)
3. A `:` colon.
4. A block (explained later) of indented code that comes after it, which is what lines 9 and 10 are. (Blocks are explained later in this chapter.)

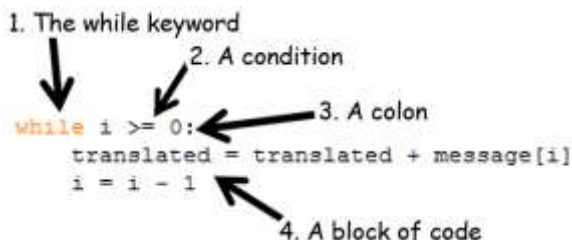


Figure 5-2. The parts of a `while` loop statement.

To understand `while` loops, we will 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 values are case-sensitive (you always need to capitalize the T and F, and leave the rest in lowercase). They are not string values. You do not put a ' quote character around `True` or `False`. We will use Boolean values (also called **bools**) with comparison operators to form conditions. (Explained later after Comparison Operators.)

Like a value of any other data type, bools can be stored in variables. Type this into the interactive shell:

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

Comparison Operators

In line 8 of our program, look at the expression after the `while` keyword:

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

reverseCipher.py

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 an operator (the `>=` sign, called the “greater than or equal” operator). The `>=` operator is called a **comparison operator**.

The comparison operator is used to compare two values and evaluate to a `True` or `False` Boolean value. Table 5-1 lists the comparison operators.

Table 5-1. 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`. 50 is not less than 10.5, so `50 < 10.5` is `False`. 10.5 is less than 11.3, so `10.5 < 11.3` evaluates to `True`.

Look again at `10 < 10`. It is `False` because the number 10 is not smaller than the number 10. They are exactly the same size. If Alice were the same height as Bob, you wouldn't say that Alice is shorter than Bob. That statement would be false.

Try typing in some expressions using the other comparison operators:

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



```
>>>
```

Remember that for the “less than or equal to” and “greater than or equal to” operators, the < or > sign always comes **before** the = sign.

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

Notice the difference between the assignment operator (=) and the “equal to” comparison operator (==). The equal (=) sign is used to assign a value to a variable, and the equal to (==) sign is used in expressions to see whether two values are the same. If you’re asking Python if two things are equal, use ==. If you are telling Python to set a variable to a value, use =.

String and integer values will always be not-equal to each other. For example, try entering the following into the interactive shell:

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


On line 5, the amount of indentation has decreased to 4, so we know that the block on the previous line has ended. Line 4 is the only line in that block. Since line 5 has the same amount of indentation as the block from line 3, we know that the block has continue on to line 5.

Line 6 is a blank line, so we just skip it.

Line 7 has four spaces on 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 has ended.

There are two blocks in the above make-believe code. The first block goes from line 2 to line 7. The second block is just made up of line 4 (and is inside the other block).

(As a side note, it doesn't always have to be four spaces. The blocks can use any number of spaces, but the convention is to use four spaces.)

The `while` Loop Statement

```

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

Let's look at the `while` statement on line 8 again. What a `while` statement tells Python to do is first check to see what the condition (which on line 8 is `i >= 0`) evaluates to. If the condition evaluates to `True`, then the program execution enters the block following the `while` statement. From looking at the indentation, this block is made up of lines 9 and 10.

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

If the condition was `True`, the program execution starts at the top of the block and executes each line in turn going down. 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 is still `True`, the execution jumps into the block again. If it is `False`, the program execution will skip past it.

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".

“Growing” a String

Remember on line 7 that the `i` variable is first set to the length of the `message` minus one, and the `while` loop on line 8 will keep executing the lines inside the following block until the condition `i >= 0` is `False`.

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

There are two lines inside the `while` statement’s block, line 9 and line 10.

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`. In this way, the string value stored in `translated` “grows” until it becomes the fully encrypted string.

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

The next line is line 12, but since this line has less indentation, Python knows that the `while` statement’s block has ended. So rather than go 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`, then 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. The variable `i` starts off with the value of the last index of `message` and the `translated` variable starts off as the blank string. Then inside the loop, the value of `message[i]` (which is the last character in the `message` string, since `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. This means 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

what causes `translated` to hold the reverse of the string in the message. When `i` is finally set to `-1`, then the `while` loop's condition will be `False` and the execution jumps to line 12.

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

reverseCipher.py

At the end of our program on line 12, we print out 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 will show the user what the reversed string looks like.

If you are still having trouble understanding how the code in the `while` loop reverses the string, try adding this new line inside the `while` loop:

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

reverseCipher.py

This will print out the three expressions `i`, `message[i]`, and `translated` each time the execution goes through the loop (that is, on each **iteration** of the loop). The commas tell the `print()` function that we are printing three separate things, so the function will add a space in between them. Now when you run the program, you can see how the `translated` variable “grows”. The output will look like this:

```
48 . .
47 d .d
46 a .da
45 e .dae
44 d .daed
43  .daed
42 e .daed e
41 r .daed er
40 a .daed era
39  .daed era
38 m .daed era m
37 e .daed era me
36 h .daed era meh
35 t .daed era meht
34  .daed era meht
33 f .daed era meht f
32 o .daed era meht fo
31  .daed era meht fo
```

```

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

```

The first line, which shows “48 . .”, is showing 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, and 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 `print()` call displays “47 . .d”. You can see that `i` has been decremented from 48 to 47, and 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 that `translated` is now set to `' .d'`.

Now you can see how the `translated` variable’s string is slowly “grown” from a blank string to the reverse of the string stored in `message`.

Tracing Through the Program, Step by Step

The previous explanations have gone through what each line does, but let's go step by step through the program the same way the Python interpreter does. The interpreter starts at the very top, executes the first line, then moves down a line to execute the next instruction. The blank lines and comments are skipped. The `while` loop will cause the program execution will loop back to the start of the loop after it finishes.

Here is a brief explanation of each line of code in the same order that the Python interpreter executes it. Follow along with to see how the execution moves down the lines of the program, but sometimes jumps back to a previous line.

```

1. # Reverse Cipher
2. # http://inventwithpython.com/hacking (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)
reverseCipher.py
```

Step 1	Line 1	This is a comment, so the Python interpreter skips it.
Step 2	Line 2	This is a comment, and skipped.
Step 3	Line 4	The string value 'Three can keep a secret, if two of them are dead.' is stored in the <code>message</code> variable.
Step 4	Line 5	The blank string '' is stored in the <code>translated</code> variable.
Step 5	Line 7	<code>len(message) - 1</code> evaluates to 48. The integer 48 is stored in the <code>i</code> variable.
Step 6	Line 8	The <code>while</code> loop's condition <code>i >= 0</code> evaluates to <code>True</code> . Since the condition is <code>True</code> , the program execution moves inside the following block.
Step 7	Line 9	<code>translated + message[i]</code> to <code>.'. '</code> . The string value <code>.'. '</code> is stored in the <code>translated</code> variable.
Step 8	Line 10	<code>i - 1</code> evaluates to 47. The integer 47 is stored in the <code>i</code> variable.
Step 9	Line 8	When the program execution reaches the end of the block, the execution moves back to the <code>while</code> statement and rechecks the condition. <code>i >= 0</code>

		evaluates to <code>True</code> , the program execution moves inside the block again.
Step 10	Line 9	<code>translated + message[i]</code> evaluates to <code>' .d'</code> . The string value <code>' .d'</code> is stored in the <code>translated</code> variable.
Step 11	Line 10	<code>i - 1</code> evaluates to 46. The integer 46 is stored in the <code>i</code> variable.
Step 12	Line 8	The <code>while</code> statement rechecks the condition. Since <code>i >= 0</code> evaluates to <code>True</code> , the program execution will move inside the block again.
Step 13 to Step 149		...The lines of the code continue to loop. We fast-forward to when <code>i</code> is set to 0 and <code>translated</code> is set to <code>' .daed era meht fo owt fi ,terces a peek nac eerh'...</code>
Step 150	Line 8	The <code>while</code> loop's condition is checked, and <code>0 >= 0</code> evaluates to <code>True</code> .
Step 151	Line 9	<code>translated + message[i]</code> evaluates to <code>' .daed era meht fo owt fi ,terces a peek nac eerhT'</code> . This string is stored in the <code>translated</code> variable.
Step 152	Line 10	<code>i - 1</code> evaluates to <code>0 - 1</code> , which evaluates to <code>-1</code> . <code>-1</code> is stored in the <code>i</code> variable.
Step 153	Line 8	The <code>while</code> loop's condition is <code>i >= 0</code> , which evaluates to <code>-1 >= 0</code> , which evaluates to <code>False</code> . Because the condition is now <code>False</code> , the program execution skips the following block of code and goes to line 12.
Step 154	Line 12	<code>translated</code> evaluates to the string value <code>' .daed era meht fo owt fi ,terces a peek nac eerhT'</code> . The <code>print()</code> function is called and this string is passed, making it appear on the screen.
		There are no more lines after line 12, so the program terminates.

Using `input()` In Our Programs

The programs in this book are all designed so that the strings that are being encrypted or decrypted are typed directly into the source code. You could also modify the assignment statements so that they call the `input()` function. You can pass a string to the `input()` function to appear as a prompt for the user to type in the string to encrypt. For example, if you change line 4 in *reverseCipher.py* to this:

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

reverseCipher.py

Then when you run the program, it will print the prompt to the screen and wait for the user to type in the message and press Enter. The message that the user types in will be the string value that is stored in the `message` variable:

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


Practice Exercises, Chapter 5, Section A

Practice exercises can be found at <http://invpy.com/hackingpractice5A>.

Summary

Now that we have learned how to deal with text, we 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 will communicate with each other.

Strings are just a different data type that we can use in our programs. We can use the `+` operator to concatenate strings together. We can use indexing and slicing to create a new string from part of a different string. The `len()` function takes a string argument and returns an integer of how many characters are in the string.

The Boolean data type has only two values: `True` and `False`. Comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=` can compare two values and evaluate to a Boolean value.

Conditions are expression that are used in several different kinds of statements. A `while` loop statement keeps executing the lines inside the block that follows it as long as its condition evaluates to `True`. A block is made up of lines with the same level of indentation, including any blocks inside of them.

A common practice in programs is to start a variable with a blank string, and then concatenate characters to it until it “grows” into the final desired string.



THE CAESAR CIPHER

Topics Covered In This Chapter:

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

“BIG BROTHER IS WATCHING YOU.”

“1984” by George Orwell

Implementing a Program

In Chapter 1, we used a cipher wheel, a St. Cyr slide, and a chart of letters and numbers to implement the Caesar cipher. In this chapter, we will use a computer program to implement the Caesar cipher.

The reverse cipher always encrypts the same way. But the Caesar cipher uses keys, which encrypt the message in a different way depending on which key is used. The keys for the Caesar cipher

are the integers from 0 to 25. Even if a cryptanalyst knows that the Caesar cipher was used, that alone does not give her enough information to break the cipher. She must also know the key.

Source Code of the Caesar Cipher Program

Type in the following code into the file editor, and then save it as *caesarCipher.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory (that is, folder) as the *caesarCipher.py* file. You can download this file from <http://invy.com/pyperclip.py>

Source code for caesarCipher.py

```

1. # Caesar Cipher
2. # http://inventwithpython.com/hacking (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. # tells the program to encrypt or decrypt
13. mode = 'encrypt' # set to 'encrypt' or 'decrypt'
14.
15. # every possible symbol that can be encrypted
16. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
17.
18. # stores the encrypted/decrypted form of the message
19. translated = ''
20.
21. # capitalize the string in message
22. message = message.upper()
23.
24. # run the encryption/decryption code on each symbol in the message string
25. for symbol in message:
26.     if symbol in LETTERS:
27.         # get the encrypted (or decrypted) number for this symbol
28.         num = LETTERS.find(symbol) # get the number of the symbol
29.         if mode == 'encrypt':
30.             num = num + key
31.         elif mode == 'decrypt':
32.             num = num - key
33.
34.         # handle the wrap-around if num is larger than the length of

```

```

35.         # LETTERS or less than 0
36.         if num >= len(LETTERS):
37.             num = num - len(LETTERS)
38.         elif num < 0:
39.             num = num + len(LETTERS)
40.
41.         # add encrypted/decrypted number's symbol at the end of translated
42.         translated = translated + LETTERS[num]
43.
44.     else:
45.         # just add the symbol without encrypting/decrypting
46.         translated = translated + symbol
47.
48. # print the encrypted/decrypted string to the screen
49. print(translated)
50.
51. # copy the encrypted/decrypted string to the clipboard
52. pyperclip.copy(translated)

```

Sample Run of the Caesar Cipher Program

When you run this program, the output will look like this:

```
GUVF VF ZL FRPERG ZRFFNTR.
```

The above text is the string 'This is my secret message.' encrypted with the Caesar cipher with key 13. The Caesar cipher program you just ran will automatically copy this encrypted string to the clipboard so you can paste it in an email or text file. This way you can easily take the encrypted output from the program and send it to another person.

To decrypt, just paste this 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`:

```

caesarCipher.py
6. # the string to be encrypted/decrypted
7. message = 'GUVF VF ZL FRPERG ZRFFNTR.'
8.
9. # the encryption/decryption key
10. key = 13
11.
12. # tells the program to encrypt or decrypt
13. mode = 'decrypt' # set to 'encrypt' or 'decrypt'

```

When you run the program now, the output will look like this:

```
THIS IS MY SECRET MESSAGE.
```

If you see this error message when running the program:

```
Traceback (most recent call last):
  File "C:\Python32\caesarCipher.py", line 4, in <module>
    import pyperclip
ImportError: No module named pyperclip
```

...then you have not downloaded the `pyperclip` module into the right folder. If you still cannot get the module working, just delete lines 4 and 52 (which have the text “`pyperclip`” in them) from the program. This will get rid of the code that depends on the `pyperclip` module.

Checking Your Source Code with the Online Diff Tool

To compare the code you typed to the code that is in this book, you can use the online diff tool on this book’s website. Open <http://invpy.com/hackingdiff> in your web browser. Copy and paste your code into the text field on this web page, and then click the Compare button. The diff tool will show any differences between your code and the code in this book. This can help you find any typos you made when typing out the program.

Practice Exercises, Chapter 6, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice6A>.

How the Program Works

Let’s go over exactly what each of the lines of code in this program does.

Importing Modules with the `import` Statement

```

1. # Caesar Cipher
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import pyperclip
caesarCipher.py
```

Line 4 is a new kind of statement called an `import` statement. While Python includes many built-in functions, some functions exist in separate programs called modules. **Modules** are Python programs that contain additional functions that can be used by your program. In this case, we’re importing a module named `pyperclip` so that we can call the `pyperclip.copy()` function later in this program.

The `import` statement is made up of the `import` keyword followed by the module name. Line 4 is an `import` statement that imports the `pyperclip` module, which contains several functions related to copying and pasting text to the clipboard.

```
caesarCipher.py
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. # tells the program to encrypt or decrypt
13. mode = 'encrypt' # set to 'encrypt' or 'decrypt'
```

The next few lines set three variables: `message` will store the string to be encrypted or decrypted, `key` will store the integer of the encryption key, and `mode` will store either the string `'encrypt'` (which will cause code later in the program to encrypt the string in `message`) or `'decrypt'` (which will tell the program to decrypt rather than encrypting).

Constants

```
caesarCipher.py
15. # every possible symbol that can be encrypted
16. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

We also need a string that contains all the capital letters of the alphabet in order. It would be tiring to type the full `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` string value each time we use it in the program (and we might make typos when typing it, which would cause errors in our program). So instead we will type the code for the string value once and place it in a variable named `LETTERS`. This string contains all the letters that our cipher program can possibly encrypt. This set of letters (which don't have to be just letters but can also be numbers, punctuation, or any other symbol) is called the cipher's **symbol set**. The end of this chapter will tell you how to expand this program's symbol set to include other characters besides letters.

The `LETTERS` variable name is in all capitals. This is the programming convention for constant variables. **Constants** are variables whose values are not meant to be changed when the program runs. Although we *can* change `LETTERS` just like any other variable, the all-caps reminds the programmer to not write code that does so.

Like all conventions, we don't *have* to follow it. But doing it this way makes it easier for other programmers to understand how these variables are used. (It even can help you if you are looking at code you wrote yourself a long time ago.)

The `upper()` and `lower()` String Methods

```

caesarCipher.py
18. # stores the encrypted/decrypted form of the message
19. translated = ''
20.
21. # capitalize the string in message
22. message = message.upper()

```

On line 19, the program stores a blank string in a variable named `translated`. Just like in the reverse cipher from last chapter, 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.

Line 22 is an assignment statement that stores a value in a variable named `message`, but the expression on the right side of the `=` operator is something we haven't seen before:

```
message.upper()
```

This is a method call. **Methods** are just like functions, except they are attached to a non-module value (or in the case of line 22, a variable containing a value) with a period. The name of this method is `upper()`, and it is being called on the string value stored in the `message` variable.

A function is not a method just because it is in a module. You will see on line 52 that we call `pyperclip.copy()`, but `pyperclip` is a module that was imported on line 4, so `copy()` is not a method. It is just a function that is inside the `pyperclip` module. If this is confusing, then you can always call methods and functions a “function” and people will know what you're talking about.

Most data types (such as strings) have methods. Strings have a method called `upper()` and `lower()` which will evaluate to an uppercase or lowercase version of that string, respectively.

Try typing the following into the interactive shell:

```

>>> 'Hello world!'.upper()
'HELLO WORLD!'
>>> 'Hello world!'.lower()
'hello world!'
>>>

```

Because the `upper()` method returns a string value, you can call a method on *that* string as well. Try typing `'Hello world!'.upper().lower()` into the shell:

```
>>> 'Hello world!'.upper().lower()
'hello world!'
>>>
```

`'Hello world!'.upper()` evaluates to the string `'HELLO WORLD!'`, and then we call the `lower()` method on *that* string. This returns the string `'hello world!'`, which is the final value in the evaluation. The order is important. `'Hello world!'.lower().upper()` is not the same as `'Hello world!'.upper().lower()`:

```
>>> 'Hello world!'.lower().upper()
'HELLO WORLD!'
>>>
```

If a string is stored in a variable, you can call any string method (such as `upper()` or `lower()`) on that variable. Look at this example:

```
>>> fizz = 'Hello world!'
>>> fizz.upper()
'HELLO WORLD!'
>>> fizz
'Hello world!'
```

Calling the `upper()` or `lower()` method on a string value in a variable does not change the value inside a variable. Methods are just part of expressions that evaluate to a value. (Think about it like this: the expression `fizz + 'ABC'` would not change the string stored in `fizz` to have `'ABC'` concatenated to the end of it, unless we used it in an assignment statement like `fizz = fizz + 'ABC'`.)

Different data types have different methods. You will learn about other methods as you read this book. A list of common string methods is at <http://invy.com/stringmethods>.

The `for` Loop Statement

```
caesarCipher.py
24. # run the encryption/decryption code on each symbol in the message string
25. for symbol in message:
```


The `for` loop is very good at looping over a string or list of values (we will learn about lists later). This is different from the `while` loop, which loops as long as a certain condition is `True`. A `for` statement has six parts:

1. The `for` keyword.
2. A variable name.
3. The `in` keyword.
4. A string value (or a variable containing a string value).
5. A colon.
6. A block of code.

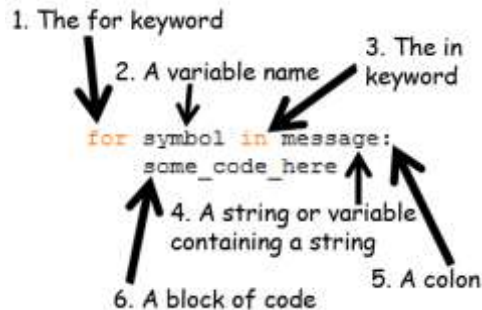


Figure 6-1. The parts of a `for` loop statement.

Each time the program execution goes through the loop (that is, on each iteration through the loop) the variable in the `for` statement takes on the value of the next character in the string.

For example, type the following into the interactive shell. Note that after you type the first line, the `>>>` prompt will turn into `. . .` (although in IDLE, it will just print three spaces) 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
>>>
```

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 much less code to type. You can make a `while` loop that acts the same way as a `for` loop by adding a little extra code:

```
>>> i = 0
>>> while i < len('Howdy'):
...     letter = 'Howdy'[i]
```

```

...     print('The letter is ' + letter)
...     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 does the exact same thing that the `for` loop does, but is not as short and simple as the `for` loop.

Before we can understand lines 26 to 32 of the Caesar cipher program, we need to first learn about the `if`, `elif`, and `else` statements, the `in` and `not in` operators, and the `find()` string method.

Practice Exercises, Chapter 6, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice6B>.

The `if` Statement

An `if` statement can be read as “If this condition is `True`, execute the code in the following block. Otherwise if it is `False`, skip the block.” Open the file editor and type in the following small program. Then save the file as `password.py` and press **F5** to run it.

Source code for `password.py`

```

1. print('What is the password?')
2. password = input()
3. if password == 'rosebud':
4.     print('Access granted.')
5. if password != 'rosebud':
6.     print('Access denied.')
7. print('Done.')

```

When the `password = input()` line is executed, the user can type in anything she wants and it will be stored as a string in the variable `password`. If she typed in “rosebud” (in all lowercase letters), then the expression `password == 'rosebud'` will evaluate to `True` and the program execution will enter the following block to print the `'Access granted.'` string.

If `password == 'rosebud'` is `False`, then this block of code is skipped. Next, the second `if` statement will have its condition also evaluated. If this condition, `password !=`

'rosebud' is `True`, then the execution jumps inside of the following block to print out 'Access denied.'. If the condition is `False`, then this block of code is skipped.

The `else` Statement

Often we want to test a condition and execute one block of code if it is `True` and another block of code if it is `False`. The previous *password.py* example is like this, but it used two `if` statements.

An `else` statement can be used after an `if` statement's block, and its block of code will be executed if the `if` statement's condition is `False`. You can read the code as “if this condition is true, execute this block, or else execute this block.”

Type in the following program and save it as *password2.py*. Notice that it does the same thing as the previous *password.py* program, except it uses an `if` and `else` statement instead of two `if` statements:

Source code for *password2.py*

```
1. print('What is the password?')
2. password = input()
3. if password == 'rosebud':
4.     print('Access granted.')
5. else:
6.     print('Access denied.')
7. print('Done.')
```

The `elif` Statement

There is also an “else if” statement called the `elif` statement. 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 was `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.” Type in this example program into the file editor and save it as *elifeggs.py*:

Source code for *elifeggs.py*

```
1. numberOfEggs = 12
2. if numberOfEggs < 4:
3.     print('That is not that many eggs.')
4. elif numberOfEggs < 20:
5.     print('You have quite a few eggs.')
6. elif numberOfEggs == 144:
```

```
7.     print('You have a lot of eggs. Gross!')
8. else:
9.     print('Eat ALL the eggs!')
```

When you run this program, the integer 12 is stored in the variable `numberOfEggs`. Then the condition `numberOfEggs < 4` is checked to see if it is `True`. If it isn't, the execution skips the block and checks `numberOfEggs < 20`. If it isn't `True`, execution skips that block and checks if `numberOfEggs == 144`. If all of these conditions have been `False`, then the `else` block is executed.

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 `else` statements, and the `else` statement always comes last.

The `in` and `not in` Operators

An expression of two strings connected by the `in` operator will evaluate to `True` if the first string is inside the second string. Otherwise the expression evaluates to `False`. Notice that the `in` and `not in` operators are case-sensitive. Try typing the following in the interactive shell:

```
>>> 'hello' in 'hello world!'
True
>>> 'ello' in 'hello world!'
True
>>> 'HELLO' in 'hello world!'
False
>>> 'HELLO' in 'HELLO world!'
True
>>> '' in 'Hello'
True
>>> '' in ''
True
>>> 'D' in 'ABCDEF'
True
>>>
```

The `not in` operator will evaluate to the opposite of `in`. Try typing the following into the interactive shell:

```
>>> 'hello' not in 'hello world!'
False
>>> 'ello' not in 'hello world!'
False
```

```

>>> 'HELLO' not in 'hello world!'
True
>>> 'HELLO' not in 'HELLO world!'
False
>>> '' not in 'Hello'
False
>>> '' not in ''
False
>>> 'D' not in 'ABCDEF'
False
>>>

```

Expressions using the `in` and `not in` operators are handy for conditions of `if` statements so that we can execute some code if a string exists inside of another string.

Also, the `in` keyword used in `for` statements is not the same as the `in` operator used here. They are just typed the same.

The `find()` String Method

Just like the `upper()` method can be called on a string values, the `find()` method is a string method. The `find()` method takes one string argument and returns the integer index of where that string appears in the method's string. Try typing the following into the interactive shell:

```

>>> 'hello'.find('e')
1
>>> 'hello'.find('o')
4
>>> fizz = 'hello'
>>> fizz.find('h')
0
>>>

```

If the string argument cannot be found, the `find()` method returns the integer `-1`. Notice that the `find()` method is case-sensitive. Try typing the following into the interactive shell:

```

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

```

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. Try typing 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 if a string exists in another string, but also tells you where.

Practice Exercises, Chapter 6, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice6C>.

Back to the Code

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

```
caesarCipher.py
26.     if symbol in LETTERS:
27.         # get the encrypted (or decrypted) number for this symbol
28.         num = LETTERS.find(symbol) # get the number of the symbol
```

If the string in `symbol` (which the `for` statement has set to be only a single character) is a capital letter, then the condition `symbol in LETTERS` will be `True`. (Remember that on line 22 we converted `message` to an uppercase version with `message = message.upper()`, so `symbol` cannot possibly be a lowercase letter.) The only time the condition is `False` is if `symbol` is something like a punctuation mark or number string value, such as `'?'` or `'4'`.

We want to check if `symbol` is an uppercase letter because our program will only encrypt (or decrypt) uppercase letters. Any other character will be added to the `translated` string without being encrypted (or decrypted).

There is a new block that starts after the `if` statement on line 26. If you look down the program, you will notice that this block stretches all the way to line 42. The `else` statement on line 44 is paired to the `if` statement on line 26.

```

caesarCipher.py
29.         if mode == 'encrypt':
30.             num = num + key
31.         elif mode == 'decrypt':
32.             num = num - key

```

Now that we have the current symbol’s number stored in `num`, we can do the encryption or decryption math on it. The Caesar cipher adds the key number to the letter’s number to encrypt it, or subtracts the key number from the letter’s number to decrypt it.

The `mode` variable contains a string that tells the program whether or not it should be encrypting or decrypting. If this string is `'encrypt'`, then the condition for line 29’s `if` statement will be `True` and line 30 will be executed (and the block after the `elif` statement will be skipped). If this string is any other value besides `'encrypt'`, then the condition for line 29’s `if` statement is `False` and the program execution moves on to check the `elif` statement’s condition.

This is how our program knows when to encrypt (where it is adding the key) or decrypt (where it is subtracting the key). If the programmer made an error and stored `'pineapples'` in the `mode` variable on line 13, then both of the conditions on lines 29 and 31 would be `False` and nothing would happen to the value stored in `num`. (You can try this yourself by changing line 13 and re-running the program.)

```

caesarCipher.py
34.         # handle the wrap-around if num is larger than the length of
35.         # LETTERS or less than 0
36.         if num >= len(LETTERS):
37.             num = num - len(LETTERS)
38.         elif num < 0:
39.             num = num + len(LETTERS)

```

Remember that when we were implementing the Caesar cipher with paper and pencil, sometimes the number after adding or subtracting the key would be greater than or equal to 26 or less than 0. In those cases, we had to add or subtract 26 to the number to “wrap-around” the number. This “wrap-around” is what lines 36 to 39 do for our program.

If `num` is greater than or equal to 26, then the condition on line 36 is `True` and line 37 is executed (and the `elif` statement on line 38 is skipped). Otherwise, Python will check if `num` is less than 0. If that condition is `True`, then line 39 is executed.

The Caesar cipher adds or subtracts 26 because that is the number of letters in the alphabet. If English only had 25 letters, then the “wrap-around” would be done by adding or subtracting 25.

Notice that instead of using the integer value 26 directly, we use `len(LETTERS)`. The function call `len(LETTERS)` will return the integer value 26, so this code works just as well. But the reason that we use `len(LETTERS)` instead of 26 is that the code will work no matter what characters we have in `LETTERS`.

We can modify the value stored in `LETTERS` so that we encrypt and decrypt more than just the uppercase letters. How this is done will be explained at the end of this chapter.

```
caesarCipher.py
41.         # add encrypted/decrypted number's symbol at the end of translated
42.         translated = translated + LETTERS[num]
```

Now that the integer in `num` has been modified, it will be the index of the encrypted (or decrypted) letter in `LETTERS`. We want to add this encrypted/decrypted letter to the end of the `translated` string, so line 42 uses string concatenation to add it to the end of the current value of `translated`.

```
caesarCipher.py
44.     else:
45.         # just add the symbol without encrypting/decrypting
46.         translated = translated + symbol
```

Line 44 has four spaces of indentation. If you look at the indentation of the lines above, you’ll see that this means it comes after the `if` statement on line 26. There’s a lot of code in between this `if` and `else` statement, but it all belongs in the block of code that follows the `if` statement on line 26. If that `if` statement’s condition was `False`, then the block would have been skipped and the program execution would enter the `else` statement’s block starting at line 46. (Line 45 is skipped because it is a comment.)

This block has just one line in it. It adds the `symbol` string as it is to the end of `translated`. This is how non-letter strings like `' '` or `'.'` are added to the translated string without being encrypted or decrypted.

Displaying and Copying the Encrypted/Decrypted String

```

caesarCipher.py
48. # print the encrypted/decrypted string to the screen
49. print(translated)
50.
51. # copy the encrypted/decrypted string to the clipboard
52. pyperclip.copy(translated)

```

Line 49 has no indentation, which means it is the first line after the block that started on line 26 (the `for` loop's block). By the time the program execution reaches line 49, it has looped through each character in the `message` string, encrypted (or decrypted) the characters, and added them to `translated`.

Line 49 will call 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 wrap-around, 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 52 calls a function that is inside the `pyperclip` module. The function's name is `copy()` and it takes one string argument. Because `copy()` is a function in the `pyperclip` module, we have to 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.

You can see this error message for yourself by typing this code in the interactive shell:

```

>>> copy('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'copy' is not defined
>>>

```

Also, if you forget the `import pyperclip` line before trying to call `pyperclip.copy()`, Python will give an error message. Try typing this into the interactive shell:

```

>>> pyperclip.copy('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pyperclip' is not defined
>>>

```

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 type in a very, very long string for the value to store in the `message` variable, your computer can encrypt or decrypt a message within a second or two. Compare this to the several minutes it would take to do this with a cipher wheel or St. Cyr slide. The program even copies the encrypted text to the clipboard so the user can simply paste it into an email to send to someone.

Encrypt Non-Letter Characters

One problem with the Caesar cipher that we've implemented is that it cannot encrypt non-letters. For example, if you encrypt the string `'The password is 31337.'` with the key 20, it will encrypt to `'Dro zkccgybn sc 31337.'` This encrypted message doesn't keep the password in the message very secret. However, we can modify the program to encrypt other characters besides letters.

If you change the string that is stored in `LETTERS` to include more than just the uppercase letters, then the program will encrypt them as well. This is because on line 26, the condition `symbol` in `LETTERS` will be `True`. The value of `num` will be the index of `symbol` in this new, larger `LETTERS` constant variable. The "wrap-around" will need to add or subtract the number of characters in this new string, but that's already handled because we use `len(LETTERS)` instead of typing in 26 directly into the code. (This is why we programmed it this way.)

The only changes you have to make are to the `LETTERS` assignment statement on line 16 and commenting out line 22 which capitalizes all the letters in `message`.

```

caesarCipher.py
15. # every possible symbol that can be encrypted
16. LETTERS = ' !"#%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\ ]
   ^_`a bcdefghijklmnopqrstuvwxyz{|}~'
17.
18. # stores the encrypted/decrypted form of the message
19. translated = ''
20.
21. # capitalize the string in message
22. #message = message.upper()

```

Notice that this new string has the escape characters `\'` and `\\` in it. You can download this new version of the program from <http://invpy.com/caesarCipher2.py>.

This modification to our program is like if we had a cipher wheel or St. Cyr slide that had not only uppercase letters but numbers, punctuation, and lowercase letters on it as well.

Even though the value for `LETTERS` has to be the same when running the program for decryption as when it encrypted the message, this value doesn't have to be secret. Only the key needs to be kept secret, while the rest of program (including the code for the Caesar cipher program) can be shared with the world.

Summary

You've had to learn 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 importantly, you can understand how this code works.

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

Constant variables are by convention written in `UPPERCASE`. These variables are not meant to have their value changed (although nothing prevents the programmer from writing code that does this). 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 `upper()` and `lower()` string methods return an uppercase or lowercase version of the string they are called on. The `find()` string method returns an integer of where the string argument passed to it can be found in the string it is called on.

A `for` loop will iterate over all the characters in string value, setting a variable to each character on each iteration. The `if`, `elif`, and `else` statements can execute blocks of code based on whether a condition is `True` or `False`.

The `in` and `not in` operators can check if one string is or isn't in another string, and evaluates to `True` or `False` accordingly.

Knowing how to program gives you the power to take a process like the Caesar cipher and put it down in a language that a computer can understand. And once the computer understands how to do it, it can do it much faster than any human can and with no mistakes (unless there are mistakes in your programming.) This is an incredibly useful skill, but it turns out the Caesar cipher can easily be broken by someone who knows computer programming. In the next chapter we will use our skills to write a Caesar cipher "hacker" so we can read ciphertext that other people encrypted. So let's move on to the next chapter, and learn how to hack encryption.



HACKING THE CAESAR CIPHER WITH THE BRUTE-FORCE TECHNIQUE

Topics Covered In This Chapter:

- Kerckhoffs’s Principle and Shannon’s Maxim
- The brute-force technique
- The `range()` function
- String formatting (string interpolation)

Hacking Ciphers

We can hack the Caesar cipher by using a cryptanalytic technique called “brute-force”. Because our code breaking program is so effective against the Caesar cipher, you shouldn’t use it to encrypt your secret information.

Ideally, the ciphertext would never fall into anyone’s hands. But **Kerckhoffs’s Principle** (named after the 19th-century cryptographer Auguste Kerckhoffs) says that a cipher should still be secure even if everyone else knows how the cipher works and has the ciphertext (that is, everything except the key). This was restated by the 20th century mathematician Claude Shannon as **Shannon’s Maxim**: “The enemy knows the system.”



Figure 7-1. Auguste Kerckhoffs
January 19, 1835 - August 9, 1903

“A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.”



Figure 7-2. Claude Shannon
April 30, 1916 - February 24, 2001

“The enemy knows the system.”

The Brute-Force Attack

Nothing stops a cryptanalyst from guessing one key, decrypting the ciphertext with that key, looking at the output, and if it was not the correct key then moving on to the next key. The technique of trying every possible decryption key is called a **brute-force attack**. It isn't a very sophisticated hack, but through sheer effort (which the computer will do for us) the Caesar cipher can be broken.

Source Code of the Caesar Cipher Hacker Program

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *caesarHacker.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *caesarHacker.py* file. You can download this file from <http://invpy.com/pyperclip.py>.

Source code for caesarHacker.py

```
1. # Caesar Cipher Hacker
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. message = 'GUVF VF ZL FRPERG ZRFFNTR.'
```

```

5. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
6.
7. # loop through every possible key
8. for key in range(len(LETTERS)):
9.
10.     # It is important to set translated to the blank string so that the
11.     # previous iteration's value for translated is cleared.
12.     translated = ''
13.
14.     # The rest of the program is the same as the original Caesar program:
15.
16.     # run the encryption/decryption code on each symbol in the message
17.     for symbol in message:
18.         if symbol in LETTERS:
19.             num = LETTERS.find(symbol) # get the number of the symbol
20.             num = num - key
21.
22.             # handle the wrap-around if num is 26 or larger or less than 0
23.             if num < 0:
24.                 num = num + len(LETTERS)
25.
26.             # add number's symbol at the end of translated
27.             translated = translated + LETTERS[num]
28.
29.         else:
30.             # just add the symbol without encrypting/decrypting
31.             translated = translated + symbol
32.
33.     # display the current key being tested, along with its decryption
34.     print('Key #%s: %s' % (key, translated))

```

You will see 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 does the same steps to decrypt the key.

Sample Run of the Caesar Cipher Hacker Program

Here is what the Caesar cipher program looks like when you run it. It is trying to break the ciphertext, “GUVF VF ZL FRPERG ZRFFNTR.” Notice that the decrypted output for key 13 is plain English, so the original encryption key must have been 13.

```

Key #0: GUVF VF ZL FRPERG ZRFFNTR.
Key #1: FTUE UE YK EQODQF YQEEMSQ.
Key #2: ESTD TD XJ DPNCPE XPDDLRLP.
Key #3: DRSC SC WI COMBOD WOCCKQO.
Key #4: CQRB RB VH BNLANC VNBBJPN.

```

```

Key #5: BPQA QA UG AMKZMB UMAAIOM.
Key #6: AOPZ PZ TF ZLJYLA TLZZHNL.
Key #7: ZNOY OY SE YKIXKZ SKYYGMK.
Key #8: YMNX NX RD XJHWJY RJXXFLJ.
Key #9: XLMW MW QC WIGVIX QIWWEKI.
Key #10: WKLW LV PB VHFUHW PHVVDJH.
Key #11: VJKU KU OA UGETGV OGUUCIG.
Key #12: UIJT JT NZ TFDSFU NFTTBHF.
Key #13: THIS IS MY SECRET MESSAGE.
Key #14: SGHR HR LX RDBQDS LDRRZFD.
Key #15: RFGQ GQ KW QCAPCR KCQQYEC.
Key #16: QEFP FP JV PBZOBQ JBPPXDB.
Key #17: PDEO EO IU OAYNAP IA00WCA.
Key #18: OCDN DN HT NZXMZO HZNNVBZ.
Key #19: NBCM CM GS MYWLYN GYMMUAY.
Key #20: MABL BL FR LXVKXM FXLLTZX.
Key #21: LZAK AK EQ KWUJWL EWKKSYP.
Key #22: KYZJ ZJ DP JVTIVK DVJJRXV.
Key #23: JXYI YI CO IUSHUJ CUIIQWU.
Key #24: IWXH XH BN HTRGTI BTHHPVT.
Key #25: HVWG WG AM GSQFSH ASGGOUS.

```

How the Program Works

```

caesarHacker.py
1. # Caesar Cipher Hacker
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. message = 'GUVF VF ZL FRPERG ZRFFNTR.'
5. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

```

The hacker program will create a `message` variable that stores the ciphertext string the program tries to decrypt. The `LETTERS` constant variable contains every character that can be encrypted with the cipher. The value for `LETTERS` needs to be exactly the same as the value for `LETTERS` used in the Caesar cipher program that encrypted the ciphertext we are trying to hack, otherwise the hacker program won't work.

The `range()` Function

```

caesarHacker.py
7. # loop through every possible key
8. for key in range(len(LETTERS)):

```

Line 8 is a `for` loop that does not iterate over a string value, but instead iterates over the return value from a call to a function named `range()`. The `range()` function takes one integer argument and returns a value of the range data type. These range values can be used in `for` loops to loop a specific number of times. Try typing the following into the interactive shell:

```
>>> for i in range(4):
...     print('Hello')
...
Hello
Hello
Hello
Hello
>>>
```

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

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
>>>
```

Line 8 is a `for` loop that will set the `key` variable with the values 0 up to (but not including) 26. Instead of hard-coding the value 26 directly into our program, we use the return value from `len(LETTERS)` so that if we modify `LETTERS` the program will still work. See the “Encrypt Non-Letter Characters” section in the last chapter to read why.

So the first time the program execution goes through this loop, `key` will be set to 0 and the ciphertext in `message` will be decrypted with key 0. (The code inside the `for` loop does the decrypting.) On the next iteration of line 8's `for` loop, `key` will be set to 1 for the decryption.

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 `range()` call evaluates to a value of the “range object” data type.

Back to the Code

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

On line 12, `translated` is set to the blank string. The decryption code on the next few lines adds the decrypted text to the end of the string in `translated`. It is important that we reset `translated` to the blank string at the beginning of this `for` loop, otherwise the decrypted text will be added to the decrypted text in `translated` from the last iteration in the loop.

```
caesarHacker.py
14.     # The rest of the program is the same as the original Caesar program:
15.
16.     # run the encryption/decryption code on each symbol in the message
17.     for symbol in message:
18.         if symbol in LETTERS:
19.             num = LETTERS.find(symbol) # get the number of the symbol
```

Lines 17 to 31 are almost exactly the same as the code in the Caesar cipher program from the last chapter. It is slightly simpler, because this code only has to decrypt instead of decrypt or encrypt.

First we loop through every symbol in the ciphertext string stored in `message` on line 17. On each iteration of this loop, line 18 checks if `symbol` is an uppercase letter (that is, it exists in the `LETTERS` constant variable which only has uppercase letters) and, if so, decrypts it. Line 19

locates where `symbol` is in `LETTERS` with the `find()` method and stores it in a variable called `num`.

```

20.             num = num - key
21.
22.             # handle the wrap-around if num is 26 or larger or less than 0
23.             if num < 0:
24.                 num = num + len(LETTERS)

```

caesarHacker.py

Then we subtract the key from `num` on line 20. (Remember, in the Caesar cipher, subtracting the key decrypts and adding the key encrypts.) This may cause `num` to be less than zero and require “wrap-around”. Line 23 checks for this case and adds 26 (which is what `len(LETTERS)` returns) if it was less than 0.

```

26.             # add number's symbol at the end of translated
27.             translated = translated + LETTERS[num]

```

caesarHacker.py

Now that `num` has been modified, `LETTERS[num]` will evaluate to the decrypted symbol. Line 27 adds this symbol to the end of the string stored in `translated`.

```

29.         else:
30.             # just add the symbol without encrypting/decrypting
31.             translated = translated + symbol

```

caesarHacker.py

Of course, if the condition for line 18’s condition was `False` and `symbol` was not in `LETTERS`, we don’t decrypt the symbol at all. If you look at the indentation of line 29’s `else` statement, you can see that this `else` statement matches the `if` statement on line 18.

Line 31 just adds `symbol` to the end of `translated` unmodified.

String Formatting

```

33.     # display the current key being tested, along with its decryption
34.     print('Key #%s: %s' % (key, translated))

```

caesarHacker.py

Although line 34 is the only `print()` function call in our Caesar cipher hacker program, it will print out several lines because it gets called once per iteration of line 8’s `for` loop.

The argument for the `print()` function call is something we haven't used before. It is a string value that makes use of **string formatting** (also called **string interpolation**). String formatting with the `%s` text is a way of placing one string inside another one. The first `%s` text in the string gets replaced by the first value in the parentheses after the `%` at the end of the string.

Type 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.'
>>>
```

String formatting is often easier to type than string concatenation with the `+` operator, especially for larger strings. And one benefit of string formatting is that, unlike string concatenation, you can insert non-string values such as integers into the string. Try typing 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
>>>
```

Line 34 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'll use string formatting to put it in a string value that is passed to `print()`.

Practice Exercises, Chapter 7, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice7A>.

Summary

The critical failure of the Caesar cipher is that there aren't that many different possible keys that can be used to encrypt a message. Any computer can easily decrypt with all 26 possible keys, and it only takes the cryptanalyst a few seconds to look through them to find the one that is in English. To make our messages more secure, we will need a cipher that has more possible keys. That transposition cipher in the next chapter can provide this for us.



ENCRYPTING WITH THE TRANSPOSITION CIPHER

Topics Covered In This Chapter:

- Creating functions with `def` statements.
- `main()` functions
- Parameters
- The global and local scope, and global and local variables
- The `global` statement
- The list data type, and how lists and strings are similar
- The `list()` function
- Lists of lists
- Augmented assignment operators (`+=`, `-=`, `*=`, `/=`)
- The `join()` string method
- Return values and the `return` statement
- The special `__name__` variable

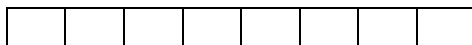
The Caesar cipher isn't secure. It doesn't take much for a computer to brute-force through all twenty-six possible keys. The transposition cipher has many more possible keys to make a brute-force attack more difficult.

Encrypting with the Transposition Cipher

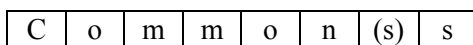
Instead of replacing characters with other characters, the transposition cipher jumbles up the message's symbols into an order that makes the original message unreadable. Before we start writing code, let's encrypt the message "Common sense is not so common." with pencil and

paper. Including the spaces and punctuation, this message has 30 characters. We will use the number 8 for the key.

The first step is to draw out a number of boxes equal to the key. We will draw 8 boxes since our key for this example is 8:



The second step is to start writing the message you want to encrypt into the boxes, with one character for each box. Remember that spaces are a character (this book marks the boxes with (s) to indicate a space so it doesn't look like an empty box).



We only have 8 boxes but there are 30 characters in the message. When you run out of boxes, draw another row of 8 boxes under the first row. Keep creating new rows until you have written out the full message:

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

We shade in the two boxes in the last row to remind us to ignore them. The ciphertext is the letters read from the top left box going down the column. “C”, “e”, “n”, and “o” are from the 1st column. 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 “Cenoonomstmme oo snnio. s s c”, which is sufficiently scrambled to keep someone from figuring out the original message by looking at it.

The steps for encrypting are:

1. Count the number of characters in the message and the key.
2. Draw a number of boxes equal to the key in a single row. (For example, 12 boxes for a key of 12.)
3. Start filling in the boxes from left to right, with one character per box.
4. When you run out of boxes and still have characters left, add another row of boxes.

5. Shade in the unused boxes in the last row.
6. Starting from the top left and going down, write out the characters. When you get to the bottom of the column, move to the next column to the right. Skip any shaded boxes. This will be the ciphertext.

Practice Exercises, Chapter 8, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice8A>.

A Transposition Cipher Encryption Program

Encrypting with paper and pencil involves a lot of work and it's easy to make mistakes. Let's look at a program that can implement transposition cipher encryption (a decryption program will be demonstrated later in this chapter).

Using the computer program has a slight problem, however. If the ciphertext has space characters at the end, then it is impossible to see them since a space is just empty... well, space. To fix this, the program adds a | character at the end of the ciphertext. (The | character is called the "pipe" character and is above the Enter key on your keyboard.) For example:

```
Hello| # There are no spaces at the end of the message.  
Hello | # There is one space at the end of the message.  
Hello  | # There are two spaces at the end of the message.
```

Source Code of the Transposition Cipher Encryption Program

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *transpositionEncrypt.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *transpositionEncrypt.py* file. You can download this file from <http://invpy.com/pyperclip.py>.

```
Source code for transpositionEncrypt.py  
1. # Transposition Cipher Encryption  
2. # http://inventwithpython.com/hacking (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 | (called "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.
25.     # Loop through each column in ciphertext.
26.     for col in range(key):
27.         pointer = col
28.
29.         # Keep looping until pointer goes past the length of the message.
30.         while pointer < len(message):
31.             # Place the character at pointer in message at the end of the
32.             # current column in the ciphertext list.
33.             ciphertext[col] += message[pointer]
34.
35.             # move pointer over
36.             pointer += key
37.
38.     # Convert the ciphertext list into a single string value and return it.
39.     return ''.join(ciphertext)
40.
41.
42. # If transpositionEncrypt.py is run (instead of imported as a module) call
43. # the main() function.
44. if __name__ == '__main__':
45.     main()

```

Sample Run of the Transposition Cipher Encryption Program

When you run the above program, it produces this output:

```
Cenoonommstme oo snnio. s s c|
```

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.

How the Program Works

```

transpositionEncrypt.py
1. # Transposition Cipher Encryption
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import pyperclip

```

The transposition cipher program, like the Caesar cipher program, will copy the encrypted text to the clipboard. So first we will import the `pyperclip` module so it can call `pyperclip.copy()`.

Creating Your Own Functions with `def` Statements

```

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

```

A function (like `print()`) is a sort of mini-program in your program. When the function is called, the execution moves to the code inside that function and then returns to the line after the function call. You can create your own functions with a `def` statement like the one on line 6.

The `def` statement on line 6 isn't a call to a function named `main()`. Instead, the `def` statement means we are creating, or **defining**, a new function named `main()` that we can call later in our program. When the execution reaches the `def` statement Python will *define* this function. We can then call it the same way we call other functions. When we *call* this function, the execution moves inside of the block of code following the `def` statement.

Open a new file editor window and type the following code into it:

```

Source code for helloFunction.py
1. def hello():
2.     print('Hello!')
3.     total = 42 + 1
4.     print('42 plus 1 is %s' % (total))
5. print('Start!')
6. hello()
7. print('Call it again.')
8. hello()

```



```
9. print('Done.')
```

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

```
Start!
Hello!
42 plus 1 is 43
Call it again.
Hello!
42 plus 1 is 43
Done.
```

When the *helloFunction.py* program runs, the execution starts at the top. The first line is a `def` statement that defines the `hello()` function. The execution skips the block after the `def` statement and executes the `print('Start!')` line. This is why 'Start!' is the first string printed when we run the program.

The next line after `print('Start!')` is a function call to our `hello()` function. The program execution jumps to the first line in the `hello()` function's block on line 2. This function will cause the strings 'Hello!' and '42 plus 1 is 43' to be printed to the screen.

When the program execution reaches the bottom of the `def` statement, the execution will jump back to the line after the line that originally called the function (line 7). In *helloFunction.py*, this is the `print('Call it again.')` line. Line 8 is *another* call to the `hello()` function. The program execution will jump back into the `hello()` function and execute the code there again. This is why 'Hello!' and '42 plus 1 is 43' are displayed on the screen two times.

After that function returns to line 9, the `print('Done.')` line executes. This is the last line in our program, so the program exits.

The Program's `main()` Function

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

transpositionEncrypt.py

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

On lines 7 and 8, the variables `myMessage` and `myKey` will store the plaintext message to encrypt and the key used to do the encryption.

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

transpositionEncrypt.py

The code that does the actual encrypting will be put into a function we define on line 21 named `encryptMessage()`. This function will take two arguments: an integer value for the key and a string value for the message to encrypt. When passing multiple arguments to a function call, separate the arguments with a comma.

The return value of `encryptMessage()` will be a string value of the encrypted ciphertext. (The code in this function is explained next.) This string will be stored in a variable named `ciphertext`.

```
12.     # Print the encrypted string in ciphertext to the screen, with
13.     # a | (called "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)
```

transpositionEncrypt.py

The ciphertext message is printed to the screen on line 15 and copied to the clipboard on line 18. The program prints a `|` character (called the “pipe” character) at the end of the message so that 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 will return to the line after the line that called it. The call to `main()` is on line 45 and is the last line in the program, so after execution returns from `main()` the program will exit.

Parameters

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

transpositionEncrypt.py

The code in the `encryptMessage()` function does the actual encryption. The `key` and `message` text in between the parentheses next to `encryptMessage()`'s `def` statement shows that the `encryptMessage()` function takes two parameters.

Parameters are the variables that contain the arguments passed when a function is called. Parameters are automatically deleted when the function returns. (This is just like how variables are forgotten when a program exits.)

When the `encryptMessage()` function gets called from line 10, two argument values are passed (on line 10, they are the values in `myKey` and `myMessage`). These values get assigned to the parameters `key` and `message` (which you can see on line 21) when the execution moves to the top of the function.

A parameter is a variable name in between the parentheses in the `def` statement. An argument is a value that is passed in between the parentheses for a function call.

Python will raise an error message if you try to call a function with too many or too few arguments for the number of parameters the function has. Try typing the following into the interactive shell:

```
>>> len('hello', 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (2 given)
>>> len()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (0 given)
>>>
```

Changes to Parameters Only Exist Inside the Function

Look at the following program, which defines and then calls a function named `func()`:

```
def func(param):
    param = 42
    spam = 'Hello'
    func(spam)
    print(spam)
```

When you run this program, the `print()` call on the last line will print out `'Hello'`, not `42`. When `func()` is called with `spam` as the argument, the `spam` variable is not being sent into the

`func()` function and having 42 assigned to it. Instead, 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 something called a list or dictionary value, but this will be explained in chapter 10 in the “List References” section.)

This is an important idea to understand. The argument value that is “passed” in a function call is *copied* to the parameter. So if the parameter is changed, the variable that provided the argument value is not changed.

Variables in the Global and Local Scope

You might wonder why we even have the `key` and `message` parameters to begin with, since we already have the variables `myKey` and `myMessage` from the `main()` function. The reason is because `myKey` and `myMessage` are in the `main()` function’s local scope and can’t be used outside of the `main()` function.

Every time a function is called, a **local scope** is created. Variables created during a function call exist in this local scope. Parameters always exist in a local scope. When the function returns, the local scope is destroyed and the local variables are forgotten. A variable in the local scope is still a separate variable from a global scope variable even if the two variables have the same name.

Variables created outside of every function exist in the **global scope**. 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 were global.)

The `global` Statement

If you want a variable that is assigned inside a function to be a global variable instead of a local variable, put a `global` statement with the variable’s name as the first line after the `def` statement.

Here are the rules for whether a variable is a global variable (that is, a variable that exists in the global scope) or local variable (that is, a variable that exists in a function call’s local scope):

1. Variables outside of all functions are always global variables.
2. If a variable in a function is never used in an assignment statement, it is a global variable.
3. If a variable in a function is not used in a `global` statement and but is used in an assignment statement, it is a local variable.
4. If a variable in a function is used in a `global` statement, it is a global variable when used in that function.

For example, type in the following short program, save it as *scope.py*, and press **F5** to run it:

Source code for *scope.py*

```

1. spam = 42
2.
3. def eggs():
4.     spam = 99 # spam in this function is local
5.     print('In eggs():', spam)
6.
7. def ham():
8.     print('In ham():', spam) # spam in this function is global
9.
10. def bacon():
11.     global spam # spam in this function is global
12.     print('In bacon():', spam)
13.     spam = 0
14.
15. def CRASH():
16.     print(spam) # spam in this function is local
17.     spam = 0
18.
19. print(spam)
20. eggs()
21. print(spam)
22. ham()
23. print(spam)
24. bacon()
25. print(spam)
26. CRASH()

```

The program will crash when Python executes line 16, and the output will look like this:

```

42
In eggs(): 99
42
In ham(): 42
42
In bacon(): 42
0
Traceback (most recent call last):
  File "C:\scope.py", line 27, in <module>
    CRASH()
  File "C:\scope.py", line 16, in CRASH

```

```
print(spam)
UnboundLocalError: local variable 'spam' referenced before assignment
```

When the `spam` variable is used on lines 1, 19, 21, 23, 25 it is outside of all functions, so this is the global variable named `spam`. In the `eggs()` function, the `spam` variable is assigned the integer 99 on line 4, so Python regards this `spam` variable as a local variable named `spam`. Python considers this local variable to be completely different from the global variable that is also named `spam`. Being assigned 99 on line 4 has no effect on the value stored in the global `spam` variable since they are different variables (they just happen to have the same name).

The `spam` variable in the `ham()` function on line 8 is never used in an assignment statement in that function, so it is the global variable `spam`.

The `spam` variable in the `bacon()` function is used in a `global` statement, so we know it is the global variable named `spam`. The `spam = 0` assignment statement on line 13 will change the value of the global `spam` variable.

The `spam` variable in the `CRASH()` function is used in an assignment statement (and not in a `global` statement) so the `spam` variable in that function is a local variable. However, notice that it is used in the `print()` function call on line 16 before it is assigned a value on line 17. This is why calling the `CRASH()` function causes our program to crash with the error, `UnboundLocalError: local variable 'spam' referenced before assignment`.

It can be confusing to have global and local variables with the same name, so even if you remember the rules for how to tell global and local variables apart, you would be better off using different names.

Practice Exercises, Chapter 8, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice8B>.

The List Data Type

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

Line 23 uses a new data type called the **list** data type. A list value can contain other values. Just like how strings begin and end with quotes, a list value begins with a `[` open bracket and ends

with] close bracket. The values stored inside the list are typed within the brackets. If there is more than one value in the list, the values are separated by commas.

Type 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 of a list are also called **items**. Lists are very good when we have to store lots and lots of values, but we don't want variables for each one. Otherwise we would have something like this:

```
>>> animals1 = 'aardvark'
>>> animals2 = 'anteater'
>>> animals3 = 'antelope'
>>> animals4 = 'albert'
>>>
```

This makes working with all the strings as a group very hard, especially if you have hundreds, thousands, or millions of different values that you want stored in a list.

Many of the things you can do with strings will 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. Try typing the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[0]
'aardvark'
>>> animals[1]
'anteater'
>>> animals[2]
'antelope'
>>> animals[3]
'albert'
>>> animals[1:3]
['anteater', 'antelope']
>>>
```

Remember, the first index is 0 and not 1. While using slices with a string value will give you a string value of part of the original string, using slices with a list value will give you a list value of part of the original list.

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

```
>>> for spam in ['aardvark', 'anteater', 'antelope', '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 antelope
For dinner we are cooking albert
>>>
```

Using the `list()` Function to Convert Range Objects to Lists

If you need a list value that has increasing integer amounts, you could have code like this to build up a list value using a `for` loop:

```
>>> myList = []
>>> for i in range(10):
...     myList = myList + [i]
...
>>> myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

However, it is simpler to directly make a list from a range object that the `range()` function returned by using the `list()` function:

```
>>> myList = list(range(10))
>>> myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

The `list()` function can also convert strings into a list value. The list will have several single-character strings that were in the original string:

```
>>> myList = list('Hello world!')
>>> myList
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
>>>
```

We won't be using the `list()` function on strings or range objects in this program, but it will come up in later in this book.

Reassigning the Items in Lists

The items inside a list can also be modified. Use the index with a normal assignment statement. Try typing the following into the interactive shell:

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

Reassigning Characters in Strings

While you can reassign items in a list, you cannot reassign a character in a string value. Try typing the following code into the interactive shell to cause this error:

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

To change a character in a string, use slicing instead. Try typing the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam = spam[:6] + 'x' + spam[7:]
>>> spam
'Hello xorld!'
>>>
```

Lists of Lists

List values can even contain other list values. Try typing 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 double index brackets used for `spam[0][0]` work because `spam[0]` evaluates to `['dog', 'cat']` and `['dog', 'cat'][0]` evaluates to `'dog'`. You could even use another set of index brackets, since string values also use them:

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

Say we had a list of lists stored in a variable named `x`. Here are the indexes for each of the items in `x`. Notice that `x[0]`, `x[1]`, `x[2]`, and `x[3]` refer to list values:

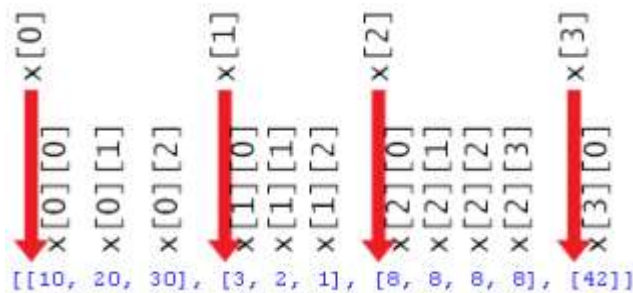


Figure 8-1. A list of lists with every item's index labeled.

Practice Exercises, Chapter 8, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice8C>.

Using `len()` and the `in` Operator with Lists

We've used the `len()` function to tell us how many characters are in a string (that is, the length of the string). The `len()` function also works on list values and returns an integer of how many items are in the list.

Try typing the following into the interactive shell:

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

We've used the `in` operator to tell us if a string exists inside another string value. The `in` operator also works for checking if a value exists in a list. Try typing the following into the interactive shell:

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

Just like how a set of quotes next to each other represents the blank string value, a set of brackets next to each other represents a blank list. Try typing the following into the interactive shell:

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

List Concatenation and Replication with the `+` and `*` Operators

Just like how the `+` and `*` operators can concatenate and replicate strings, the same operators can concatenate and replicate lists. Try typing 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 things you can do with string values will also work with list values.

Practice Exercises, Chapter 8, Set D

Practice exercises can be found at <http://invpy.com/hackingpractice8D>.

The Transposition Encryption Algorithm

We need to translate these paper-and-pencil steps into Python code. Let's take a look at encrypting the string 'Common sense is not so common.' with the key 8. If we wrote out the boxes with pencil and paper, it would look like this:

C	o	m	m	o	n	(s)	s
e	n	s	e	(s)	i	s	(s)
n	o	t	(s)	s	o	(s)	c
o	m	m	o	n	.		

Add the index of each letter in the string to the boxes. (Remember, indexes begin with 0, not 1.)

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

We can see from these boxes 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'). We can see a pattern emerging: The n^{th} column will have all the characters in the string at indexes $0 + n$, $8 + n$, $16 + n$, and $24 + n$:

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

There is an exception for the 6th and 7th columns, since $24 + 6$ and $24 + 7$ are greater than 29, which is the largest index in our string. In those cases, we only use 0, 8, and 16 to add to n (and skip 24).

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

So, for the n^{th} column's string we start at index n , and then keep adding 8 (which is the key) to get the next index. We keep adding 8 as long as the index is less than 30 (the message length), at which point we move to the next column.

If we imagine a list of 8 strings where each string is made up of the characters in each column, then the list value would look like this:

```
['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']
```

This is how we can simulate the boxes in Python code. First, we will make a list of blank strings. This list will have a number of blank strings equal to the key because each string will represent a column of our paper-and-pencil boxes. (Our list will have 8 blank strings since we are using the key 8 in our example.) Let's look at the code.

```

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

```

The `ciphertext` variable will be a list of string values. Each string in the `ciphertext` variable will represent a column of the grid. So `ciphertext[0]` is the leftmost column, `ciphertext[1]` is the column to the right of that, and so on.

The string values will have all the characters that go into one column of the grid. Let's look again at the grid from the "Common sense is not so common." example earlier in this chapter (with column numbers added to the top):

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

The `ciphertext` variable for this grid would look like this:

```

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

```

The first step to making this list is to create as many blank strings in the `ciphertext` list as there are columns. Since the number of columns is equal to the key, we 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.

```

transpositionEncrypt.py
25.     # Loop through each column in ciphertext.
26.     for col in range(key):
27.         pointer = col

```

The next step is to add text to each string in `ciphertext`. The `for` loop on line 26 will iterate once for each column, and the `col` variable will have the correct integer value to use for the index to `ciphertext`. The `col` variable will be set to 0 for the first iteration through the `for` loop, then 1 on the second iteration, then 2 and so on. This way the expression `ciphertext[col]` will be the string for the `col`th column of the grid.

Meanwhile, the `pointer` variable will be used as the index for the string value in the `message` variable. On each iteration through the loop, `pointer` will start at the same value as `col` (which is what line 27 does.)

Augmented Assignment Operators

Often when you are assigning a new value to a variable, you want it to be based off of the variable's current value. To do this you use the variable as the part of the expression that is evaluated and assigned to the variable, like this example in the interactive shell:

```

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

```

But you can instead use the `+=` augmented assignment operator as a shortcut. Try typing the following into the interactive shell:

```

>>> spam = 40
>>> spam += 2
>>> print(spam)
42
>>> spam = 'Hello'
>>> spam += ' world!'
>>> print(spam)

```

```

Hello world!
>>> spam = ['dog']
>>> spam += ['cat']
>>> print(spam)
['dog', 'cat']
>>>

```

The statement `spam += 2` does the *exact 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 8-1 shows the augmented assignment operators and what they are equivalent to:

Table 8-1. Augmented Assignment Operators

Augmented Assignment	Equivalent Normal Assignment
<code>spam += 42</code>	<code>spam = spam + 42</code>
<code>spam -= 42</code>	<code>spam = spam - 42</code>
<code>spam *= 42</code>	<code>spam = spam * 42</code>
<code>spam /= 42</code>	<code>spam = spam / 42</code>

Back to the Code

```

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

```

Inside the `for` loop that started on line 26 is a `while` loop that starts on line 30. For each column, we want to loop through the original `message` variable and pick out every `keyth` character. (In the example we've been using, we want every 8th character since we are using a key of 8.) On line 27 for the first iteration of the `for` loop, `pointer` was set to 0.

While the value in `pointer` is less than the length of the `message` string, we want to add the character at `message[pointer]` to the end of the `colth` string in `ciphertext`. We add 8 (that is, the value in `key`) to `pointer` each time through the loop on line 36. The first time it is `message[0]`, the second time `message[8]`, the third time `message[16]`, and the fourth time `message[24]`. Each of these single character strings are concatenated to the end of

ciphertext[col] (and since col is 0 on the first time through the loop, this is ciphertext[0]).

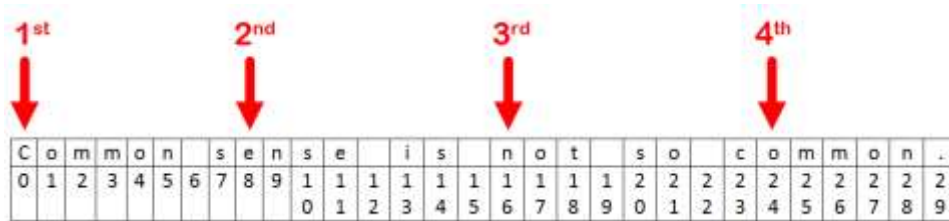


Figure 8-2. Arrows pointing to what message[pointer] refers to during the first iteration of the for loop when col is set to 0.

Figure 8-2 shows the characters at these indexes, they will be concatenated together to form the string 'Ceno'. Remember that we want the value in ciphertext to eventually look like this:

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

Storing 'Ceno' as the first string in the ciphertext list is our first step.

On the next iteration of the for loop, col will be set to 1 (instead of 0) and pointer will start at the same value as col. Now when we add 8 to pointer on each iteration of line 30's while loop, the indexes will be 1, 9, 17, and 25.

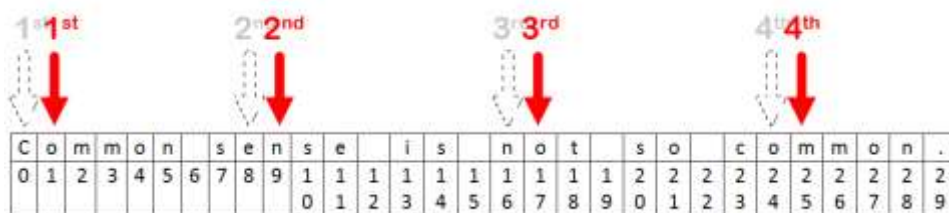


Figure 8-3. Arrows pointing to to what message[pointer] refers to during the second iteration of the for loop when col is set to 1.

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 our grid.

Once the `for` loop has finished looping for the rest of the columns, the value in `ciphertext` will be `['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c']`. We will use the `join()` string method to convert this list of strings into a single string.

The `join()` String Method

The `join()` method is used later on line 39. The `join()` method takes a list of strings and returns a single string. This single string has all of the strings in the list concatenated (that is, joined) together. The string that the `join()` method gets called on will be placed in between the strings in the list. (Most of the time, we will just use a blank string for this.) Try typing the following into the interactive shell:

```
>>> eggs = ['dogs', 'cats', 'moose']
>>> ''.join(eggs)
'dogscatsmoose'
>>> ' '.join(eggs)
'dogs cats moose'
>>> 'XYZ'.join(eggs)
'dogsXYZcatsXYZmoose'
>>> ''.join(eggs).upper().join(eggs)
'dogsDOGSCATSMOOSEcatsDOGSCATSMOOSEmoose'
>>>
```

That last expression, `''.join(eggs).upper().join(eggs)`, looks a little tricky, but if you go through the evaluation one step at a time, it will look like this:

```
''.join(eggs).upper().join(eggs)
    ↓
''.join(['dogs', 'cats', 'moose']).upper().join(eggs)
    ↓
'dogscatsmoose'.upper().join(eggs)
    ↓
'DOGSCATSMOOSE'.join(eggs)
    ↓
'DOGSCATSMOOSE'.join(['dogs', 'cats', 'moose'])
    ↓
'dogsDOGSCATSMOOSEcatsDOGSCATSMOOSEmoose'
```

Figure 8-4. The steps of evaluation for `''.join(eggs).upper().join(eggs)`

This is why `''.join(eggs).upper().join(eggs)` returns the string, `'dogsDOGSCATSMOOSEcatsDOGSCATSMOOSEmoose'`.

Whew!

Remember, no matter how complicated an expression looks, you can just evaluate it step by step to get the single value the expression evaluates to.

Return Values and `return` Statements

```

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

```

Our use of the `join()` method isn't nearly as complicated as the previous example. We just want to call `join()` on the blank string and pass `ciphertext` as the argument so that the strings in the `ciphertext` list are joined together (with nothing in between them).

Remember that a function (or method) call always evaluates to a value. We say that this is the value *returned* by the function or method call, or that it is the *return value* of the function. When we create our own functions with a `def` statement, we use a `return` statement to tell what the return value for our function is.

A `return` statement is the `return` keyword followed by the value to be returned. We can also use an expression instead of a value. In that case the return value will be whatever value that expression evaluates to. Open a new file editor window and type the following program in and save it as `addNumbers.py`, then press **F5** to run it:

Source code for `addNumbers.py`

```

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

```

When you run this program, the output will be:

```
42
```

That's because the function call `addNumbers(2, 40)` will evaluate to 42. The `return` statement in `addNumbers()` will evaluate the expression `a + b` and then return the evaluated

value. That is why `addNumbers(2, 40)` evaluates to 42, which is the value stored in `spam` on line 4 and next printed to the screen on line 5.

Practice Exercises, Chapter 8, Set E

Practice exercises can be found at <http://invpy.com/hackingpractice8E>.

Back to the Code

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

The `encryptMessage()` function's return statement returns a string value that is created by joining all of the strings in the `ciphertext` list. This final string is the result of our encryption code.

The great thing about functions is that a programmer only has to know what the function does, but not how the function's code does it. A programmer can understand that if she calls the `encryptMessage()` function and pass it an integer and a string for the `key` and `message` parameters, the function call will evaluate to an encrypted string. She doesn't need to know anything about how the code in `encryptMessage()` actually does this.

The Special `__name__` Variable

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

We can turn our transposition encryption program into a module with a special trick involving the `main()` function and a variable named `__name__`.

When a Python program is run, there is a special variable with the name `__name__` (that's two underscores before "name" and two underscores after) that is assigned the string value `'__main__'` (again, two underscores before and after "main") even before the first line of your program is run.

At the end of our script file (and, more importantly, after all of our `def` statements), we want to have some code that checks if the `__name__` variable has the `'__main__'` string assigned to it. If so, we want to call the `main()` function.

This `if` statement on line 44 ends up actually being one of the first lines of code executed when we press **F5** to run our transposition cipher encryption program (after the `import` statement on line 4 and the `def` statements on lines 6 and 21).

The reason we set up our code this way is although Python sets `__name__` to `'__main__'` when the program is run, it sets it to the string `'transpositionEncrypt'` if our program is imported by a different Python program. This is how our program can know if it is being run as a program or imported by a different program as a module.

Just like how our program imports the `pyperclip` module to call the functions in it, other programs might want to import `transpositionEncrypt.py` to call its `encryptMessage()` function. When an `import` statement is executed, Python will look for a file for the module by adding “.py” to the end of the name. (This is why `import pyperclip` will import the `pyperclip.py` file.)

When a Python program is imported, the `__name__` variable is set to the filename part before “.py” and then runs the program. When our `transpositionEncrypt.py` program is imported, we want all the `def` statements to be run (to define the `encryptMessage()` function that the importing program wants to use), but we don’t want it to call the `main()` function because that will execute the encryption code for `'Common sense is not so common.'` with key 8.

That is why we put that part of the code inside a function (which by convention is named `main()`) and then add code at the end of the program to call `main()`. If we do this, **then our program can both be run as a program on its own and also imported as a module by another program.**

Key Size and Message Length

Notice what happens when the message length is less than twice the key size:

C	o	m	m	o	n	(s)	s	e	n	s	e	(s)	i	s	(s)	n	o	t	(s)	s	o	(s)	c	o
m	m	o	n	.																				

When using a key of 25, the “Common sense is not so common.” message encrypts to “Cmommomno.n sense is not so co”. Part of the message isn’t encrypted! This happens whenever key size becomes more than twice the message length, because that causes there to only be one character per column and no characters get scrambled for that part of the message.

Because of this, the transposition cipher’s key is limited to half the length of the message it is used to encrypt. The longer a message is, the more possible keys that can be used to encrypt it.

Summary

Whew! There were a lot of new programming concepts introduced in this chapter. The transposition cipher program is much more complicated (but much more secure) than the Caesar cipher program in the last chapter. The new concepts, functions, data types, and operators we've learned in this chapter let us manipulate data in much more sophisticated ways. Just remember that much of understanding a line of code is just evaluating it step by step the way Python will.

We can organize our code into groups called functions, which we create with `def` statements. Argument values can be passed to functions for 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.

List values can store multiple other values, including other list values. Many of the things you can do with strings (such as indexing, slicing, and 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.

Feel free to go over this chapter again if you are not comfortable with these programming concepts. In the next chapter, we will cover decrypting with the transposition cipher.



DECRYPTING WITH THE TRANSPOSITION CIPHER

Topics Covered In This Chapter:

- Decrypting with the transposition cipher
- The `math.ceil()`, `math.floor()` and `round()` functions
- The `and` and `or` boolean operators
- Truth tables

“When stopping a terrorist attack or seeking to recover a kidnapped child, encountering encryption may mean the difference between success and catastrophic failures.”

Attorney General Janet Reno, September 1999

“Even the Four Horsemen of Kid Porn, Dope Dealers, Mafia and Terrorists don’t worry me as much as totalitarian governments. It’s been a long century, and we’ve had enough of them.”

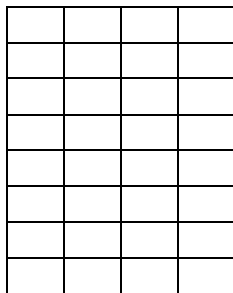
Bruce Sterling, 1994 Computers, Freedom, and Privacy conference

Unlike the Caesar cipher, the decryption process for the transposition cipher is very different from the encryption process. In this chapter we will create a separate program, *transpositionDecrypt.py*, to handle decryption.

Decrypting with the Transposition Cipher on Paper

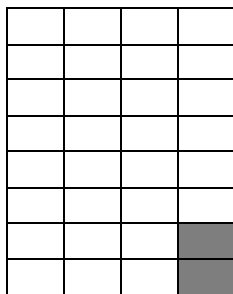
Let's pretend we send the ciphertext "Cenoonommstmme oo snnio. s s c" to a friend (and she already knows that the secret key is 8). The first step for her to decrypt the ciphertext is to calculate how many boxes she needs to draw. To find this amount, divide the length of the ciphertext message by the key and round up. The length of our ciphertext is 30 characters (exactly the same as the plaintext) and the key is 8. So calculate 30 divided by 8 to get 3.75.

3.75 rounds up to 4. This means we want to draw a grid of boxes with 4 columns (the number we just calculated) and 8 rows (the key). It will look like this:



(Note that if the length divided by the key was a whole number, like in $30 / 5 = 6.0$, then 6.0 would not "round up" to 7.)

The second thing we need to calculate is how many boxes on the rightmost column to shade in. Take the total number of boxes (32) and subtract the length of the ciphertext (30). $32 - 30 = 2$, so shade in the *bottom 2* boxes on the *rightmost* column:



Then start filling in the boxes with one character of the ciphertext per box. Start at the top left and go right, just like we did when we were encrypting. The ciphertext is “Cenoonommstmme oo snnio. s s c”, and so “Ceno” goes in the first row, then “onom” in the second row, and so on. After we are done, the boxes will look like this (where the (s) represents a space):

C	e	n	o
o	n	o	m
m	s	t	m
m	e	(s)	o
o	(s)	s	n
n	i	o	.
(s)	s	(s)	
s	(s)	c	

Our friend who received the ciphertext will see that if she reads the text going down the columns, the original plaintext has been restored: “Common sense is not so common.”

The steps for decrypting are:

1. Calculate the number of columns you will have by taking the length of the message and dividing by the key, then rounding up.
2. Draw out a number of boxes. The number of columns was 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 number of boxes (this is the number of rows and columns multiplied) 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 from the leftmost column going from top to bottom, and moving over to the top of the next column.

Note that if you use a different key, you will be drawing out the wrong number of rows. Even if you follow the other steps in the decryption process correctly, the plaintext will come out looking like random garbage (just like when you use the wrong key with the Caesar cipher).

Practice Exercises, Chapter 9, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice9A>.

A Transposition Cipher Decryption Program

Open a new file editor window and type out the following code in it. Save this program as *transpositionDecrypt.py*.

Source Code of the Transposition Cipher Decryption Program

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *transpositionDecrypt.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *transpositionDecrypt.py* file. You can download this file from <http://invpy.com/pyperclip.py>.

```
Source code for transpositionDecrypt.py
1. # Transposition Cipher Decryption
2. # http://inventwithpython.com/hacking (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 = math.ceil(len(message) / key)
26.     # The number of "rows" in our grid will need:
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 = [''] * numColumns
33.
34.     # The col and row variables point to where in the grid the next
35.     # character in the encrypted message will go.
36.     col = 0
37.     row = 0
38.
39.     for symbol in message:
40.         plaintext[col] += symbol
41.         col += 1 # point to next column
42.
43.         # If there are no more columns OR we're at a shaded box, go back to
44.         # the first column and the next row.
45.         if (col == numColumns) or (col == numColumns - 1 and row >=
numOfRows - numOfShadedBoxes):
46.             col = 0
47.             row += 1
48.
49.     return ''.join(plaintext)
50.
51.
52. # If transpositionDecrypt.py is run (instead of imported as a module) call
53. # the main() function.
54. if __name__ == '__main__':
55.     main()

```

When you run the above 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 5 and 6.

How the Program Works

```

1. # Transposition Cipher Decryption
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import math, pyperclip
5.
6. def main():
7.     myMessage = 'Cenoonommstmme oo snnio. s s c'
8.     myKey = 8
9.

```

transpositionDecrypt.py

```

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 first part of the program is very similar to the first part of *transpositionEncrypt.py*. The `pyperclip` module is imported along with a different module named `math`. If you separate the module names with commas, you can import multiple modules with one `import` statement.

The `main()` function creates variables named `myMessage` and `myKey`, and then calls the decryption function `decryptMessage()`. The return value of this function is the decrypted plaintext of the ciphertext and key that we passed it. This is stored in a variable named `plaintext`, which is then printed to the screen (with a pipe character at the end in case there are spaces at the end of the message) and copied to the clipboard.

```

19. def decryptMessage(key, message):
transpositionDecrypt.py

```

Look at the six steps to decrypting from earlier in this chapter. For example, if we are decrypting “Cenoonommstmme oo snnio. s s c” (which has 30 characters) with the key 8, then the final set of boxes will look like this:

C	e	n	o
o	n	o	m
m	s	t	m
m	e	(s)	o
o	(s)	s	n
n	i	o	.
(s)	s	(s)	
s	(s)	c	

The `decryptMessage()` function implements each of the decryption steps as Python code.

The `math.ceil()`, `math.floor()` and `round()` Functions

When you divide numbers using the `/` operator, the expression returns a floating point number (that is, a number with a decimal point). This happens even if the number divides evenly. For example, `21 / 7` will evaluate to `3.0`, not `3`.

```
>>> 21 / 7
3.0
>>>
```

This is useful because if a number does not divide evenly, the numbers after the decimal point will be needed. For example, $22 / 5$ evaluates to 4.4 :

```
>>> 22 / 5
4.4
>>>
```

(If the expression $22 / 5$ evaluates to 4 instead of 4.4 , then you are using version 2 of Python instead of version 3. Please go to the <http://python.org> website and download and install Python 3.)

If you want to round this number to the nearest integer, you can use the `round()` function. Type the following into the interactive shell:

```
>>> round(4.2)
4
>>> round(4.5)
4
>>> round(4.9)
5
>>> round(5.0)
5
>>> round(22 / 5)
4
>>>
```

If you only want to round up then use the `math.ceil()` function, which stands for “ceiling”. If you only want to round down then use the `math.floor()` function. These functions exist in the `math` module, so you will need to import the `math` module before calling them. Type 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.ceil()` function will implement step 1 of the transposition decryption.

```

transpositionDecrypt.py
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 = math.ceil(len(message) / key)
26.     # The number of "rows" in our grid will need:
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 (step 1 of decrypting) 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`.

Line 27 calculates the number of rows (step 2 of decrypting), 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 (step 3 of decrypting), which will be the number of columns times rows, minus the length of the message.

If we are decrypting “Cenoonommstmme oo snnio. s s c” with key 8, `numOfColumns` will be set to 4, `numOfRows` will be set to 8, and `numOfShadedBoxes` will be set to 2.

```

transpositionDecrypt.py
31.     # Each string in plaintext represents a column in the grid.
32.     plaintext = [''] * numOfColumns

```

Just like the encryption program had a variable named `ciphertext` that was a list of strings to represent the grid of ciphertext, the `decryptMessage()` function will have a variable named `plaintext` that is a list of strings. These strings start off as blank strings, and we will need one

string for each column of the grid. Using list replication, we can multiply a list of one blank string by `numOfColumns` to make a list of several blank strings.

(Remember that each function call has its own local scope. The `plaintext` in `decryptMessage()` exists in a different local scope than the `plaintext` variable in `main()`, so they are two different variables that just happen to have the same name.)

Remember that the grid for our 'Cenoonommstme oo snnio. s s c' example looks like this:

C	e	n	o
o	n	o	m
m	s	t	m
m	e	(s)	o
o	(s)	s	n
n	i	o	.
(s)	s	(s)	
s	(s)	c	

The `plaintext` variable will have a list of strings. Each string in the list is a single column of this grid. For this decryption, we want `plaintext` to end up with this value:

```
>>> plaintext = ['Common s', 'ense is ', 'not so c', 'ommon.']
>>> plaintext[0]
'Common s'
```

That way, we can join all the list's strings together to get the 'Common sense is not so common.' string value to return.

```

                                                                    transpositionDecrypt.py
34.     # The col and row variables point to where in the grid the next
35.     # character in the encrypted message will go.
36.     col = 0
37.     row = 0
38.
39.     for symbol in message:
```

The `col` and `row` variables will track the column and row where the next character in `message` should go. We will start these variables at 0. Line 39 will start a `for` loop that iterates over the characters in the `message` string. Inside this loop the code will adjust the `col` and `row` variables so that we concatenate `symbol` to the correct string in the `plaintext` list.

```
40.         plaintext[col] += symbol
41.         col += 1 # point to next column
```

transpositionDecrypt.py

As the first step in this loop we concatenate `symbol` to the string at index `col` in the `plaintext` list. Then we add 1 to `col` (that is, we **increment** `col`) on line 41 so that on the next iteration of the loop, `symbol` will be concatenated to the next string.

The and and or Boolean Operators

The Boolean operators `and` and `or` can help us 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`. Type the following into the interactive shell:

```
>>> 10 > 5 and 2 < 4
True
>>> 10 > 5 and 4 != 4
False
>>>
```

The first expression above evaluates to `True` because the two expressions on the sides of the `and` operator both evaluate to `True`. This means that the expression `10 > 5 and 2 < 4` evaluates to `True and True`, which in turn evaluates to `True`.

However, for the second above expression, although `10 > 5` evaluates to `True` the expression `4 != 4` evaluates to `False`. This means the whole expression evaluates to `True and False`. Since both expressions have to be `True` for the `and` operator to evaluate to `True`, instead they evaluate to `False`.

Type the following into the interactive shell:

```
>>> 10 > 5 or 4 != 4
True
>>> 10 < 5 or 4 != 4
False
>>>
```

For the `or` operator, only one of the sides must be `True` for the `or` operator to evaluate them both to `True`. This is why `10 > 5 or 4 != 4` evaluates to `True`. However, because both

the expression `10 < 5` and the expression `4 != 4` are both `False`, this makes the second above expression evaluate to `False or False`, which in turn evaluates to `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`. Type 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
>>>
```

Practice Exercises, Chapter 9, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice9B>.

Truth Tables

If you ever forget how the Boolean operators work, you can look at these charts, which are called **truth tables**:

Table 6-1: The `and` operator's truth table.

A	and	B	is	Entire statement
True	and	True	is	True
True	and	False	is	False
False	and	True	is	False
False	and	False	is	False

Table 6-2: The `or` operator's truth table.

A	or	B	is	Entire statement
True	or	True	is	True
True	or	False	is	True
False	or	True	is	True
False	or	False	is	False

Table 6-3: The `not` operator's truth table.

not A	is	Entire statement
not True	is	False
not False	is	True

The `and` and `or` Operators are Shortcuts

Just like `for` loops let us do the same thing as `while` loops but with less code, the `and` and `or` operators let us shorten our code also. Type in the following into the interactive shell. Both of these bits of code do the same thing:

```
>>> if 10 > 5:
...     if 2 < 4:
...         print('Hello!')
...
Hello!
>>>
>>> if 10 > 5 and 2 < 4:
...     print('Hello!')
...
Hello!
>>>
```

So you can see that the `and` operator basically takes the place of two `if` statements (where the second `if` statement is inside the first `if` statement's block.)

You can also replace the `or` operator with an `if` and `elif` statement, though you will have to copy the code twice. Type 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!
>>>
```

Order of Operations for Boolean Operators

Just like the math operators have an order of operations, the `and`, `or`, and `not` operators also have an order of operations: first `not`, then `and`, and then `or`. Try typing 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
```

Back to the Code

```

                                                                    transpositionDecrypt.py
43.         # If there are no more columns OR we're at a shaded box, go back to
44.         # the first column and the next row.
45.         if (col == numOfColumns) or (col == numOfColumns - 1 and row >=
numOfRows - numOfShadedBoxes):
46.             col = 0
47.             row += 1
```

There are two cases where we want to reset `col` back to 0 (so that on the next iteration of the loop, symbol is added to the first string in the list in `plaintext`). The first is if we have incremented `col` past the last index in `plaintext`. In this case, the value in `col` will be equal to `numOfColumns`. (Remember that the last index in `plaintext` will be `numOfColumns` minus one. So when `col` is equal to `numOfColumns`, it is already past the last index.)

The second case is if both `col` is at the last index and the `row` variable is pointing to a row that has a shaded box in the last column. Here's the complete decryption grid with the column indexes along the top and the row indexes down the side:

	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	(s) 14	o 15
4	o 16	(s) 17	s 18	n 19
5	n 20	i 21	o 22	. 23
6	(s) 24	s 25	(s) 26	
7	s 27	(s) 28	c 29	

You can see that the shaded boxes are in the last column (whose index will be `numOfColumns - 1`) and rows 6 and 7. To have our program calculate which row indexes are shaded, we use the expression `row >= numOfRows - numOfShadedBoxes`. If this expression is `True`, and `col` is equal to `numOfColumns - 1`, then we know that we want to reset `col` to 0 for the next iteration.

These two cases are why the condition on line 45 is `(col == numOfColumns) or (col == numOfColumns - 1 and row >= numOfRows - numOfShadedBoxes)`. That looks like a big, complicated expression but remember that you can break it down into smaller parts. The block of code that executes will change `col` back to the first column by setting it to 0. We will also increment the `row` variable.

```
49.         return ''.join(plaintext) transpositionDecrypt.py
```

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 that they are now in the decrypted order (if the correct key was used, that is). The strings in the `plaintext` list are joined together (with a blank string in between each string) by the `join()` string method. The string that this call to `join()` returns will be the value that our `decryptMessage()` function returns.

For our example decryption, plaintext will be ['Common s', 'ense is ', 'not so c', 'ommon.'], so ''.join(plaintext) will evaluate to 'Common sense is not so common.'

```

                                                                    transpositionDecrypt.py
52. # If transpositionDecrypt.py is run (instead of imported as a module) call
53. # the main() function.
54. if __name__ == '__main__':
55.     main()

```

The first line that our program runs after importing modules and executing the `def` statements is the `if` statement on line 54. Just like in the transposition encryption program, we check if this program has been run (instead of imported by a different program) by checking if the special `__name__` variable is set to the string value `'__main__'`. If so, we execute the `main()` function.

Practice Exercises, Chapter 9, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice9C>.

Summary

That's it for the decryption program. Most of the program is in the `decryptMessage()` function. We can see that our programs can encrypt and decrypt the message "Common sense is not so common." with the key 8. But we should try several other messages and keys to see that a message that is encrypted and then decrypted will result in the same thing as the original message. Otherwise, we know that either the encryption code or the decryption code doesn't work.

We could start changing the `key` and `message` variables in our *transpositionEncrypt.py* and *transpositionDecrypt.py* and then running them to see if it works. But instead, let's automate this by writing a program to test our program.



PROGRAMMING A PROGRAM TO TEST OUR PROGRAM

Topics Covered In This Chapter:

- The `random.seed()` function
- The `random.randint()` function
- List references
- The `copy.deepcopy()` Functions
- The `random.shuffle()` function
- Randomly scrambling a string
- The `sys.exit()` function

“It is poor civic hygiene to install technologies that could someday facilitate a police state.”

Bruce Schneier, cryptographer

We can try out the transposition encryption and decryption programs from the previous chapter by encrypting and decrypting a few messages with different keys. It seems to work pretty well. But does it *always* work?

You won't know unless you test the `encryptMessage()` and `decryptMessage()` functions with different values for the `message` and `key` parameters. This would take a lot of time. You'll have to type out 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. And you'll want to repeat that with several different keys and messages!

That's a lot of boring work. Instead we can write a program to test the cipher programs for us. This new program can generate a random message and a random key. It will then encrypt the message with the `encryptMessage()` function from *transpositionEncrypt.py* and then pass the ciphertext from that to the `decryptMessage()` in *transpositionDecrypt.py*. If the plaintext returned by `decryptMessage()` is the same as the original message, the program can know that the encryption and decryption messages work. This is called **automated testing**.

There are several different message and key combinations to try, but it will only take the computer a minute or so to test thousands different combinations. If all of those tests pass, then we can be much more certain that our code works.

Source Code of the Transposition Cipher Tester Program

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *transpositionTest.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *transpositionTest.py* file. You can download this file from <http://inropy.com/pyperclip.py>.

Source code for transpositionTest.py

```

1. # Transposition Cipher Test
2. # http://inventwithpython.com/hacking (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)
17.        random.shuffle(message)

```

```

18.         message = ''.join(message) # convert list to string
19.
20.         print('Test #%s: "%s..." % (i+1, message[:50]))
21.
22.         # Check all possible keys for each message.
23.         for key in range(1, len(message)):
24.             encrypted = transpositionEncrypt.encryptMessage(key, message)
25.             decrypted = transpositionDecrypt.decryptMessage(key, encrypted)
26.
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)
32.                 sys.exit()
33.
34.         print('Transposition cipher test passed.')
35.
36.
37. # If transpositionTest.py is run (instead of imported as a module) call
38. # the main() function.
39. if __name__ == '__main__':
40.     main()

```

Sample Run of the Transposition Cipher Tester Program

When you run this program, the output will look like this:

```

Test #1: "KQDXSFQDBPMMRGXFKCGIQUGWFFLAJIJKFJGSYOSAWGYBGUNTQX..."
Test #2: "IDDXEEWUMWUJPSZ FJSGAOMFIOWWEYANRXISCJKXZRHMNCFYW..."
Test #3: "DKAYRSAGSGCSIQWKGARQHAOZDLGKJISQVMDFGYXKCRMPQMOWJM..."
Test #4: "MZIBCOEXGRDTFXZKVNFWQWMIROJAOKTWISTDWAHZRVIGXOLZA..."
Test #5: "TINIECNCFKJBRDIUTNGDINHULYSVTGHBARDQMZCNHZOTNYHSX..."
Test #6: "JZQIHCVNDWRDUFHFXCIASYDSTGQATQOYL IHUFPKEXSOZXQGPP..."
Test #7: "BMKJUJERFNGIDGWAPQMDZNHOQPLEOQDYCIIWRKPVEIPLAGZCJVN..."
Test #8: "IPASTGZSLPYCORCVEKWHOLOVUFPOMGQWZVJNYQIYVEOFLUWLMQ..."
Test #9: "AHRYJAPTACZQNNFOTONMIPYECOORDGEYESYFHROZDASFIPKSOP..."
Test #10: "FSXAAPLSQHSFUPQZGTIXXDLDMOIVMWFHGPBPJROOSEGPEVRXSX..."
Test #11: "IVBCXBIHLWPTDHGEGANBGXWQZMVXQPNJZQPKMRUMPLLPAPFITN..."
Test #12: "LLNSYMNRXZVYNPRVTNIBFRSUGIWUJREMPZVCMJATMLAMCEEHNW..."
Test #13: "IMWUJ JHRWAABHYIHGNPSJUOVKRRKBSJKDHOBLOUJDCXIVDME..."
Test #14: "IZVXWHTIGKGHKJGGWMOBAKTWZJPHGNEQPINYZIBERJPUNWJMX..."
Test #15: "BQGFNMQCIBOTRHZZOBHZFJZVSRVHIUJFOWRFBNWKRNHGOHEQ..."
Test #16: "LNKGKSYIPHMCVDKLDNDVFCIFGEWQGUJYJICUYIVXARMUCBNUWM..."

```



```

Test #17: "WGNRHKIQZMOPBQTCRYPSEPWHLRDXZMJOUTJCLECKEZZRRMQRNI..."
Test #18: "PPVTELDHJRZFPBNMJRLAZWRXRQVKHUUMRPNFKXJCUKFOXAGEHM..."
Test #19: "UXUIGAYKGLYUQTFBWQUTFNSOPEGMIWMQYEZAVCALGOHUXJZPTY..."
Test #20: "JSYTDGLVLBCVVSITPTQPHBCYIZHKFOFMBWOZNFKCADHDKPJSJA..."
Transposition cipher test passed.

```

Our testing program works by importing the *transpositionEncrypt.py* and *transpositionDecrypt.py* programs as modules. This way, we can call the `encryptMessage()` and `decryptMessage()` functions in these programs. Our testing program will create a random message and choose a random key. It doesn't matter that the message is just random letters, we just need to check that encrypting and then decrypting the message will result in the original message.

Our program will repeat this test twenty times by putting this code in a loop. If at any point the returned string from `transpositionDecrypt()` is not the exact same as the original message, our program will print an error message and exit.

How the Program Works

```

                                                                    transpositionTest.py
1. # Transposition Cipher Test
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import random, sys, transpositionEncrypt, transpositionDecrypt
5.
6. def main():

```

First our program imports two modules that come with Python, `random` and `sys`. We also want to import the transposition cipher programs we've written: *transpositionEncrypt.py* and *transpositionDecrypt.py*. Note that we don't put the `.py` extension in our `import` statement.

Pseudorandom Numbers and the `random.seed()` Function

```

                                                                    transpositionTest.py
7.     random.seed(42) # set the random "seed" to a static value

```

Technically, the numbers produced by Python's `random.randint()` function are not really random. They are produced from a pseudorandom number generator algorithm, and this algorithm is well known and the numbers it produces are predictable. We call these random-looking (but predictable) numbers **pseudorandom numbers** because they are not truly random.

The pseudorandom number generator algorithm starts with an initial number called the **seed**. All of the random numbers produced from a seed are predictable. You can reset Python’s random seed by calling the `random.seed()` function. Type the following into the interactive shell:

```
>>> import random
>>> random.seed(42)
>>> for i in range(5):
...     print(random.randint(1, 10))
...
7
1
3
3
8
>>> random.seed(42)
>>> for i in range(5):
...     print(random.randint(1, 10))
...
7
1
3
3
8
>>>
```

When the seed for Python’s pseudorandom number generator is set to 42, the first “random” number between 1 and 10 will **always** be 7. The second “random” number will **always** be 1, and the third number will **always** be 3, and so on. When we reset the seed back to 42 again, the same set of pseudorandom numbers will be returned from `random.randint()`.

Setting the random seed by calling `random.seed()` will be useful for our testing program, because we want predictable numbers so that the same pseudorandom messages and keys are chosen each time we run the automated testing program. Our Python programs only seem to generate “unpredictable” random numbers because the seed is set to the computer’s current clock time (specifically, the number of seconds since January 1st, 1970) when the `random` module is first imported.

It is important to note that not using truly random numbers is a common security flaw of encryption software. If the “random” numbers in your programs can be predicted, then this can provide a cryptanalyst with a useful hint to breaking your cipher. More information about generating truly random numbers with Python using the `os.urandom()` function can be found at <http://invy.com/random>.

The `random.randint()` Function

```

transpositionTest.py
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)

```

The code that does a single test will be in this `for` loop's block. We want this program to run multiple tests since the more tests we try, the more certain that we know our programs work.

Line 13 creates a random message from the uppercase letters and stores it in the `message` variable. Line 13 uses string replication to create messages of different lengths. The `random.randint()` function takes two integer arguments and returns a random integer between those two integers (including the integers themselves). Type the following into the interactive shell:

```

>>> import random
>>> random.randint(1, 20)
20
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(100, 200)
107
>>>

```

Of course, since these are pseudorandom numbers, the numbers you get will probably be different than the ones above. Line 13 creates a random message from the uppercase letters and stores it in the `message` variable. Line 13 uses string replication to create messages of different lengths.

References

Technically, variables do not store list values in them. Instead, they store reference values to list values. Up until now the difference hasn't been important. But storing list references instead of lists becomes important if you copy a variable with a list reference to another variable. Try entering the following into the shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
>>>
```

This makes sense from what we know so far. We assign 42 to the `spam` variable, and then we copy the value in `spam` and assign it to the variable `cheese`. When we later change the value in `spam` to 100, this doesn't affect the value in `cheese`. This is because `spam` and `cheese` are different variables that each store their own values.

But lists don't work this way. When you assign a list to a variable with the `=` sign, you are actually assigning a list reference to the variable. A **reference** is a value that points to some bit of data, and a **list reference** is a value that points to a list. Here is some code that will make this easier to understand. Type this into the shell:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

This looks odd. The code only changed the `cheese` list, but it seems that both the `cheese` and `spam` lists have changed.

Notice that the line `cheese = spam` copies the list *reference* in `spam` to `cheese`, instead of copying the list value itself. This is because the value stored in the `spam` variable is a list reference, and not the list value itself. This means that the values stored in both `spam` and `cheese` refer to the same list. There is only one list because the list was not copied, the reference to the list was copied. So when you modify `cheese` in the `cheese[1] = 'Hello!'` line, you are modifying the same list that `spam` refers to. This is why `spam` seems to have the same list value that `cheese` does.

Remember that variables are like boxes that contain values. List variables don't actually contain lists at all, they contain references to lists. Here are some pictures that explain what happens in the code you just typed in:

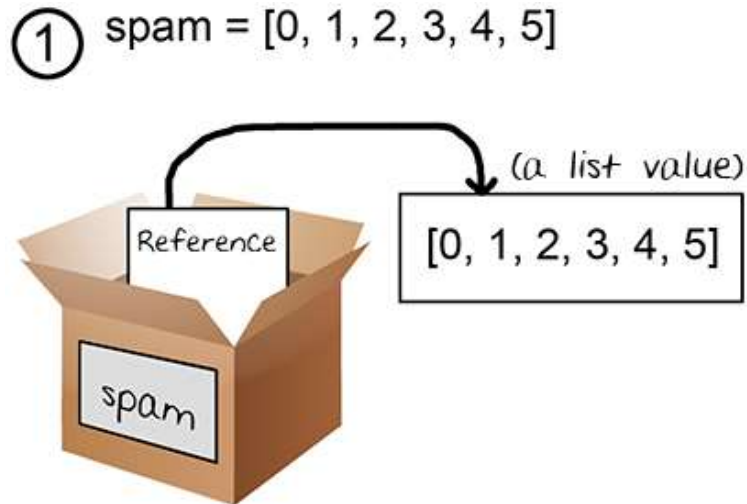


Figure 10-1. Variables do not store lists, but rather references to lists.

On the first line, the actual list is not contained in the `spam` variable but a reference to the list. The list itself is not stored in any variable.

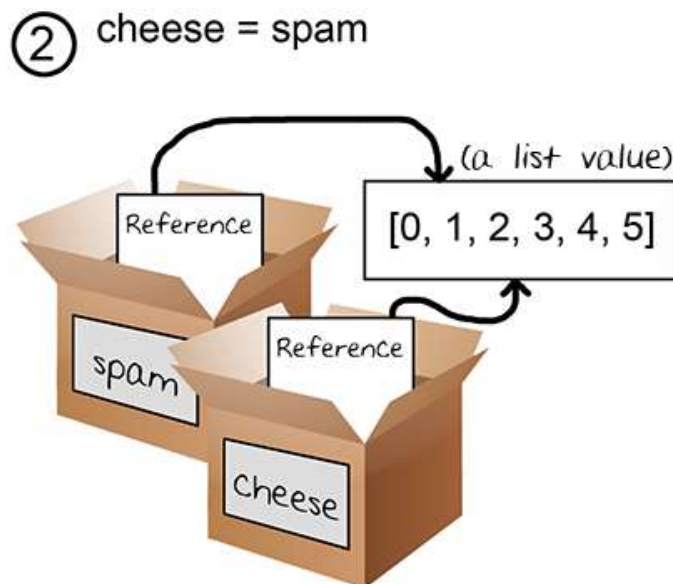


Figure 10-2. Two variables store two references to the same list.

When you assign the reference in `spam` to `cheese`, the `cheese` variable contains a copy of the reference in `spam`. Now both `cheese` and `spam` refer to the same list.

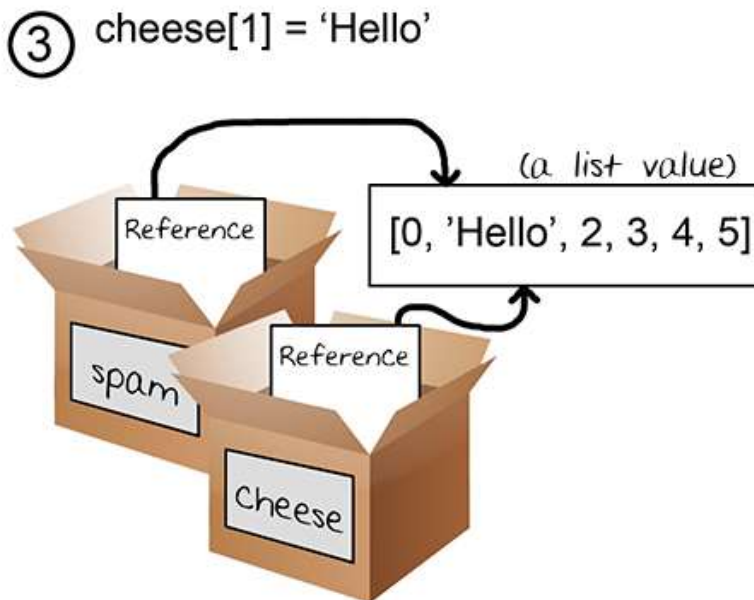


Figure 10-3. Changing the list changes all variables with references to that list.

When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed because they refer to the same list. If you want `spam` and `cheese` to store two different lists, you have to create two different lists instead of copying a reference:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = [0, 1, 2, 3, 4, 5]
```

In the above example, `spam` and `cheese` have two different lists stored in them (even though these lists are identical in content). Now if you modify one of the lists, it will not affect the other because `spam` and `cheese` have references to two different lists:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = [0, 1, 2, 3, 4, 5]
>>> cheese[1] = 'Hello!'
>>> spam
[0, 1, 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Figure 10-4 shows how the two references point to two different lists:

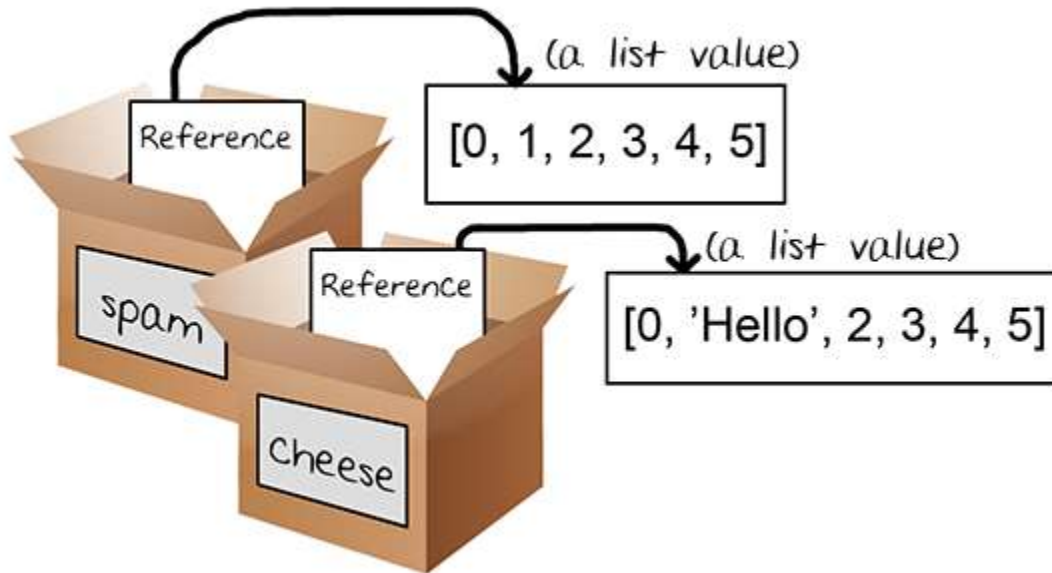


Figure 10-4. Two variables each storing references to two different lists.

The `copy` . `deepcopy` () Functions

As we saw in the previous example, the following code only copies the reference value, not the list value itself:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam    # copies the reference, not the list
```

If we want to copy the list value itself, we can import the `copy` module to call the `copy.deepcopy()` function, which will return 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]
>>>
```

The `copy.deepcopy()` function isn't used in this chapter's program, but it is helpful when you need to make a duplicate list value to store in a different variable.

Practice Exercises, Chapter 10, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice10A>.

The `random.shuffle()` Function

The `random.shuffle()` function is also in the `random` module. It accepts a list argument, and then randomly rearranges items in the list. Type the following into the interactive shell:

```
>>> 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 thing 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.) We say that the `shuffle()` function modifies the list **in-place**. This is why we only need to execute `random.shuffle(spam)` instead of `spam = random.shuffle(spam)`.

Remember that you can use the `list()` function to convert a string or range object to a list value. Type the following into the interactive shell:

```
>>> import random
>>> eggs = list('Hello')
>>> eggs
['H', 'e', 'l', 'l', 'o']
>>> random.shuffle(eggs)
>>> eggs
['o', 'H', 'l', 'l', 'e']
>>>
```


And also remember you can use the `join()` string method to pass a list of strings and return a single string:

```
>>> eggs
['o', 'H', 'l', 'l', 'e']
>>> eggs = ''.join(eggs)
>>> eggs
'oHlle'
>>>
```

Randomly Scrambling a String

```
transpositionTest.py
15.         # Convert the message string to a list to shuffle it.
16.         message = list(message)
17.         random.shuffle(message)
18.         message = ''.join(message) # convert list to string
```

In order to shuffle the characters in a string value, first we convert the string to a list with `list()`, then shuffle the items in the list with `shuffle()`, and then convert back to string value with the `join()` string method. Try typing the following into the interactive shell:

```
>>> import random
>>> spam = 'Hello world!'
>>> spam = list(spam)
>>> random.shuffle(spam)
>>> spam = ''.join(spam)
>>> spam
'wl de!Ho!orl'
>>>
```

We use this technique to scramble the letters in the `message` variable. This way we can test many different messages just in case our transposition cipher can encrypt and decrypt some messages but not others.

Back to the Code

```
transpositionTest.py
20.         print('Test #%s: "%s..." % (i+1, message[:50]))
```

Line 20 has a `print()` call that displays which test number we are on (we add one to `i` because `i` starts at 0 and we want the test numbers to start at 1). Since the string in `message` can be very long, we use string slicing to show only the first 50 characters of `message`.

Line 20 uses string interpolation. The value that `i+1` evaluates to will replace the first `%s` in the string and the value that `message[:50]` evaluates to will replace the second `%s`. When using string interpolation, be sure the number of `%s` in the string matches the number of values that are in between the parentheses after it.

```

transpositionTest.py
22.         # Check all possible keys for each message.
23.         for key in range(1, len(message)):
```

While the key for the Caesar cipher could be an integer from 0 to 25, the key for the transposition cipher can be between 1 and the length of the message. We want to test every possible key for the test message, so the `for` loop on line 23 will run the test code with the keys 1 up to (but not including) the length of the message.

```

transpositionTest.py
24.         encrypted = transpositionEncrypt.encryptMessage(key, message)
25.         decrypted = transpositionDecrypt.decryptMessage(key, encrypted)
```

Line 24 encrypts the string in `message` using our `encryptMessage()` function. Since 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 use the same key for both function calls. The return value from `decryptMessage()` is stored in a variable named `decrypted`. If the functions worked, then the string in `message` should be the exact same as the string in `decrypted`.

The `sys.exit()` Function

```

transpositionTest.py
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)
32.             sys.exit()
33.
34.     print('Transposition cipher test passed.')
```

Line 29 tests if `message` and `decrypted` are equal. If they aren't, we want to display an error message on the screen. We print the `key`, `message`, and `decrypted` values. This information could help us figure out what happened. Then we will exit the program.

Normally our programs exit once the execution reaches the very bottom and there are no more lines to execute. However, we can make the program exit sooner than that by calling the `sys.exit()` function. When `sys.exit()` is called, the program will immediately end.

But if the values in `message` and `decrypted` are equal to each other, the program execution skips the `if` statement's block and the call to `sys.exit()`. The next line is on line 34, but you can see from its indentation that it is the first line after line 9's `for` loop.

This means that after line 29's `if` statement's block, the program execution will jump back to line 23's `for` loop for the next iteration of that loop. If it has finished looping, then instead the execution jumps back to line 9's `for` loop for the next iteration of that loop. And if it has finished looping for that loop, then it continues on to line 34 to print out the string `'Transposition cipher test passed.'`

```

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

Here we do the trick of checking if the special variable `__name__` is set to `'__main__'` and if so, calling the `main()` function. This way, if another program imports `transpositionTest.py`, the code inside `main()` will not be executed but the `def` statements that create the `main()` function will be.

Testing Our Test Program

We've written a test program that tests our encryption programs, but how do we know that the test program works? What if there is a bug with our test program, and it is just saying that our transposition cipher programs work when they really don't?

We can test our test program by purposefully adding bugs to our encryption or decryption functions. Then when we run the test program, if it does not detect a problem with our cipher program, then we know that the test program is not correctly testing our cipher programs.

Change `transpositionEncrypt.py`'s line 36 from this:

```

                                                                    transpositionEncrypt.py
```

```

35.         # move pointer over
36.         pointer += key

```

...to this:

```

35.         # move pointer over
36.         pointer += key + 1

```

transpositionEncrypt.py

Now that the encryption code is broken, when we run the test program it should give us an error:

```

Test #1: "JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQ..."
Mismatch with key 1 and message
JEQLDFKJZWALCOYACUPLTRRMLWHOBXQNEAWSLGWAGQQSRSIUIQTRGJHDVCZECRESZJARAVIPFOBWZXX
TBFOFHVSIGBWIBBHGKUWHEUUDYONYTZVKNVVTYZPDDMIDKBHTYJAHBNDVJUZDCEMFMLUXEONCZXWAWG
XZSFTMJNLJOKKIJXLWAPQNYCIQOFTEAUHRJODKLGRIZSJXBQPBMPFPGMVUZHKFVPGNMRYXROMSCEE
XLUSCFHNELYPYKCNTOUQGBFSRDDMVIGXNYPHVPQISTATKVKM.
JQDKZACYCPTRLHBQEWLWQRIITGHVZCEZAAIFBZXBOHSGWBHKWEUYNTVNVYPDIKHYABDJZCMMUENZWW
XSTJLOKJLACNCQFEUROKGISBQBQPGVZKMGMYRMCELSFNLPKNTUGFRDVGPNPVQSAKK

```

Summary

We can use our programming skills for more than just writing programs. We can also program the computer to test those programs to make sure they work for different inputs. It is a common practice to write code to test code.

This chapter covered a few new functions such as the `random.randint()` function for producing pseudorandom numbers. Remember, pseudorandom numbers aren't random enough for cryptography programs, but they are good enough for this chapter's testing program. The `random.shuffle()` function is useful for scrambling the order of items in a list value.

The `copy.deepcopy()` function will create copies of list values instead of reference values. The difference between a list and list reference is explained in this chapter as well.

All of our programs so far have only encrypted short messages. In the next chapter, we will learn how to encrypt and decrypt entire files on your hard drive.



ENCRYPTING AND DECRYPTING FILES

Topics Covered In This Chapter:

- Reading and writing files
- The `open()` function
- The `read()` file object method
- The `close()` file object method
- The `write()` file object method
- The `os.path.exists()` function
- The `startswith()` string method
- The `title()` string method
- The `time` module and `time.time()` function

“Why do security police grab people and torture them? To get their information. If you build an information management system that concentrates information from dozens of people, you’ve made that dozens of times more attractive. You’ve focused the repressive regime’s attention on the hard disk. And hard disks put up no resistance to torture. You need to give the hard disk a way to resist. That’s cryptography.”

Up until now our programs have only worked on small messages that we type directly into the source code as string values. The cipher program in this chapter will use the transposition cipher to encrypt and decrypt entire files, which can be millions of characters in size.

Plain Text Files

This program will encrypt and decrypt plain text files. These are the kind of files that only have text data and usually have the .txt file extension. Files from word processing programs that let you change the font, color, or size of the text do not produce plain text files. You can write your own text files using Notepad (on Windows), TextMate or TextEdit (on OS X), or gedit (on Linux) or a similar plain text editor program. You can even use IDLE's own file editor and save the files with a .txt extension instead of the usual .py extension.

For some samples, you can download the following text files from this book's website:

- <http://invpy.com/devilsdictionary.txt>
- <http://invpy.com/frankenstein.txt>
- <http://invpy.com/siddhartha.txt>
- <http://invpy.com/thetimemachine.txt>

These are text files of some books (that are now in the public domain, so it is perfectly legal to download them.) For example, download Mary Shelley's classic novel "Frankenstein" from <http://invpy.com/frankenstein.txt>. Double-click the file to open it in a text editor program. There are over 78,000 words in this text file! It would take some time to type this into our encryption program. But if it is in a file, the program can read the file and do the encryption in a couple seconds.

If you get an error that looks like "UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 148: character maps to <undefined>" then you are running the cipher program on a non-plain text file, also called a "binary file".

To find other public domain texts to download, go to the Project Gutenberg website at <http://www.gutenberg.org/>.

Source Code of the Transposition File Cipher Program

Like our transposition cipher testing program, the transposition cipher file program will import our *transpositionEncrypt.py* and *transpositionDecrypt.py* files so we can use the

`encryptMessage()` and `decryptMessage()` functions in them. This way we don't have to re-type the code for these functions in our new program.

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *transpositionFileCipher.py*. Press **F5** to run the program. Note that first you will need to download *frankenstein.txt* and place this file in the same directory as the *transpositionFileCipher.py* file. You can download this file from <http://invpy.com/frankenstein.txt>.

Source code for *transpositionFileCipher.py*

```

1. # Transposition Cipher Encrypt/Decrypt File
2. # http://inventwithpython.com/hacking (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'
13.
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()
18.
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()
25.
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()))
32.
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('Encryption time: %s seconds' % (myMode.title(), totalTime))
41.
42.     # Write out the translated message to the output file.
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
46.
47.     print('Done %sing %s (%s characters).' % (myMode, inputFilename,
len(content)))
48.     print('%sed file is %s.' % (myMode.title(), outputFilename))
49.
50.
51. # If transpositionCipherFile.py is run (instead of imported as a module)
52. # call the main() function.
53. if __name__ == '__main__':
54.     main()

```

In the directory that *frankenstein.txt* and *transpositionFileCipher.py* files are in, there will be a new file named *frankenstein.encrypted.txt* that contains the content of *frankenstein.txt* in encrypted form. If you double-click the file to open it, it should look something like this:

```

PtFiyedleo a arnvm t eneeGLchongnes Mmuyedlsu0#uiSHTGA r sy,n t ys
s nuaoGeL
sc7s,
(the rest has been cut out for brevity)

```

To decrypt, make the following changes to the source code (written in bold) and run the transposition cipher program again:

```

                                                                    transpositionFileCipher.py
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 will appear in the folder named *frankenstein.decrypted.txt* that is identical to the original *frankenstein.txt* file.

Sample Run of the Transposition File Cipher Program

When you run the above program, it produces 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 will have been created in the same directory as *transpositionFileCipher.py*. If you open this file with IDLE’s file editor, you will see the encrypted contents of *frankenstein.py*. You can now email this encrypted file to someone for them to decrypt.

Reading From Files

Up until now, any input we want to give our programs would have to be typed in by the user. Python programs can open and read files directly off of the hard drive. There are three steps to reading the contents of a file: opening the file, reading into a variable, and then closing the file.

The `open ()` Function and File Objects

The `open ()` function’s first parameter is a string for the name of the file to open. If the file is in the same directory as the Python program then you can just type in the name, such as `'thetimemachine.txt'`. You can always specify the **absolute path** of the file, which includes the directory that it is in. For example, `'c:\\Python32\\frankenstein.txt'` (on Windows) and `'/usr/foobar/frankenstein.txt'` (on OS X and Linux) are absolute filenames. (Remember that the `\` backslash must be escaped with another backslash before it.)

The `open ()` function returns a value of the “file object” data type. This value has several methods for reading from, writing to, and closing the file.

The `read ()` File Object Method

The `read ()` method will return a string containing all the text in the file. For example, say the file *spam.txt* contained the text “Hello world!”. (You can create this file yourself using IDLE’s file editor. Just save the file with a `.txt` extension.) Run the following from the interactive shell (this code assumes you are running Windows and the *spam.txt* file is in the `c:\` directory):

```
>>> fo = open('c:\\spam.txt', 'r')
>>> content = fo.read()
>>> print(content)
```

```
Hello world!
>>>
```

If your text file has multiple lines, the string returned by `read()` will have `\n` newline characters in it at the end of each line. When you try to print a string with newline characters, the string will print across several lines:

```
>>> print('Hello\nworld!')
Hello
world!
>>>
```

If you get an error message that says “`IOError: [Errno 2] No such file or directory`” then double check that you typed the filename (and if it is an absolute path, the directory name) correctly. Also make sure that the file actually is where you think it is.

The `close()` File Object Method

After you have read the file’s contents into a variable, you can tell Python that you are done with the file by calling the `close()` method on the file object.

```
>>> fo.close()
>>>
```

Python will automatically close any open files when the program terminates. But when you want to re-read the contents of a file, you must close the file object and then call the `open()` function on the file again.

Here’s the code in our transposition cipher program that reads the file whose filename is stored in the `inputFilename` variable:

```

26.     # Read in the message from the input file
27.     fileObj = open(inputFilename)
28.     content = fileObj.read()
29.     fileObj.close()

```

transpositionFileCipher.py

Writing To Files

We read the original file and now will write the encrypted (or decrypted) form to a different file. The file object returned by `open()` has a `write()` function, although you can only use this

function if you open the file in “write” mode instead of “read” mode. You do this by passing the string value 'w' as the second parameter. For example:

```
>>> fo = open('filename.txt', 'w')
>>>
```

Along with “read” and “write”, there is also an “append” mode. The “append” is like “write” mode, except any strings written to the file will be appended to the end of any content that is already in the file. “Append” mode will not overwrite the file if it already exists. To open a file in append mode, pass the string 'a' as the second argument to `open()`.

(Just in case you were curious, you could pass the string 'r' to `open()` to open the file in read mode. But since passing no second argument at all also opens the file in read mode, there’s no reason to pass 'r'.)

The `write()` File Object Method

You can write text to a file by calling the file object’s `write()` method. The file object must have been opened in write mode, otherwise, you will get a “`io.UnsupportedOperation: not readable`” error message. (And if you try to call `read()` on a file object that was opened in write mode, you will get a “`io.UnsupportedOperation: not readable`” error message.)

The `write()` method takes one argument: a string of text that is to be written to the file. Lines 43 to 45 open a file in write mode, write to the file, and then close the file object.

```

                                                                    transpositionFileCipher.py
42.     # Write out the translated message to the output file.
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
```

Now that we have the basics of reading and writing files, let’s look at the source code to the transposition file cipher program.

How the Program Works

```

                                                                    transpositionFileCipher.py
1. # Transposition Cipher Encrypt/Decrypt File
2. # http://inventwithpython.com/hacking (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 first part of the program should look familiar. Line 4 is an `import` statement for our *transpositionEncrypt.py* and *transpositionDecrypt.py* programs. It also imports the Python's `time`, `os`, and `sys` modules.

The `main()` function will be called after the `def` statements have been executed to define all the functions in the program. The `inputFilename` variable holds a string of the file to read, and the encrypted (or decrypted) text is written to the file with the name in `outputFilename`.

The transposition cipher uses an integer for a key, stored in `myKey`. If `'encrypt'` is stored in `myMode`, the program will encrypt the contents of the `inputFilename` file. If `'decrypt'` is stored in `myMode`, the contents of `inputFilename` will be decrypted.

The `os.path.exists()` Function

Reading files is always harmless, but we need to be careful when writing files. If we call the `open()` function in write mode with a filename that already exists, that file will first be deleted to make way for the new file. This means we could accidentally erase an important file if we pass the important file's name to the `open()` function. Using the `os.path.exists()` function, we can check if a file with a certain filename already exists.

The `os.path.exists()` file has a single string parameter for the filename, and returns `True` if this file already exists and `False` if it doesn't. The `os.path.exists()` function exists inside the `path` module, which itself exists inside the `os` module. But if we import the `os` module, the `path` module will be imported too.

Try typing the following into the interactive shell:

```
>>> import os
>>> os.path.exists('abcdef')
False
>>> os.path.exists('C:\\Windows\\System32\\calc.exe')
True
>>>
```

(Of course, you will only get the above results if you are running Python on Windows. The *calc.exe* file does not exist on OS X or Linux.)

```

                                                                    transpositionFileCipher.py
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()

```

We use the `os.path.exists()` function to check that the filename in `inputFilename` actually exists. Otherwise, we have no file to encrypt or decrypt. In that case, we display a message to the user and then quit the program.

The `startswith()` and `endswith()` String Methods

```

                                                                    transpositionFileCipher.py
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()

```

If the file the program will write to already exists, the user is asked to type in “C” if they want to continue running the program or “Q” to quit the program.

The string in the response variable will have `lower()` called on it, and the returned string from `lower()` will have the string method `startswith()` called on it. The `startswith()` method will return `True` if its string argument can be found at the beginning of the string. Try typing the following into the interactive shell:

```

>>> 'hello'.startswith('h')
True
>>> 'hello world!'.startswith('hello wo')
True
>>> 'hello'.startswith('H')
False
>>> spam = 'Albert'
>>> spam.startswith('Al')
True

```

```
>>>
```

On line 23, if the user did not type in 'c', 'continue', 'C', or another string that begins with C, then `sys.exit()` will be called to end the program. Technically, the user doesn't have to enter "Q" to quit; any string that does not begin with "C" will cause the `sys.exit()` function to be called to quit the program.

There is also an `endswith()` string method that can be used to check if a string value ends with another certain string value. Try typing the following into the interactive shell:

```
>>> 'Hello world!'.endswith('world!')
True
>>> 'Hello world!'.endswith('world')
False
>>>
```

The `title()` String Method

Just like the `lower()` and `upper()` string methods will return a string in lowercase or uppercase, the `title()` string method returns a string in "title case". Title case is where every word is uppercase for the first character and lowercase for the rest of the characters. Try typing the following into the interactive shell:

```
>>> 'hello'.title()
'Hello'
>>> 'HELLO'.title()
'Hello'
>>> 'hElLo'.title()
'Hello'
>>> 'hello world! HOW ARE YOU?'.title()
'Hello World! How Are You?'
>>> 'extra! extra! man bites shark!'.title()
'Extra! Extra! Man Bites Shark!'
>>>
```

```
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()))
```

transpositionFileCipher.py

Lines 27 to 29 open the file with the name stored in `inputFilename` and read in its contents into the `content` variable. On line 31, we display a message telling the user that the encryption or decryption has begun. Since `myMode` should either contain the string `'encrypt'` or `'decrypt'`, calling the `title()` string method will either display `'Encrypting...'` or `'Decrypting...'`.

The `time` Module and `time.time()` Function

All computers have a clock that keeps track of the current date and time. Your Python programs can access this clock by calling the `time.time()` function. (This is a function named `time()` that is in a module named `time`.)

The `time.time()` function will return a float value of the number of seconds since January 1st, 1970. This moment is called the **Unix Epoch**. Try typing the following into the interactive shell:

```
>>> import time
>>> time.time()
1349411356.892
>>> time.time()
1349411359.326
>>>
```

The float value shows that the `time.time()` function can be precise down to a **millisecond** (that is, 1/1,000 of a second). Of course, the numbers that `time.time()` displays for you will depend on the moment in time that you call this function. It might not be clear that 1349411356.892 is Thursday, October 4th, 2012 around 9:30 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 our program has been running.

```

                                                                    transpositionFileCipher.py
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('Encryption time: %s seconds' % (myMode.title(), totalTime))
```

We want to measure how long the encryption or decryption process takes for the contents of the file. Lines 35 to 38 call the `encryptMessage()` or `decryptMessage()` (depending on whether `'encrypt'` or `'decrypt'` is stored in the `myMode` variable). Before this code

however, we will call `time.time()` and store the current time in a variable named `startTime`.

On line 39 after the encryption or decryption function calls have returned, we will call `time.time()` again and subtract `startTime` from it. This will give us the number of seconds between the two calls to `time.time()`.

For example, if you subtract the floating point values returned when I called `time.time()` before in the interactive shell, you would get the amount of time in between those calls while I was typing:

```
>>> 1349411359.326 - 1349411356.892
2.434000015258789
>>>
```

(The difference Python calculated between the two floating point values is not precise due to rounding errors, which cause very slight inaccuracies when doing math with floats. For our programs, it will not matter. But you can read more about rounding errors at <http://invpy.com/rounding>.)

The `time.time() - startTime` expression evaluates to a value that is passed to the `round()` function which rounds to the nearest two decimal points. This value is stored in `totalTime`. On line 40, the amount of time is displayed to the user by calling `print()`.

Back to the Code

```

                                                                    transpositionFileCipher.py
42.     # Write out the translated message to the output file.
43.     outputFileObj = open(outputFilename, 'w')
44.     outputFileObj.write(translated)
45.     outputFileObj.close()
```

The encrypted (or decrypted) file contents are now stored in the `translated` variable. But this string will be forgotten when the program terminates, so we want to write the string out to a file to store it on the hard drive. The code on lines 43 to 45 do this by opening a new file (passing 'w' to `open()` to open the file in write mode) and then calling the `write()` file object method.

```

                                                                    transpositionFileCipher.py
47.     print('Done %sing %s (%s characters).' % (myMode, inputFilename,
len(content)))
48.     print('%sed file is %s.' % (myMode.title(), outputFilename))
```



```
49.  
50.  
51. # If transpositionCipherFile.py is run (instead of imported as a module)  
52. # call the main() function.  
53. if __name__ == '__main__':  
54.     main()
```

Afterwards, we print some more messages to the user telling them that the process is done and what the name of the written file is. Line 48 is the last line of the `main()` function.

Lines 53 and 54 (which get executed after the `def` statement on line 6 is executed) will call the `main()` function if this program is being run instead of being imported. (This is explained in Chapter 8's “The Special `__name__` Variable” section.)

Practice Exercises, Chapter 11, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice11A>.

Summary

Congratulations! There wasn't much to this new program aside from the `open()`, `write()`, `read()`, and `close()` functions, but this lets us encrypt text files on our hard drive that are megabytes or gigabytes in size. It doesn't take much new code because all of the implementation for the cipher has already been written. We can extend our programs (such as adding file reading and writing capabilities) by importing their functions for use in new programs. This greatly increases our ability to use computers to encrypt information.

There are too many possible keys to simply brute-force and examine the output of a message encrypted with the transposition cipher. But if we can write a program that recognizes English (as opposed to strings of gibberish), we can have the computer examine the output of thousands of decryption attempts and determine which key can successfully decrypt a message to English.



DETECTING ENGLISH PROGRAMMATICALLY

Topics Covered In This Chapter:

- Dictionaries
- The `split()` Method
- The `None` Value
- "Divide by Zero" Errors
- The `float()`, `int()`, and `str()` Functions and Python 2 Division
- The `append()` List Method
- Default Arguments
- Calculating Percentage

The gaffer says something longer and more complicated. After a while, Waterhouse (now wearing his cryptanalyst 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.

“Cryptonomicon” by Neal Stephenson

A message encrypted with the transposition cipher can have thousands of possible keys. Your computer can still easily brute-force this many keys, but you would then have to look through thousands of decryptions to find the one correct plaintext. This is a big problem for the brute-force method of cracking the transposition cipher.

When the computer decrypts a message with the wrong key, the resulting plaintext is garbage text. We need to program the computer to be able to recognize if the plaintext is garbage text or English text. That way, if the computer decrypts with the wrong key, it knows to go on and try the next possible key. And when the computer tries a key that decrypts to English text, it can stop and bring that key to the attention of the cryptanalyst. Now the cryptanalyst won't have to look through thousands of incorrect decryptions.

How Can a Computer Understand English?

It can't. At least, not in the way that human beings like you or I understand English. Computers don't really understand math, chess, or lethal military androids either, any more than a clock understands lunchtime. Computers just execute instructions one after another. But these instructions can mimic very complicated behaviors that solve math problems, win at chess, or hunt down the future leaders of the human resistance.

Ideally, what we need is a Python function (let's call it `isEnglish()`) that has a string passed to it and then returns `True` if the string is English text and `False` if it's random gibberish. Let's take a look at some English text and some garbage text and try to see what patterns the two have:

```
Robots are your friends. Except for RX-686. She will try to eat you.
```

```
ai-pey e. xrx ne augur iir16 Rtiyt fhubE6d hrSei t8..ow eo.telyoosEs t
```

One thing we can notice is that the English text is made up of words that you could find in a dictionary, but the garbage text is made up of words that you won't. Splitting up the string into individual words is easy. There is already a Python string method named `split()` that will do this for us (this method will be explained later). The `split()` method just sees when each word begins or ends by looking for the space characters. Once we have the individual words, we can test to see if each word is a word in the dictionary with code like this:

```
if word == 'aardvark' or word == 'abacus' or word == 'abandon' or word ==
'abandoned' or word == 'abbreviate' or word == 'abbreviation' or word ==
'abdomen' or ...
```

We *can* write code like that, but we probably shouldn't. The computer won't mind running through all this code, but you wouldn't want to type it all out. Besides, somebody else has already

typed out a text file full of nearly all English words. These text files are called **dictionary files**. So we just need to write a function that checks if the words in the string exist somewhere in that file.

Remember, a *dictionary file* is a text file that contains a large list of English words. A *dictionary value* is a Python value that has key-value pairs.

Not every word will exist in our “dictionary file”. Maybe the dictionary file is incomplete and doesn’t have the word, say, “aardvark”. There are also perfectly good decrypts that might have non-English words in them, such as “RX-686” in our above English sentence. (Or maybe the plaintext is in a different language besides English. But we’ll just assume it is in English for now.)

And garbage text might just happen to have an English word or two in it by coincidence. For example, it turns out the word “augur” means a person who tries to predict the future by studying the way birds are flying. Seriously.

So our function will not be foolproof. But if most of the words in the string argument are English words, it is a good bet to say that the string is English text. It is a very low probability that a ciphertext will decrypt to English if decrypted with the wrong key.

The dictionary text file will have one word per line in uppercase. It will look like this:

```
AARHUS
AARON
ABABA
ABACK
ABAFT
ABANDON
ABANDONED
ABANDONING
ABANDONMENT
ABANDONS
```

...and so on. You can download this entire file (which has over 45,000 words) from <http://invpy.com/dictionary.txt>.

Our `isEnglish()` function will have to split up a decrypted string into words, check if each word is in a file full of thousands of English words, and if a certain amount of the words are English words, then we will say that the text is in English. And if the text is in English, then there’s a good bet that we have decrypted the ciphertext with the correct key.

And that is how the computer can understand if a string is English or if it is gibberish.

Practice Exercises, Chapter 12, Section A

Practice exercises can be found at <http://invpy.com/hackingpractice12A>.

The Detect English Module

The *detectEnglish.py* program that we write in this chapter isn't a program that runs by itself. Instead, it will be imported by our encryption programs so that they can call the `detectEnglish.isEnglish()` function. This is why we don't give *detectEnglish.py* a `main()` function. The other functions in the program are all provided for `isEnglish()` to call.

Source Code for the Detect English Module

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *detectEnglish.py*. Press **F5** to run the program.

Source code for detectEnglish.py

```

1. # Detect English module
2. # http://inventwithpython.com/hacking (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 English
8. # words in it, one word per line. You can download this from
9. # http://invpy.com/dictionary.txt)
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

```

How the Program Works

```

                                                                    detectEnglish.py
1. # Detect English module
2. # http://inventwithpython.com/hacking (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 English
8. # words in it, one word per line. You can download this from
9. # http://invpy.com/dictionary.txt)

```

These comments at the top of the file give instructions to programmers on how to use this module. They give the important reminder that if there is no file named *dictionary.txt* in the same

directory as *detectEnglish.py* then this module will not work. If the user doesn't have this file, the comments tell them they can download it from <http://inropy.com/dictionary.txt>.

```

detectEnglish.py
10. UPPERLETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
11. LETTERS_AND_SPACE = UPPERLETTERS + UPPERLETTERS.lower() + ' \t\n'
```

Lines 10 and 11 set up a few variables that are constants, which is why they have uppercase names. `UPPERLETTERS` is a variable containing the 26 uppercase letters, and `LETTERS_AND_SPACE` contain these letters (and the lowercase letters returned from `UPPERLETTERS.lower()`) but also the space character, the tab character, and the newline character. The tab and newline characters are represented with escape characters `\t` and `\n`.

```

detectEnglish.py
13. def loadDictionary():
14.     dictionaryFile = open('dictionary.txt')
```

The dictionary file sits on the user's hard drive, but we need to load the text in this file as a string value so our Python code can use it. First, we get a file object by calling `open()` and passing the string of the filename `'dictionary.txt'`. Before we continue with the `loadDictionary()` code, let's learn about the dictionary data type.

Dictionaries and the Dictionary Data Type

The **dictionary** data type has values which can contain multiple other values, just like lists do. In list values, you use an integer index value to retrieve items in the list, like `spam[42]`. For each item in the dictionary value, there is a key used to retrieve it. (Values stored inside lists and dictionaries are also sometimes called items.) The key can be an integer or a string value, like `spam['hello']` or `spam[42]`. Dictionaries let us organize our program's data with even more flexibility than lists.

Instead of typing square brackets like list values, dictionary values (or simply, dictionaries) use curly braces. Try typing the following into the interactive shell:

```

>>> emptyList = []
>>> emptyDictionary = {}
>>>
```

A dictionary's values are typed out as key-value pairs, which are separated by colons. Multiple key-value pairs are separated by commas. To retrieve values from a dictionary, just use square

brackets with the key in between them (just like indexing with lists). Try typing the following into the interactive shell:

```
>>> spam = {'key1':'This is a value', 'key2':42}
>>> spam['key1']
'This is a value'
>>> spam['key2']
42
>>>
```

It is important to know that, just as with lists, variables do not store dictionary values themselves, but references to dictionaries. The example code below has two variables with references to the same dictionary:

```
>>> spam = {'hello': 42}
>>> eggs = spam
>>> eggs['hello'] = 99
>>> eggs
{'hello': 99}
>>> spam
{'hello': 99}
>>>
```

Adding or Changing Items in a Dictionary

You can add or change values in a dictionary with indexes as well. Try typing the following into the interactive shell:

```
>>> spam = {42:'hello'}
>>> print(spam[42])
hello
>>> spam[42] = 'goodbye'
>>> print(spam[42])
goodbye
>>>
```

And just like lists can contain other lists, dictionaries can also contain other dictionaries (or lists). Try typing 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']
```



```
'A1'
>>> foo['moo']
['a', 'brown', 'cow']
>>> foo['moo'][1]
'brown'
>>>
```

Practice Exercises, Chapter 12, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice12B>.

Using the len () Function with Dictionaries

The len () function can tell you how many items are in a list or how many characters are in a string, but it can also tell you how many items are in a dictionary as well. Try typing the following into the interactive shell:

```
>>> spam = {}
>>> len(spam)
0
>>> spam['name'] = 'A1'
>>> spam['pet'] = 'Zophie the cat'
>>> spam['age'] = 89
>>> len(spam)
3
>>>
```

Using the in Operator with Dictionaries

The in operator can also be used to see if a certain key value exists in a dictionary. It is important to remember that the in operator checks if a key exists in the dictionary, not a value. Try typing the following into the interactive shell:

```
>>> eggs = {'foo': 'milk', 'bar': 'bread'}
>>> 'foo' in eggs
True
>>> 'blah blah blah' in eggs
False
>>> 'milk' in eggs
False
>>> 'bar' in eggs
True
>>> 'bread' in eggs
False
```

```
>>>
```

The `not in` operator works with dictionary values as well.

Using `for` Loops with Dictionaries

You can also iterate over the keys in a dictionary with `for` loops, just like you can iterate over the items in a list. Try typing the following into the interactive shell:

```
>>> spam = {'name':'Al', 'age':99}
>>> for k in spam:
...     print(k)
...     print(spam[k])
...     print('=====')
...
age
99
=====
name
Al
=====
>>>
```

Practice Exercises, Chapter 12, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice12C>.

The Difference Between Dictionaries and Lists

Dictionaries are like lists in many ways, but there are a few important differences:

1. Dictionary items are not in any order. There is no “first” or “last” item in a dictionary like there is in a list.
2. Dictionaries do not have concatenation with the `+` operator. If you want to add a new item, you can just use indexing with a new key. For example, `foo['a new key'] = 'a string'`
3. Lists only have integer index values that range from 0 to the length of the list minus one. But dictionaries can have any key. If you have a dictionary stored in a variable `spam`, then you can store a value in `spam[3]` without needing values for `spam[0]`, `spam[1]`, or `spam[2]` first.

Finding Items is Faster with Dictionaries Than Lists

```
15. englishWords = {}
```

detectEnglish.py

In the `loadDictionary()` function, we will store all the words in the “dictionary file” (as in, a file that has all the words in an English dictionary book) in a dictionary value (as in, the Python data type.) The similar names are unfortunate, but they are two completely different things.

We could have also used a list to store the string values of each word from the dictionary file. The reason we use a dictionary is because the `in` operator works faster on dictionaries than lists. Imagine that we had the following list and dictionary values:

```
>>> listVal = ['spam', 'eggs', 'bacon']
>>> dictionaryVal = {'spam':0, 'eggs':0, 'bacon':0}
```

Python can evaluate the expression `'bacon' in dictionaryVal` a little bit faster than `'bacon' in listVal`. The reason is technical and you don’t need to know it for the purposes of this book (but you can read more about it at <http://invpy.com/listvsdict>). This faster speed doesn’t make that much of a difference for lists and dictionaries with only a few items in them like in the above example. But our `detectEnglish` module will have tens of thousands of items, and the expression `word in ENGLISH_WORDS` will be evaluated many times when the `isEnglish()` function is called. The speed difference really adds up for the `detectEnglish` module.

The `split()` Method

The `split()` string method returns a list of several strings. The “split” between each string occurs wherever a space is. For an example of how the `split()` string method works, try typing this into the 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. The 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 just a space. Try typing the following into the interactive shell:

```
>>> 'helloXXXworlXXXhowXXXareXXyou?'.split('XXX')
```

```
['hello', 'world', 'how', 'areXXyou?']  
>>>
```

```
16.     for word in dictionaryFile.read().split('\n'):
```

```
detectEnglish.py
```

Line 16 is a `for` loop that will set the `word` variable to each value in the list `dictionaryFile.read().split('\n')`. Let's break this expression down. `dictionaryFile` is the variable that stores the file object of the opened file. The `dictionaryFile.read()` method call will read the entire file and return it as a very large string value. On this string, we will call the `split()` method and split on newline characters. This `split()` call will return a list value made up of each word in the dictionary file (because the dictionary file has one word per line.)

This is why the expression `dictionaryFile.read().split('\n')` will evaluate to a list of string values. Since the dictionary text file has one word on each line, the strings in the list that `split()` returns will each have one word.

The None Value

`None` is a special value that you can assign to a variable. The **None value** represents the lack of a value. `None` is the only value of the data type `NoneType`. (Just like how the Boolean data type has only two values, the `NoneType` data type has only one value, `None`.) It can be very useful to use the `None` value when you need a value that means “does not exist”. The `None` value is always written without quotes and with a capital “N” and lowercase “one”.

For example, say you had a variable named `quizAnswer` which holds the user's answer to some True-False pop quiz question. You could set `quizAnswer` to `None` if the user skipped the question and did not answer it. Using `None` would be better because if you set it to `True` or `False` before assigning the value of the user's answer, it may look like the user gave an answer for the question even though they didn't.

Calls to functions that do not return anything (that is, they exit by reaching the end of the function and not from a `return` statement) will evaluate to `None`.

```
detectEnglish.py
```

```
17.         englishWords[word] = None
```

In our program, we only use a dictionary for the `englishWords` variable so that the `in` operator can find keys in it. We don't care what is stored for each key, so we will just use the `None` value. The `for` loop that starts on line 16 will iterate over each word in the dictionary file, and line 17 will use that word as a key in `englishWords` with `None` stored for that key.

Back to the Code

```
18.         dictionaryFile.close()
19.         return englishWords
```

detectEnglish.py

After the `for` loop finishes, the `englishWords` dictionary will have tens of thousands of keys in it. At this point, we close the file object since we are done reading from it and then return `englishWords`.

```
21. ENGLISH_WORDS = loadDictionary()
```

detectEnglish.py

Line 21 calls `loadDictionary()` and stores the dictionary value it returns in a variable named `ENGLISH_WORDS`. We want to call `loadDictionary()` before the rest of the code in the `detectEnglish` module, but Python has to execute the `def` statement for `loadDictionary()` before we can call the function. This is why the assignment for `ENGLISH_WORDS` comes after the `loadDictionary()` function's code.

```
24. def getEnglishCount(message):
25.     message = message.upper()
26.     message = removeNonLetters(message)
27.     possibleWords = message.split()
```

detectEnglish.py

The `getEnglishCount()` function will take one string argument and return a float value indicating the amount of recognized English words in it. The value `0.0` will mean none of the words in `message` are English words and `1.0` will mean all of the words in `message` are English words, but most likely `getEnglishCount()` will return something in between `0.0` and `1.0`. The `isEnglish()` function will use this return value as part of whether it returns `True` or `False`.

First we must create a list of individual word strings from the string in `message`. Line 25 will convert it to uppercase letters. Then line 26 will remove the non-letter characters from the string, such as numbers and punctuation, by calling `removeNonLetters()`. (We will see how this function works later.) Finally, the `split()` method on line 27 will split up the string into individual words that are stored in a variable named `possibleWords`.

So if the string `'Hello there. How are you?'` was passed when `getEnglishCount()` was called, the value stored in `possibleWords` after lines 25 to 27 execute would be `['HELLO', 'THERE', 'HOW', 'ARE', 'YOU']`.

```
detectEnglish.py
29.     if possibleWords == []:
30.         return 0.0 # no words at all, so return 0.0
```

If the string in `message` was something like `'12345'`, all of these non-letter characters would have been taken out of the string returned from `removeNonLetters()`. The call to `removeNonLetters()` would return the blank string, and when `split()` is called on the blank string, it will return an empty list.

Line 29 does a special check for this case, and returns `0.0`. This is done to avoid a “divide-by-zero” error (which is explained later on).

```
detectEnglish.py
32.     matches = 0
33.     for word in possibleWords:
34.         if word in ENGLISH_WORDS:
35.             matches += 1
```

The float value that is returned from `getEnglishCount()` ranges between `0.0` and `1.0`. To produce this number, we will divide the number of the words in `possibleWords` that are recognized as English by the total number of words in `possibleWords`.

The first part of this is to count the number of recognized English words in `possibleWords`, which is done on lines 32 to 35. The `matches` variable starts off as `0`. The `for` loop on line 33 will loop over each of the words in `possibleWords`, and checks if the word exists in the `ENGLISH_WORDS` dictionary. If it does, the value in `matches` is incremented on line 35.

Once the `for` loop has completed, the number of English words is stored in the `matches` variable. Note that technically this is only the number of words that are recognized as English because they existed in our dictionary text file. As far as the program is concerned, if the word exists in `dictionary.txt`, then it is a real English word. And if it doesn't exist in the dictionary file,

it is not an English word. We are relying on the dictionary file to be accurate and complete in order for the `detectEnglish` module to work correctly.

“Divide by Zero” Errors

```
36.     return float(matches) / len(possibleWords)
```

`detectEnglish.py`

Returning a float value between 0.0 and 1.0 is a simple matter of dividing the number of recognized words by the total number of words.

However, whenever we divide numbers using the `/` operator in Python, we should be careful not to cause a “divide-by-zero” error. In mathematics, dividing by zero has no meaning. If we try to get Python to do it, it will result in an error. Try typing the following into the interactive shell:

```
>>> 42 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    42 / 0
ZeroDivisionError: int division or modulo by zero
>>>
```

But a divide by zero can’t possibly happen on line 36. The only way it could is if `len(possibleWords)` evaluated to 0. And the only way that would be possible is if `possibleWords` were the empty list. However, our code on lines 29 and 30 specifically checks for this case and returns 0.0. So if `possibleWords` had been set to the empty list, the program execution would have never gotten past line 30 and line 36 would not cause a “divide-by-zero” error.

The `float()`, `int()`, and `str()` Functions and Integer

Division

```
36.     return float(matches) / len(possibleWords)
```

`detectEnglish.py`

The value stored in `matches` is an integer. However, we pass this integer to the `float()` function which returns a float version of that number. Try typing the following into the interactive shell:

```
>>> float(42)
42.0
```

```
>>>
```

The `int()` function returns an integer version of its argument, and the `str()` function returns a string. Try typing 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'
>>>
```

The `float()`, `int()`, and `str()` functions are helpful if you need a value's equivalent in a different data type. But you might be wondering why we pass `matches` to `float()` on line 36 in the first place.

The reason is to make our `detectEnglish` module work with Python 2. Python 2 will do integer division when both values in the division operation are integers. This means that the result will be rounded down. So using Python 2, `22 / 7` will evaluate to 3. However, if one of the values is a float, Python 2 will do regular division: `22.0 / 7` will evaluate to `3.142857142857143`. This is why line 36 calls `float()`. This is called making the code **backwards compatible** with previous versions.

Python 3 always does regular division no matter if the values are floats or ints.

Practice Exercises, Chapter 12, Set D

Practice exercises can be found at <http://invpy.com/hackingpractice12D>.

Back to the Code

```
39. def removeNonLetters(message):
40.     lettersOnly = []
41.     for symbol in message:
```

`detectEnglish.py`

The previously explained `getEnglishCount()` function calls the `removeNonLetters()` function to return a string that is the passed argument, except with all the numbers and punctuation characters removed.

The code in `removeNonLetters()` starts with a blank list and loops over each character in the `message` argument. If the character exists in the `LETTERS_AND_SPACE` string, then it is added to the end of the list. If the character is a number or punctuation mark, then it won't exist in the `LETTERS_AND_SPACE` string and won't be added to the list.

The `append()` List Method

```
42.         if symbol in LETTERS_AND_SPACE:
43.             lettersOnly.append(symbol)
```

detectEnglish.py

Line 42 checks if `symbol` (which is set to a single character on each iteration of line 41's `for` loop) exists in the `LETTERS_AND_SPACE` string. If it does, then it is added to the end of the `lettersOnly` list with the `append()` list method.

If you want to add a single value to the end of a list, you could put the value in its own list and then use list concatenation to add it. Try typing the following into the interactive shell, where the value 42 is added to the end of the list stored in `spam`:

```
>>> spam = [2, 3, 5, 7, 9, 11]
>>> spam
[2, 3, 5, 7, 9, 11]
>>> spam = spam + [42]
>>> spam
[2, 3, 5, 7, 9, 11, 42]
>>>
```

When we add a value to the end of a list, we say we are **appending** the value to the list. This is done with lists so frequently in Python that there is an `append()` list method which takes a single argument to append to the end of the list. Try typing the following into the shell:

```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
>>> eggs.append(42)
>>> eggs
['hovercraft', 'eels', 42]
>>>
```

For technical reasons, using the `append()` method is faster than putting a value in a list and adding it with the `+` operator. The `append()` method modifies the list in-place to include the new value. You should always prefer the `append()` method for adding values to the end of a list.

```
44.     return ''.join(lettersOnly)
```

```
detectEnglish.py
```

After line 41's `for` loop is done, only the letter and space characters are in the `lettersOnly` list. To make a single string value from this list of strings, we call the `join()` string method on a blank string. This will join the strings in `lettersOnly` together with a blank string (that is, nothing) between them. This string value is then returned as `removeNonLetters()`'s return value.

Default Arguments

```
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).
```

```
detectEnglish.py
```

The `isEnglish()` function will accept a string argument and return a Boolean value that indicates whether or not it is English text. But when you look at line 47, you can see it has three parameters. The second and third parameters (`wordPercentage` and `letterPercentage`) have equal signs and values next to them. These are called **default arguments**. Parameters that have default arguments are optional. If the function call does not pass an argument for these parameters, the default argument is used by default.

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. Table 12-1 shows function calls to `isEnglish()`, and what they are equivalent to:

Table 12-1. Function calls with and without default arguments.

Function Call	Equivalent To
<code>isEnglish('Hello')</code>	<code>isEnglish('Hello', 20, 85)</code>
<code>isEnglish('Hello', 50)</code>	<code>isEnglish('Hello', 50, 85)</code>
<code>isEnglish('Hello', 50, 60)</code>	<code>isEnglish('Hello', 50, 60)</code>
<code>isEnglish('Hello', letterPercentage=60)</code>	<code>isEnglish('Hello', 20, 60)</code>

When `isEnglish()` is called with no second and third argument, the function will require that 20% of the words in `message` are English words that exist in the dictionary text file and 85% of the characters in `message` are letters. These percentages work for detecting English in most cases. But sometimes a program calling `isEnglish()` will want looser or more restrictive thresholds. If so, a program can just pass arguments for `wordPercentage` and `letterPercentage` instead of using the default arguments.

Calculating Percentage

A percentage is a number between 0 and 100 that shows how much of something there is proportional to the total number of those things. In the string value `'Hello cat MOOSE fsdkl ewpin'` there are five “words” but only three of them are English words. **To calculate the percentage of English words, you divide the number of English words by the total number of words and multiply by 100.** The percentage of English words in `'Hello cat MOOSE fsdkl ewpin'` is $3 / 5 * 100$, which is 60.

Table 12-2 shows some percentage calculations:

Table 12-2. Some percentage calculations.

Number of English Words	Total Number of Words	English Words / Total	* 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% (meaning none of the words) and 100% (meaning all of the words). Our `isEnglish()` function will consider a string to be English if at least 20% of the words are English words that exist in the dictionary file and 85% of the characters in the string are letters (or spaces).

```

51.      wordsMatch = getEnglishCount(message) * 100 >= wordPercentage
detectEnglish.py
```

Line 51 calculates the percentage of recognized English words in `message` by passing `message` to `getEnglishCount()`, which does the division for us and returns a float between 0.0 and 1.0. To get a percentage from this float, we just have to multiply it by 100. If this number is greater than or equal to the `wordPercentage` parameter, then `True` is stored in

`wordsMatch`. (Remember, the `>=` comparison operator evaluates expressions to a Boolean value.) Otherwise, `False` is stored in `wordsMatch`.

```

                                                                    detectEnglish.py
52.     numLetters = len(removeNonLetters(message))
53.     messageLettersPercentage = float(numLetters) / len(message) * 100
54.     lettersMatch = messageLettersPercentage >= letterPercentage

```

Lines 52 to 54 calculate the percentage of letter characters in the `message` string. To determine the percentage of letter (and space) characters in `message`, our code must divide the number of letter characters by the total number of characters in `message`. Line 52 calls `removeNonLetters(message)`. This call will return a string that has the number and punctuation characters removed from the string. Passing this string to `len()` will return the number of letter and space characters that were in `message`. This integer is stored in the `numLetters` variable.

Line 53 determines the percentage of letters getting a float version of the integer in `numLetters` and dividing this by `len(message)`. The return value of `len(message)` will be the total number of characters in `message`. (The call to `float()` was made so that if the programmer who imports our `detectEnglish` module is running Python 2, the division done on line 53 will always be regular division instead of integer division.)

Line 54 checks if 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`.

```

                                                                    detectEnglish.py
55.     return wordsMatch and lettersMatch

```

We want `isEnglish()` to return `True` only if both the `wordsMatch` and `lettersMatch` variables contain `True`, so we put them in an expression with the `and` operator. If both the `wordsMatch` and `lettersMatch` variables are `True`, then `isEnglish()` will declare that the `message` argument is English and return `True`. Otherwise, `isEnglish()` will return `False`.

Practice Exercises, Chapter 12, Set E

Practice exercises can be found at <http://inropy.com/hackingpractice12E>.

Summary

The dictionary data type is useful because like a list it can contain multiple values. However unlike the list, we can index values in it with string values instead of only integers. Most of the things we can do with lists we can also do with dictionaries, such as pass it to `len()` or use the `in` and `not in` operators on it. In fact, using the `in` operator on a very large dictionary value executes much faster than using `in` on a very large list.

The `NoneType` data type is also a new data type introduced in this chapter. It only has one value: `None`. This value is very useful for representing a lack of a value.

We can convert values to other data types by using the `int()`, `float()`, and `str()` functions. This chapter brings up “divide-by-zero” errors, which we need to add code to check for and avoid. The `split()` string method can convert a single string value into a list value of many strings. The `split()` string method is sort of the reverse of the `join()` list method. The `append()` list method adds a value to the end of the list.

When we define functions, we can give some of the parameters “default arguments”. If no argument is passed for these parameters when the function is called, the default argument value is used instead. This can be a useful shortcut in our programs.

The transposition 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. A computer has no problem decrypting a message with thousands of different keys, but to hack this cipher, we need to write code that can determine if a string value is valid English or not.

Since this code will probably be useful in our other hacking programs, we will put it in its own module so it can be imported by any program that wants to call its `isEnglish()` function. All of the work we’ve done in this chapter is so that any program can do the following:

```
>>> import detectEnglish
>>> detectEnglish.isEnglish('Is this sentence English text?')
True
>>>
```

Now armed with code that can detect English, let’s move on to the next chapter and hack the transposition cipher!



HACKING THE TRANSPOSITION CIPHER

Topics Covered In This Chapter:

- Multi-line Strings with Triple Quotes
- The `strip()` String Method

To hack the transposition cipher, we will use a brute-force approach. Of the thousands of keys, the correct key is most likely that only one that will result in readable English. We developed English-detection code in the last chapter so the program can realize when it has found the correct key.

Source Code of the Transposition Cipher Hacker Program

Open a new file editor window by clicking on **File ▶ New Window**. Type in the following code into the file editor, and then save it as *transpositionHacker.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *transpositionHacker.py* file. You can download this file from <http://invpy.com/pyperclip.py>.

Source code for transpositionHacker.py

```
1. # Transposition Cipher Hacker
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import pyperclip, detectEnglish, transpositionDecrypt
5.
```

```

6. def main():
7.     # You might want to copy & paste this text from the source code at
8.     # http://invpy.com/transpositionHacker.py
9.     myMessage = """Cb b rssti aieih rooaopbrtnsceee er es no npfgcwu plri
ch nitaalr eiuengiteehb(e1 hilincegeoamn fubehgtarndcstudmd nM eu eacBoltaetee
oinebcdkyremdteghn.aa2r81a condari fmps" tad  l t oisn sit u1rnd stara nvhn fs
edbh ee,n e necrg6 8nmisv l nc muiftegiitm tutmg cm shSs9fcie ebintcaets h a
ihda cctrhe ele 107 aaoem waoaatdahretnhechaopnooeapece9etfncdbgsoeb uuteitgna.
rteoh add e,D7c1Etnpneehtn beete" evecoal lsfmcr1 iu1cifgo ai. slrchdnheev sh
meBd ies e9t)nh,htcnoeclrrh ,ide hmtlme. phealeM,toeifgn t e9yce da' eN eMp a
ffn Fc1o ge eohg dere.eec s nfap yox hla yon. lnnsreaBoa t,e eitsw il ulpbdfg
BRe bwlmprraio po droB wtinue r Pieno nc ayieeto'lulcih sfnc ownaSserbereiaSm
-eaiah, nrtrtgC maciiritvledastinideI nn rms iehn tsigaBmuoetctias rn"""
10.
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)
19.
20.
21. def hackTransposition(message):
22.     print('Hacking...')
23.
24.     # Python programs can be stopped at any time by pressing Ctrl-C (on
25.     # Windows) or Ctrl-D (on Mac and Linux)
26.     print('(Press Ctrl-C or Ctrl-D to quit at any time.)')
27.
28.     # brute-force by looping through every possible key
29.     for key in range(1, len(message)):
30.         print('Trying key #s...' % (key))
31.
32.         decryptedText = transpositionDecrypt.decryptMessage(key, message)
33.
34.         if detectEnglish.isEnglish(decryptedText):
35.             # Check with user to see if the decrypted key has been found.
36.             print()
37.             print('Possible encryption hack:')
38.             print('Key %s: %s' % (key, decryptedText[:100]))
39.             print()
40.             print('Enter D for done, or just press Enter to continue
hacking:')
41.             response = input('> ')

```



```

42.
43.         if response.strip().upper().startswith('D'):
44.             return decryptedText
45.
46.     return None
47.
48. if __name__ == '__main__':
49.     main()

```

Sample Run of the Transposition Breaker Program

When you run this program, the output will look this:

```

Hacking...
(Press Ctrl-C or Ctrl-D to quit at any time.)
Trying key #1...
Trying key #2...
Trying key #3...
Trying key #4...
Trying key #5...
Trying key #6...
Trying key #7...
Trying key #8...
Trying key #9...
Trying key #10...

Possible encryption hack:
Key 10: Charles Babbage, FRS (26 December 1791 - 18 October 1871) was an
English mathematician, philosopher,

Enter D for done, or just press Enter to continue hacking:
> D
Copying hacked message to clipboard:
Charles Babbage, FRS (26 December 1791 - 18 October 1871) was an English
mathematician, philosopher, inventor and mechanical engineer who originated the
concept of a programmable computer. Considered a "father of the computer",
Babbage is credited with inventing the first mechanical computer that
eventually led to more complex designs. Parts of his uncompleted mechanisms are
on display in the London Science Museum. In 1991, a perfectly functioning
difference engine was constructed from Babbage's original plans. Built to
tolerances achievable in the 19th century, the success of the finished engine
indicated that Babbage's machine would have worked. Nine years later, the
Science Museum completed the printer Babbage had designed for the difference
engine.

```

When the hacker program has found a likely correct decryption, it will pause and wait for the user to press “D” and then Enter. If the decryption is a false positive, the user can just press Enter and the program will continue to try other keys.

Run the program again and skip the correct decryption by just pressing Enter. The program assumes that it was not a correct decryption and continues brute-forcing through the other possible keys. Eventually the program runs through all the possible keys and then gives up, telling the user that it was unable to hack the ciphertext:

```
Trying key #757...
Trying key #758...
Trying key #759...
Trying key #760...
Trying key #761...
Failed to hack encryption.
```

How the Program Works

```
transpositionHacker.py
1. # Transposition Cipher Hacker
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import pyperclip, detectEnglish, transpositionDecrypt
```

The transposition hacker program is under 50 lines of code because much of it exists in other programs. Several modules are imported on line 4.

Multi-line Strings with Triple Quotes

```
transpositionHacker.py
6. def main():
7.     # You might want to copy & paste this text from the source code at
8.     # http://invpy.com/transpositionHacker.py
9.     myMessage = """Cb b rssti aieih rooaopbrtnsceee er es no npfgcwu plri
ch nitaalr eiuengiteehb(e1 hilincegeoamn fubehgtarndcstudmd nM eu eacBoltaetee
oinebcdkyremdteghn.aa2r81a condari fmps" tad l t oisn sit ulrnd stara nvhn fs
edbh ee,n e necrg6 8nmisv l nc muiftegiitm tutmg cm shSs9fcie ebintcaets h a
ihda cctrhe ele 107 aaoem waoatdahretnehechaopnooeapece9etfncdbgsoeb uuteitgna.
rteoh add e,D7c1Etnpneehtn beete" evecoal lsfmcr1 iulcifgo ai. slrchnheev sh
meBd ies e9t)nh,htcnoeclrrh ,ide hmtlme. phealeM,toeinfgn t e9yce da' eN eMp a
ffn Fc1o ge eohg dere.eec s nfap yox hla yon. lnnsreaBoa t,e eitsw il ulpbdfg
BRe bwlmprraio po droB wtinue r Pieno nc ayieeto'lulcih sfnc ownaSserbereiaSm
-eaiah, nrrttgcC maciiritvledastinideI nn rms iehn tsigaBmuoetcteias rn"""
```

The ciphertext to be hacked is stored in the `myMessage` variable. Line 9 has a string value that begins and ends with triple quotes. These strings do not have to have literal single and double quotes escaped inside of them. Triple quote strings are also called multi-line strings, because they can also contain actual newlines within them. Try typing 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,
Bob"""
>>> 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,
Bob
>>>
```

Notice that this string value can span over multiple lines. Everything after the opening triple quotes will be interpreted as part of the string until it reaches triple quotes ending it. Multi-line strings can either use three double quote characters or three single quote characters.

Multi-line strings are useful for putting very large strings into the source code for a program, which is why it is used on line 9 to store the ciphertext to be broken.

Back to the Code

```
11.         hackedMessage = hackTransposition(myMessage) transpositionHacker.py
```

The ciphertext hacking code exists inside the `hackTransposition()` function. This function takes one string argument: the encrypted ciphertext message to be broken. If the function can hack the ciphertext, it returns a string of the decrypted text. Otherwise, it returns the `None` value. This value is stored in the `hackedMessage` variable.

```
13.         if hackedMessage == None: transpositionHacker.py
14.             print('Failed to hack encryption.')
```

If `None` was stored in `hackedMessage`, the program prints that it was unable to break the encryption on the message.

```

transpositionHacker.py
15.     else:
16.         print('Copying hacked message to clipboard:')
17.         print(hackedMessage)
18.         pyperclip.copy(hackedMessage)

```

Otherwise, the text of the decrypted message is printed to the screen on line 17 and also copied to the clipboard on line 18.

```

transpositionHacker.py
21. def hackTransposition(message):
22.     print('Hacking...')
23.
24.     # Python programs can be stopped at any time by pressing Ctrl-C (on
25.     # Windows) or Ctrl-D (on Mac and Linux)
26.     print('(Press Ctrl-C or Ctrl-D to quit at any time.)')

```

Because there are many keys the program can go through, the program displays a message to the user telling her that the hacking has started. The `print()` call on line 26 also tells her that she can press Ctrl-C (on Windows) or Ctrl-D (on OS X and Linux) to exit the program at any point. (Pressing these keys will always exit a running Python program.)

```

transpositionHacker.py
28.     # brute-force by looping through every possible key
29.     for key in range(1, len(message)):
30.         print('Trying key #s...' % (key))

```

The range of possible keys for the transposition cipher is the integers between 1 and the length of the message. The `for` loop on line 29 will run the hacking part of the function with each of these keys.

To provide feedback to the user, the key that is being tested is printed to the string on line 30, using string interpolation to place the integer in `key` inside the `'Trying key #s...' % (key)` string.

```

transpositionHacker.py
32.         decryptedText = transpositionDecrypt.decryptMessage(key, message)

```

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.

```

                                                                    transpositionHacker.py
34.         if detectEnglish.isEnglish(decryptedText):
35.             # Check with user to see if the decrypted key has been found.
36.             print()
37.             print('Possible encryption hack:')
38.             print('Key %s: %s' % (key, decryptedText[:100]))
39.             print()
40.             print('Enter D for done, or just press Enter to continue
hacking:')
41.             response = input('> ')

```

The decrypted output in `decryptedText` will most likely only be English if the correct key was used (otherwise, it will appear to be random garbage). The string in `decryptedText` is passed to the `detectEnglish.isEnglish()` function we wrote in the last chapter.

But just because `detectEnglish.isEnglish()` returns `True` (making the program execution enter the block following the `if` statement on line 34) doesn't mean the program has found the correct key. It could be a “false positive”. To be sure, line 38 prints out the first 100 characters of the `decryptedText` string (by using the slice `decryptedText[:100]`) on the screen for the user to look at.

The program pauses when line 41 executes, waiting for the user to type something in either D or nothing before pressing Enter. This input is stored as a string in `response`.

The `strip()` String Method

The `strip()` string method returns a version of the string that has any whitespace at the beginning and end of the string stripped out. Try typing in the following into the interactive shell:

```

>>> '    Hello'.strip()
'Hello'
>>> 'Hello    '.strip()
'Hello'
>>> '    Hello World    '.strip()
'Hello World'
>>> 'Hello    x'.strip()
'Hello    x'
>>>

```

The `strip()` method can also have a string argument passed to it that tells the method which characters should be removed from the start and end of the string instead of removing whitespace.

The **whitespace characters** are the space character, the tab character, and the newline character. Try typing the following into the interactive shell:

```
>>> 'Helloxxxxx'.strip('x')
'Hello'
>>> 'aaaaaHELLOaa'.strip('a')
'HELLO'
>>> 'ababaHELLOab'.strip('ab')
'HELLO'
>>> 'abccabcbacbXYZabcXYZacccab'.strip('abc')
'XYZabcXYZ'
>>>
```

```

43.             if response.strip().upper().startswith('D'):
44.                 return decryptedText
transpositionHacker.py
```

The expression on line 43 used for the `if` statement’s condition lets the user have some flexibility with what has to be typed in. If the condition were `response == 'D'`, then the user would have to type in exactly “D” and nothing else in order to end the program.

If the user typed in 'd' or ' D' or 'Done' then the condition would be `False` and the program would continue. To avoid this, the string in `response` has any whitespace removed from the start or end with the call to `strip()`. Then the string that `response.strip()` evaluates to has the `upper()` method called on it. If the user typed in either “d” or “D”, the string returned from `upper()` will be 'D'. Little things like this make our programs easier for the user to use.

If the user has indicated that the decrypted string is correct, the decrypted text is returned from `hackTransposition()` on line 44.

```

46.         return None
transpositionHacker.py
```

Line 46 is the first line after the `for` loop that began on line 29. If the program execution reaches this point, it’s because the `return` statement on line 44 was never reached. That would only 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 has failed.

```

48. if __name__ == '__main__':
transpositionHacker.py
```

```
49.     main()
```

Lines 48 and 49 call the `main()` function if this program was run by itself, rather than imported by another program that wants to use its `hackTransposition()` function.

Practice Exercises, Chapter 13, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice13A>.

Summary

This chapter was short like the “Breaking the Caesar Cipher with the Brute-Force Technique” chapter because (also like that chapter) most of the code was already written in other programs. Our hacking program can import functions from these other programs by importing them as modules.

The `strip()` string method is useful for removing whitespace (or other) characters from the beginning or end of a string. If we use triple quotes, then a string value can span across multiple lines in our source code.

The `detectEnglish.py` program removes a lot of the work of inspecting the decrypted output to see if it’s English. This allows the brute-force technique to be applied to a cipher that can have thousands of keys.

Our programs are becoming more sophisticated. Before we learn the next cipher, we should learn how to use Python’s debugger tool to help us find bugs in our programs.



MODULAR ARITHMETIC WITH THE MULTIPLICATIVE AND AFFINE CIPHERS

Topics Covered In This Chapter:

- Modular Arithmetic
- “Mod” is “Remainder Of”(Sort Of)
- GCD: Greatest Common Divisor (aka Greatest Common Factor)
- Multiple Assignment Trick
- Euclid’s Algorithm for Finding the GCD of Two Numbers
- “Relatively Prime”
- The Multiplicative Cipher
- Finding Modular Inverses
- The `cryptomath` Module

“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

The multiplicative and affine ciphers are similar to the Caesar cipher, except instead of adding a key to a symbol's index in a string, these ciphers use multiplication. But before we learn how to encrypt and decrypt with these ciphers, we're going to need to learn a little math. This knowledge is also needed for the last cipher in this book, the RSA cipher.

Oh No Math!

Don't let it scare you that you need to learn some math. The principles here are easy to learn from pictures, and we'll see that they are directly useful in cryptography.

Math Oh Yeah!

That's more like it.

Modular Arithmetic (aka Clock Arithmetic)

This is a clock in which I've replaced the 12 with a 0. (I'm a programmer. I think it's weird that the day begins at 12 AM instead of 0 AM.) Ignore the hour, minute, and second hands. We just need to pay attention to the numbers.



Figure 14-1. A clock with a zero o'clock.

3 O'Clock + 5 Hours = 8 O'Clock

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. Think of the hour hand on the clock in Figure 14-1 starting at 3, and then moving 5 hours clockwise. It will end up at 8. This is one way we can double-check our math.

**10 O'Clock + 5 Hours = 3 O'Clock**

If the current time is 10 o'clock, what time will it be in 5 hours? If you add $10 + 5$, you get 15. But 15 o'clock doesn't make sense for clocks like the one to the right. It only goes up to 12. So to find out what time it will be, we subtract $15 - 12 = 3$. The answer is it will be 3 o'clock. (Whether or not it is 3 AM or 3PM depends on if the current time is 10 AM or 10 PM. But it doesn't matter for modular arithmetic.)



If you think of the hour hand as starting at 10 and then moving forward 5 hours, it will land on 3. So double-checking our math by moving the hour hand clockwise shows us that we are correct.

10 O'Clock + 200 Hours = 6 O'Clock

If the current time is 10 o'clock, what time will it be in 200 hours? $200 + 10 = 210$, and 210 is larger than 12. So we subtract $210 - 12 = 198$. But 198 is still larger than 12, so we subtract 12 again. $198 - 12 = 186$. If we keep subtracting 12 until the difference is less than 12, we end up with 6. If the current time is 10 o'clock, the time 200 hours later will be 6 o'clock.

If we wanted to double check our 10 o'clock + 200 hours math, we would keep moving the hour hand around and around the clock face. When we've moved the hour hand the 200th time, it will end up landing on 6.



The % Mod Operator

This sort of “wrap-around” arithmetic is called **modular arithmetic**. We say “fifteen mod twelve” is equal to 3. (Just like how “15 o’clock” mod twelve would be “3 o’clock.”) In Python, the mod operator is the % percent sign. Try typing the following into the interactive shell:

```
>>> 15 % 12
3
>>> 210 % 12
6
>>> 10 % 10
0
>>> 20 % 10
0
>>>
```

“Mod” is “Division Remainder”(Sort Of)

You can think of the mod operator as a “division remainder” operator. $21 \div 5 = 4$ remainder 1. And $21 \% 5 = 1$. This works pretty well for positive numbers, but not for negative numbers. $-21 \div 5 = -4$ remainder -1. But the result of a mod operation will never be negative. Instead, think of that -1 remainder as being the same as $5 - 1$, which comes to 4. This is exactly what $-21 \% 5$ evaluates to:

```
>>> -21 % 5
4
>>>
```

But for the purposes of cryptography in this book, we’ll only be modding positive numbers.

Practice Exercises, Chapter 14, Set A

Practice exercises can be found at <http://invpy.com/hackingpractice14A>.

GCD: Greatest Common Divisor (aka Greatest Common Factor)

Factors are the numbers that can be multiplied to produce a particular number. Look at this simple multiplication:

$$4 \times 6 = 24$$

In the above math problem, we say 4 and 6 are factors of 24. (Another name for factor is **divisor**.) The number 24 also has some other factors:

$$8 \times 3 = 24$$

$$12 \times 2 = 24$$

$$24 \times 1 = 24$$

From the above three math problems, we can see that 8 and 3 are also factors of 24, as are 12 and 2, and 24 and 1. So we can say the factors of 24 are: 1, 2, 3, 4, 6, 8, 12, and 24.

Let's look at the factors of 30:

$$1 \times 30 = 30$$

$$2 \times 15 = 30$$

$$3 \times 10 = 30$$

$$5 \times 6 = 30$$

So the factors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30. (Notice that any number will always have 1 and itself as factors.) If you look at the list of factors for 24 and 30, you can see that the factors that they have in common are 1, 2, 3, and 6. The greatest number of these is 6, so we call 6 the greatest common factor (or, more commonly, the greatest common divisor) of 24 and 30.

Visualize Factors and GCD with Cuisenaire Rods

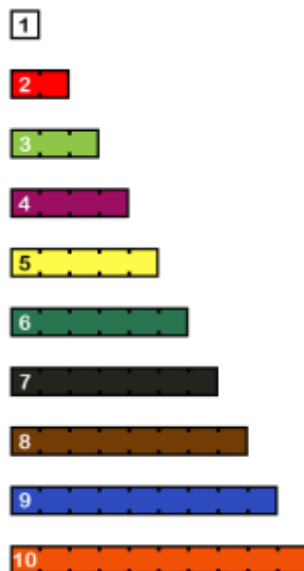


Figure 14-2. Each Cuisenaire rod has a different color for each integer length.

Above are some rectangular blocks with a width of 1 unit, 2 units, 3 units, and so on. The block's length can be used to represent a number. You can count the number of squares in each block to determine the length and number. These blocks (sometimes called Cuisenaire rods) can be used to visualize math operations, like $3 + 2 = 5$ or $5 \times 3 = 15$:

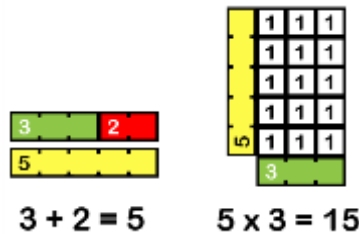


Figure 14-3. Using Cuisenaire rods to demonstrate addition and multiplication.

If we represent the number 30 as a block that is 30 units long, a number is a factor of 30 if the number's blocks can evenly fit with the 30-block. You can see that 3 and 10 are factors of 30:

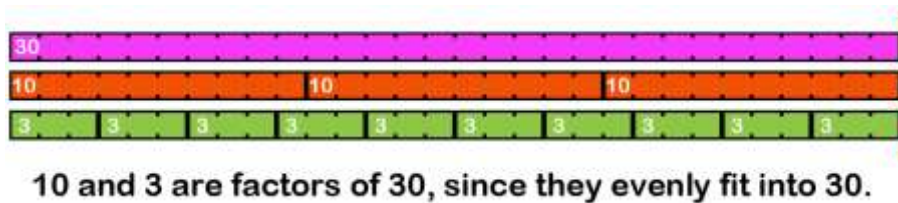


Figure 14-4. Cuisenaire rods demonstrating factors.

But 4 and 7 are not factors of 30, because the 4-blocks and 7-blocks won't evenly fit into the 30-block:

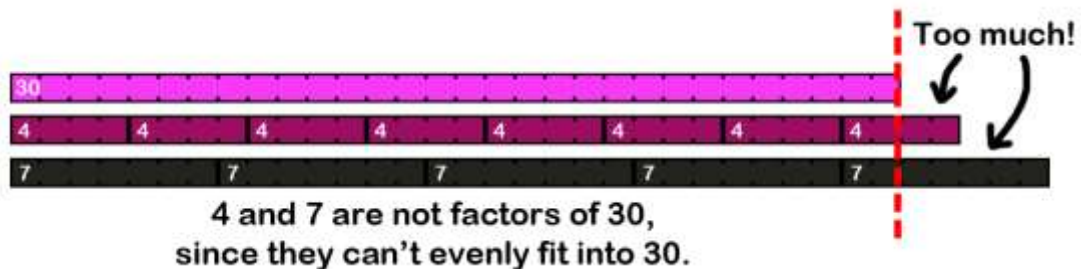


Figure 14-5. Cuisenaire rods demonstrating numbers that are not factors of 30.

The Greatest Common Divisor of two blocks (that is, two numbers represented by those blocks) is the **longest** block that can evenly fit **both** blocks.

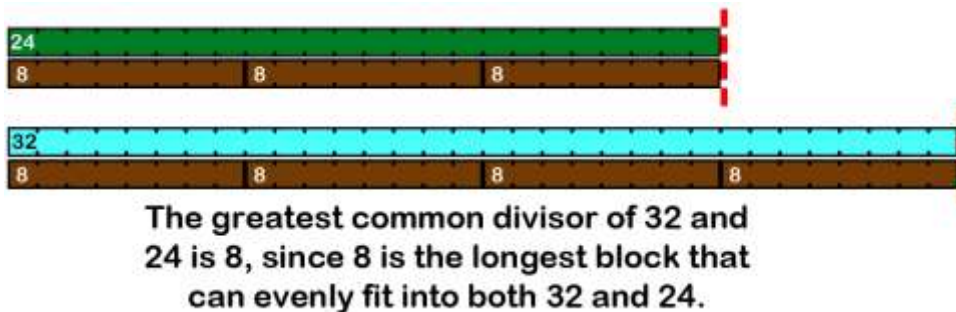


Figure 14-6. Cuisenaire rods demonstrating Greatest Common Divisor.

More information about Cuisenaire rods can be found at <http://invpy.com/cuisenaire>.

Practice Exercises, Chapter 14, Set B

Practice exercises can be found at <http://invpy.com/hackingpractice14B>.

Multiple Assignment

Our GCD function will use Python's multiple assignment trick. The multiple assignment trick lets you assign more than one variable with a single assignment statement. Try typing the following into the interactive shell:

```
>>> spam, eggs = 42, 'Hello'
>>> spam
42
>>> eggs
'Hello'
>>> a, b, c, d = ['Alice', 'Bob', 'Carol', 'David']
>>> a
'Alice'
>>> b
'Bob'
>>> c
'Carol'
>>> d
'David'
>>>
```

The variable names on the left side of the = operator and the values on the right side of the = operator are separated by a comma. You can also assign each of the values in a list to its own variable, if the number of items in the list is the same as the number of variables on the left side of the = operator.

Be sure to have the same number of variables as you have values, otherwise Python will raise an error that says the call needs more or has too many values:

```
>>> a, b, c = 1, 2
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a, b, c = 1, 2
ValueError: need more than 2 values to unpack

>>> a, b, c = 1, 2, 3, 4, 5, 6
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a, b, c = 1, 2, 3, 4, 5, 6
ValueError: too many values to unpack
>>>
```

Swapping Values with the Multiple Assignment Trick

One of the main uses of the multiple assignment trick is to swap the values in two variables. Try typing the following into the interactive shell:

```
>>> spam = 'hello'
>>> eggs = 'goodbye'
>>> spam, eggs = eggs, spam
>>> spam
'goodbye'
>>> eggs
'hello'
```

We will use this swapping trick in our implementation of Euclid's algorithm.

Euclid's Algorithm for Finding the GCD of Two Numbers

Figuring out the GCD of two numbers will be important for doing the multiplicative and affine ciphers. It seems simple enough: just look at the numbers and write down any factors you can think of, then compare the lists and find the largest number that is in both of them.

But to program a computer to do it, we'll need to be more precise. We need an algorithm (that is, a specific series of steps we execute) to find the GCD of two numbers.

A mathematician who lived 2,000 years ago named Euclid came up with an algorithm for finding the greatest common divisor of two numbers. Here's a statue of Euclid at Oxford University:



Figure 14-7. Euclid may or may not have looked like this.

Of course since no likeness or description of Euclid exists in any historical document, no one knows what he actually looked like at all. (Artists and sculptors just make it up.) This statue could also be called, “Statue of Some Guy with a Beard”.

Euclid’s GCD algorithm is short. Here’s a function that implements his algorithm as Python code, which returns the GCD of integers *a* and *b*:

```
def gcd(a, b):  
    while a != 0:  
        a, b = b % a, a  
    return b
```

If you call this function from the interactive shell and pass it 24 and 30 for the *a* and *b* parameters, the function will return 6. You could have done this yourself with pencil and paper. But since you’ve programmed a computer to do this, it can easily handle very large numbers:

```
>>> gcd(24, 30)  
6  
>>> gcd(409119243, 87780243)  
6837  
>>>
```

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.

“Relatively Prime”

Relatively prime numbers are used for the multiplicative and affine ciphers. We say that two numbers are relatively prime if their greatest common divisor is 1. That is, the numbers a and b are relatively prime to each other if $\text{gcd}(a, b) == 1$.

Practice Exercises, Chapter 14, Set C

Practice exercises can be found at <http://invpy.com/hackingpractice14C>.

The Multiplicative Cipher

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.

What if instead of adding the key to do the encryption, we use multiplication? There would be a “wrap-around” issue, but the mod operator would solve that. For example, let’s use the symbol set of just uppercase letters and the key 7. Here’s a list of the letters and their numbers:

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

To find what the symbol F encrypts to with key 7, multiply its number (5) by 7 and mod by 26 (to handle the “wrap-around” with our 26-symbol set). Then use that number’s symbol. $(5 \times 7) \bmod 26 = 9$, and 9 is the number for the symbol J. So F encrypts to J in the multiplicative cipher with key 7. Do the same with all of the letters:

Table 14-1. Encrypting each letter with the multiplicative cipher with key 7.

Plaintext Symbol	Number	Encryption with Key 7	Ciphertext Symbol
A	0	$(0 * 7) \% 26 = 0$	A
B	1	$(1 * 7) \% 26 = 7$	H
C	2	$(2 * 7) \% 26 = 14$	O
D	3	$(3 * 7) \% 26 = 21$	V
E	4	$(4 * 7) \% 26 = 2$	C
F	5	$(5 * 7) \% 26 = 9$	J
G	6	$(6 * 7) \% 26 = 16$	Q
H	7	$(7 * 7) \% 26 = 23$	X
I	8	$(8 * 7) \% 26 = 4$	E
J	9	$(9 * 7) \% 26 = 11$	L
K	10	$(10 * 7) \% 26 = 18$	S
L	11	$(11 * 7) \% 26 = 25$	Y
M	12	$(12 * 7) \% 26 = 6$	G
N	13	$(13 * 7) \% 26 = 13$	N
O	14	$(14 * 7) \% 26 = 20$	U
P	15	$(15 * 7) \% 26 = 1$	B
Q	16	$(16 * 7) \% 26 = 8$	I
R	17	$(17 * 7) \% 26 = 15$	P
S	18	$(18 * 7) \% 26 = 22$	W
T	19	$(19 * 7) \% 26 = 3$	D
U	20	$(20 * 7) \% 26 = 10$	K
V	21	$(21 * 7) \% 26 = 17$	R
W	22	$(22 * 7) \% 26 = 24$	Y
X	23	$(23 * 7) \% 26 = 5$	F
Y	24	$(24 * 7) \% 26 = 12$	M
Z	25	$(25 * 7) \% 26 = 19$	T

You will end up with this mapping for the key 7: to encrypt you replace the top letter with the letter under it, and vice versa to decrypt:

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
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
A	H	O	V	C	J	Q	X	E	L	S	Y	G	N	U	B	I	P	W	D	K	R	Y	F	M	T

It wouldn't take long for an attacker to brute-force through the first 7 keys. But the good thing about the multiplicative cipher is that it can work with very large keys, like 8,953,851 (which has the letters of the alphabet map to the letters AXUROLIFCZWTQNKHEBYVSPMJGD). It would take quite some time for a computer to brute-force through nearly nine million keys.

Practice Exercises, Chapter 14, Set D

Practice exercises can be found at <http://invpy.com/hackingpractice14D>.

Multiplicative Cipher + Caesar Cipher = The Affine Cipher

One downside to the multiplicative cipher is that the letter A always maps to the letter A. This is because A's number is 0, and 0 multiplied by anything will always be 0. We can fix this by adding a second key that performs a Caesar cipher encryption after the multiplicative cipher's multiplication and modding is done.

This is called the affine cipher. The affine cipher has two keys. "Key A" is the integer that the letter's number is multiplied by. After modding this number by 26, "Key B" is the integer that is added to the number. This sum is also modded by 26, just like in the original Caesar cipher.

This means that the affine cipher has 26 times as many possible keys as the multiplicative cipher. It also ensures that the letter A does not always encrypt to the letter A.

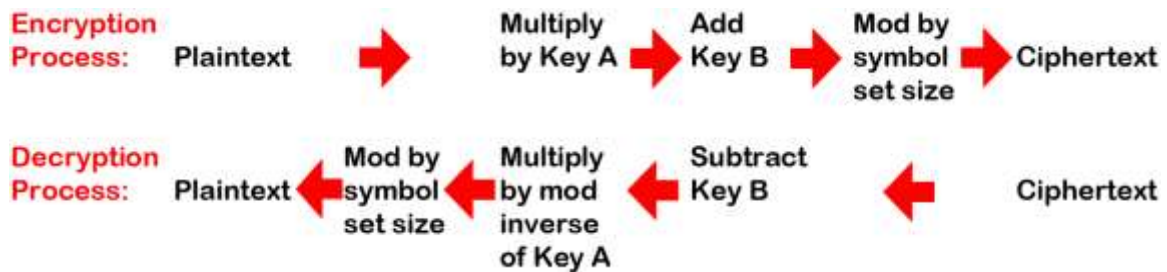


Figure 14-8. The encryption and decryption are mirrors of each other.

The First Affine Key Problem

There are two problems with the multiplicative cipher's key and affine cipher's Key A. You cannot just use any number for Key A. For example, if you chose the key 8, here is the mapping you would end up with:

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
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
A	I	Q	Y	G	O	W	E	M	U	C	K	S	A	I	Q	Y	G	O	W	E	M	U	C	K	S

This mapping doesn't work at all! Both the letters C and P encrypt to Q. When we encounter a Q in the ciphertext, how do we know which it decrypts to?! The same problem exists for encrypting A and N, F and S, and many others.

So some keys will work in the affine cipher while others will not. The secret to determining which key numbers will work is this:

In the affine cipher, the Key A number and the size of the symbol set must be relatively prime to each other. That is, $\text{gcd}(\text{key}, \text{size of symbol set}) == 1$.

We can use the `gcd()` function we wrote earlier to test this. The key 7 works as an affine cipher key because `gcd(7, 26)` returns 1. The larger key 8,953,851 will also work because `gcd(8953851, 26)` also returns 1. However, the key 8 did not work because `gcd(8, 26)` is 2. If the GCD of the key and the symbol set size is not 1, then they are not relatively prime and the key won't work.

The math we learned earlier sure is coming in handy now. We need to know how mod works because it is part of the GCD and affine cipher algorithms. And we need to know how GCD works because that will tell us if a pair of numbers is relatively prime. And we need to know if a pair of numbers is relatively prime or not in order to choose valid keys for the affine cipher.

The second problem with affine cipher's key is discussed in the next chapter.

Decrypting with the Affine Cipher

In the Caesar cipher, we used addition to encrypt and subtraction to decrypt. In the affine cipher, we use multiplication to encrypt. You might think that we need to divide to decrypt with the affine cipher. But if you try this yourself, you'll quickly see that it doesn't work. To decrypt with the affine cipher, we need to multiply by the key's modular inverse.

A **modular inverse** (which we will call i) of two numbers (which we will call a and m) is such that $(a * i) \% m == 1$. For example, let's find the modular inverse of "5 mod 7". There is some number i where $(5 * i) \% 7$ will equal "1". We will have to brute-force this calculation:

- 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 $(5 * 3) \% 7 = 1$.

The encryption key and decryption keys for the affine cipher are two different numbers. The encryption key can be anything we choose as long as it is relatively prime to 26 (which is the size of our symbol set). If we have chosen the key 7 for encrypting with the affine cipher, the decryption key will be the modular inverse of 7 mod 26:

- 1 is not the modular inverse of 7 mod 26, because $(7 * 1) \% 26 = 7$.
- 2 is not the modular inverse of 7 mod 26, because $(7 * 2) \% 26 = 14$.

- 3 is not the modular inverse of 7 mod 26, because $(7 * 3) \% 26 = 21$.
- 4 is not the modular inverse of 7 mod 26, because $(7 * 4) \% 26 = 2$.
- 5 is not the modular inverse of 7 mod 26, because $(7 * 5) \% 26 = 9$.
- 6 is not the modular inverse of 7 mod 26, because $(7 * 6) \% 26 = 16$.
- 7 is not the modular inverse of 7 mod 26, because $(7 * 7) \% 26 = 23$.
- 8 is not the modular inverse of 7 mod 26, because $(7 * 8) \% 26 = 4$.
- 9 is not the modular inverse of 7 mod 26, because $(7 * 9) \% 26 = 11$.
- 10 is not the modular inverse of 7 mod 26, because $(7 * 10) \% 26 = 18$.
- 11 is not the modular inverse of 7 mod 26, because $(7 * 11) \% 26 = 25$.
- 12 is not the modular inverse of 7 mod 26, because $(7 * 12) \% 26 = 6$.
- 13 is not the modular inverse of 7 mod 26, because $(7 * 13) \% 26 = 13$.
- 14 is not the modular inverse of 7 mod 26, because $(7 * 14) \% 26 = 20$.
- 15 is the modular inverse of 7 mod 26, because $(7 * 15) \% 26 = 1$.

So the affine cipher decryption key is 15. To decrypt a ciphertext letter, we take that letter's number and multiply it by 15, and then mod 26. This will be the number of the original plaintext's letter.

Finding Modular Inverses

In order to calculate the modular inverse to get the decryption key, we could take a brute-force approach and start testing the integer 1, and then 2, and then 3, and so on like we did above. But this will be very time-consuming for large keys like 8,953,851.

There is an algorithm for finding the modular inverse just like there was for finding the Greatest Common Divisor. Euclid's Extended Algorithm can be used to find the modular inverse of a number:

```
def findModInverse(a, m):
    if gcd(a, m) != 1:
        return None # no mod inverse exists 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 # // 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 in order to make use of it. We're just going to have our programs call this function. If you'd like to learn more about how it works, you can read <http://invpy.com/euclid>.

The // Integer Division Operator

You may have noticed the // operator used in the `findModInverse()` function above. This is the integer division operator. It divides two numbers and rounds down. Try typing the following into the interactive shell:

```
>>> 41 // 7
5
>>> 41 / 7
5.857142857142857
>>> 10 // 5
2
>>> 10 / 5
2.0
>>>
```

Notice that an expression with the // integer division operator always evaluates to an int, not a float.

Source Code of the `cryptomath` Module

The `gcd()` and `findModInverse()` functions will be used by more than one of our cipher programs later in this book, so we should put this code into a separate module. In the file editor, type in the following code and save it as `cryptomath.py`:

```
Source code for cryptomath.py
1. # Cryptomath Module
2. # http://inventwithpython.com/hacking (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.     # Returns 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 # // 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

```

The GCD algorithm is described earlier in this chapter. The `findModInverse()` function implements an algorithm called Euclid’s Extended Algorithm. How these functions work is beyond the scope of this book, but you don’t have to know how the code works in order to make use of it.

From the interactive shell, you can try out these functions after importing the module. Try typing 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
>>>

```

Practice Exercises, Chapter 14, Set E

Practice exercises can be found at <http://invpy.com/hackingpractice14E>.

Summary

Since the multiplicative cipher is the same thing as the affine cipher except using Key B of 0, we won’t have a separate program for the multiplicative cipher. And since it is just a less secure version of the affine cipher, you shouldn’t use it anyway. The source code to our affine cipher program will be presented in the next chapter.

The math presented in this chapter isn’t so hard to understand. Modding with the `%` operator finds the “remainder” between two numbers. The Greatest Common Divisor function returns the largest number that can divide two numbers. If the GCD of two numbers is 1, we say that those numbers are “relatively prime” to each other. The most useful algorithm to find the GCD of two numbers is Euclid’s Algorithm.

The affine cipher is sort of like the Caesar cipher, except it uses multiplication instead of addition to encrypt letters. Not all numbers will work as keys though. The key number and the size of the symbol set must be relatively prime towards each other.

To decrypt with the affine cipher we also use multiplication. To decrypt, the modular inverse of the key is the number that is multiplied. The modular inverse of “a mod m” is a number i such that $(a * i) \% m == 1$. To write a function that finds the modular inverse of a number, we use Euclid’s Extended Algorithm.

Once we understand these math concepts, we can write a program for the affine cipher in the next chapter.